Введение в теорию автоматов, языков и вычислений

2-Е ИЗДАНИЕ

Introduction to Automata Theory, Languages, and Computation

SECOND EDITION

JOHN E. HOPCROFT
Cornell University

RAJEEV MOTWANI
Stanford University

JEFFREY D. ULLMAN
Stanford University

ADDISON-WESLEY PUBLISHING COMPANY

Boston • San Francisco • New York
London • Toronto • Sydney • Tokyo • Singapore • Madrid
Mexico City • Munich • Paris • Cape Town • Hong Kong • Montreal

Введение в теорию автоматов, языков и вычислений

2-Е ИЗДАНИЕ

ДЖОН ХОПКРОФТ
РАДЖИВ МОТВАНИ
ДЖЕФФРИ УЛЬМАН

Москва • Санкт-Петербург • Киев
2008

ББК 32.973.26-018.2.75
Х78
УДК 681.3.07

Издательский дом “Вильямс”

Перевод с английского *О.И. Васылык*, *М. Саит-Аметова*,
канд.физ.-мат.наук *А.Б. Ставровского*

Под редакцией канд.физ.-мат.наук *А.Б. Ставровского*

По общим вопросам обращайтесь в Издательский дом “Вильямс”
по адресу: info@williamspublishing.com, http://www.williamspublishing.com

Хопкрофт, Джон, Э., **Мотвани**, Раджив, **Ульман**, Джеффри, Д..

Х78 Введение в теорию автоматов, языков и вычислений, 2-е изд.. : Пер. с англ. — М. : Издательский дом “Вильямс”, 2008. — 528 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1347-0 (рус.)

Книга известных американских ученых посвящена теории автоматов и соответствующих формальных языков и грамматик - как регулярных, так и контекстно-свободных. Во второй части рассматриваются различные машины Тьюринга, при помощи которых формализуются понятия разрешимых и неразрешимых проблем, а также определяются функции временной и емкостной оценки сложности алгоритмов. Изложение ведется строго, но доступно, и сопровождается многочисленными примерами, а также задачами для самостоятельного решения.

Книга будет полезна читателям различных категорий - студентам, аспирантам, научным сотрудникам, преподавателям высших учебных заведений, а также всем, кто интересуется математическими основами современной вычислительной техники.

ББК 32.973.26-018.2.75

ISBN 978-5-8459-1347-0 (рус.)
ISBN 0-201-44124-1 (англ.)

Оглавление

Содержание

Предисловие

В предисловии к своей книге 1979 года, предшествовавшей данному изданию, Дж. Хопкрофт и Дж. Ульман с удивлением отмечали, что за время, прошедшее после выхода их первой книги в 1969 году, произошел взрыв в развитии теории автоматов. Действительно, книга, вышедшая в 1979 году, содержала множество тем, не затронутых в предыдущей работе, и по объему была почти вдвое больше. Сравнив эту книгу с книгой 1979 года, вы увидите, что она, как автомобили 1970-х "больше снаружи, но меньше изнутри". И хотя это может показаться шагом назад, мы считаем такие изменения целесообразными в силу ряда причин.

Во-первых, в 1979 году теория автоматов и языков еще активно развивалась, и одной из целей той книги было пробудить интерес математически одаренных студентов к исследованиям в этой области. На сегодня в теории автоматов имеется лишь узкое направление для исследований (чего не скажешь о ее приложениях). Поэтому, на наш взгляд, не имеет смысла сохранять здесь лаконичный, сугубо математический стиль книги 1979 года.

Во-вторых, за последние двадцать лет изменилась роль теории автоматов и языков. В 1979 году это был сложный предмет, требующий от читателя высокого уровня подготовки. Читатель нашей книги, в особенности последних ее глав, представлялся нам хорошо подготовленным студентом-старшекурсником. Сегодня же этот предмет входит в стандартную вузовскую программу для младших курсов. Поэтому содержание книги должно быть по возможности доступным, а следовательно, содержать больше подробных доказательств и обоснований по сравнению с предыдущей книгой.

В-третьих, за два последних десятилетия информатика (Computer Science) невообразимо разрослась. И если в 1979 году вузовскую программу приходилось искусственно заполнять предметами, которые, как нам казалось, могли послужить новой волне развития технологий, то сегодня таких дисциплин уже слишком много.

В-четвертых, за эти годы информатика стала практическим предметом, и многие изучающие ее студенты настроены прагматически. Мы по-прежнему убеждены, что теория автоматов является мощным инструментом для исследований во множестве новых областей. А упражнения теоретического, развивающего плана, которые содержит обычный курс теории автоматов, сохраняют свое значение вне зависимости от того, предпочитает ли студент изучать лишь практические, "денежные" технологии. Однако для того, чтобы обеспечить данному предмету достойное место среди прочих дисциплин, изучаемых студентами-информатиками, нам представляется необходимым, наряду с математической частью, подчеркнуть и практические приложения теории. С этой целью мы заменили часть довольно сложных для понимания тем из предыдущей книги актуальными примерами применения этих идей на практике. В то время как приложения теории автоматов

и языков для компиляторов сегодня настолько понятны и естественны, что входят во всякий стандартный курс по компиляторам, существует немало и совершенно новых приложений. Например, алгоритмы проверки моделей, используемые для верификации протоколов, и языки описания документов, построенные по образцу контекстно-свободных грамматик.

И, наконец, последнее объяснение одновременного увеличения объема книги и уменьшения ее фактического содержания состоит в том, что при ее верстке использовались все преимущества издательских систем TEX и LATEX, которые поощряют "открытый" стиль печати, благодаря чему книга увеличивается в объеме, но становится удобнее для чтения. Мы с благодарностью отмечаем труд создателей этих издательских систем Дона Кнута и Леса Лампорта.

Как пользоваться книгой

Эта книга рассчитана на полусеместровый или семестровый курс лекций для студентов второго года обучения и выше. На ее основе в Стэндфордском университете Раджи-вом и Джеффом читается полусеместровый курс (CS154) по теории автоматов и языков. Ввиду ограниченности по времени в этом курсе не охвачены глава 11 и часть материала главы 10, например, довольно сложные вопросы о полиномиально-временно*й* сводимости. На Web-сайте книги (см. ниже) помещено несколько сокращенных вариантов этого курса с замечаниями.

Несколько лет назад мы столкнулись с тем, что многие студенты, поступившие в Стэнфорд после окончания колледжа, прошли курс теории автоматов, не содержавший теорию сложности. Преподавательский состав Стэнфорда полагает, что для всякого, кто серьезно занимается информатикой, эти идеи чрезвычайно важны. Тут недостаточно знать лишь то, что "для решения NP-полной задачи требуется очень много времени". Специально для таких студентов был разработан курс лекций (CS154N), который содержит материал только 8, 9 и 10 глав. Фактически, для того, чтобы освоить курс CS154N, они прослушивают лишь последнюю треть курса CS154. И по сегодняшний день на каждом курсе находится несколько студентов, желающих заниматься именно таким образом. Мы рекомендуем этот подход, поскольку он не требует чрезмерных усилий.

Требования к уровню подготовки

Чтение этой книги не вызовет затруднений у студентов, освоивших основы дискретной математики, в том числе изучивших графы, деревья, логику и методы доказательств. Кроме того, мы предполагаем, что читатель в достаточной степени знаком с программированием и, в частности, имеет представление об общих структурах данных, рекурсии и роли таких главных системных компонентов, как компиляторы. Эта сумма знаний соответствует стандартной программе первых двух лет обучения для студентов, изучающих информатику.

Упражнения

Книга содержит большое число упражнений, по несколько почти в каждом разделе. Упражнения или части упражнений повышенной сложности отмечены восклицательным знаком. Наиболее трудные из них отмечены двумя восклицательными знаками.

Некоторые упражнения отмечены звездочкой. Их решения мы намерены поместить на Web-странице книги. Они предназначены для самопроверки и доступны широкой аудитории. Отметим, что в некоторых случаях в одном упражнении *Б* требуется определенным образом изменить или адаптировать ваше решение другого упражнения *А*. Если какая-то часть упражнения *А* имеет решение, то естественно ожидать, что соответствующая часть упражнения *Б* также имеет решение.

Поддержка в World Wide Web

Адрес книги в Internet:

`http://www-db.stanford.edu/~ullman/ialc.html`

Здесь вы найдете решения заданий, отмеченных звездочкой, список замеченных опечаток и некоторые вспомогательные материалы. По мере чтения лекций по курсу CS154 мы постараемся размещать тут наши замечания по поводу домашних заданий, решений упражнений и экзаменов.

Благодарности

На часть материала главы 1 повлияла рукопись Крейга Сильверштейна (Creig Silverstein) о том, “как строить доказательства”. Мы благодарим также следующих лиц: Зое Абрамс (Zoe Abrams), Джордж Кандеа (George Candea), Хаовен Чен (Haowen Chen), Бьен-Ган Чан (Byong-Gun Chun), Джеффри Шаллит (Jeffrey Shallit), Брет Тейлор (Bret Taylor), Джейсон Таунсенд (Jason Townsend) и Эрик Узурео (Erik Uzureau), которые высказали свои замечания и указали на ряд опечаток в черновом варианте книги. Ошибки, оставшиеся незамеченными, авторы безусловно относят на свой счет.

Джон Хопкрофт
Раджив Мотвани
Джефри Ульман
Итака, Нью-Йорк и Стэнфорд, Калифорния
Сентябрь, 2000.

ГЛАВА 1

Автоматы: методы и понятия

Теория автоматов занимается изучением абстрактных вычислительных устройств, или "машин". В 1930-е годы, задолго до появления компьютеров, А. Тьюринг исследовал абстрактную машину, которая, по крайней мере в области вычислений, обладала всеми возможностями современных вычислительных машин. Целью Тьюринга было точно описать границу между тем, что вычислительная машина может делать, и тем, чего она не может. Полученные им результаты применимы не только к абстрактным *машинам Тьюринга*, но и к реальным современным компьютерам.

В 1940-х и 1950-х годах немало исследователей занимались изучением простейших машин, которые сегодня называются "конечными автоматами". Такие автоматы вначале были предложены в качестве модели функционирования человеческого мозга. Однако вскоре они оказались весьма полезными для множества других целей, которые будут упомянуты в разделе 1.1. А в конце 1950-х лингвист Н. Хомский занялся изучением формальных "грамматик". Не будучи машинами в точном смысле слова, грамматики, тем не менее, тесно связаны с абстрактными автоматами и служат основой некоторых важнейших составляющих программного обеспечения, в частности, компонентов компиляторов.

В 1969 году С. Кук развил результаты Тьюринга о вычислимости и невычислимости. Ему удалось разделить задачи на те, которые могут быть эффективно решены вычислительной машиной, и те, которые, в принципе, могут быть решены, но требуют для этого так много машинного времени, что компьютер оказывается бесполезным для решения практически всех экземпляров задачи, за исключением небольших. Задачи последнего класса называют "трудно разрешимыми" ("труднорешаемыми") или "NP-трудными". Даже при экспоненциальном росте быстродействия вычислительных машин ("закон Мура") весьма маловероятно, что нам удастся достигнуть значительных успехов в решении задач этого класса.

Все эти теоретические построения непосредственно связаны с тем, чем занимаются ученые в области информатики сегодня. Некоторые из введенных понятий, такие, например, как конечные автоматы и некоторые типы формальных грамматик, используются при проектировании и создании важных компонентов программного обеспечения. Другие понятия, например, машина Тьюринга, помогают нам уяснить принципиальные возможности программного обеспечения. В частности, теория сложности вычислений

позволяет определить, можем ли мы решить ту или иную задачу “в лоб” и написать соответствующую программу для ее решения (если эта задача не принадлежит классу “трудно разрешимых”), или же нам следует искать решение данной трудно разрешимой задачи в обход, используя приближенный, эвристический, или какой-либо другой метод, с помощью которого удастся ограничить время, затрачиваемое программой на ее решение.

В этой вводной главе дается обзор содержания теории автоматов и ее приложений. Основная часть главы посвящена рассмотрению различных методов доказательств и способов их нахождения. Рассматриваются дедуктивные доказательства, утверждения, получаемые переформулировкой, доказательства от противного, доказательства по индукции и другие важные понятия. В последней части вводится ряд основополагающих понятий теории автоматов: алфавиты, цепочки и языки.

1.1. Зачем изучается теория автоматов?

В силу ряда причин теория автоматов и теория сложности являются важнейшей частью ядра информатики. В этом разделе мы познакомим читателя с главными мотивами, побуждающими нас к их изучению, а также обрисуем в общих чертах основные темы, затрагиваемые в книге.

1.1.1. Введение в теорию конечных автоматов

Конечные автоматы являются моделью для многих компонентов аппаратного и программного обеспечения. Ниже (начиная со второй главы) мы рассмотрим примеры их использования, а сейчас просто перечислим наиболее важные из них.

1. Программное обеспечение, используемое для разработки и проверки цифровых схем.
2. “Лексический анализатор” стандартного компилятора, т.е. тот компонент компилятора, который отвечает за разбивку исходного текста на такие логические единицы, как идентификаторы, ключевые слова и знаки пунктуации.
3. Программное обеспечение для сканирования таких больших текстовых массивов, как наборы Web-страниц, с целью поиска заданных слов, фраз или других последовательностей символов (шаблонов).
4. Программное обеспечение для проверки различного рода систем (протоколы связи или протоколы для защищенного обмена информацией), которые могут находиться в конечном числе различных состояний.

С точными определениями автоматов различного типа мы встретимся вскоре. А начнем с краткого описания того, что представляет собой конечный автомат, и как он действует. Существует множество различных систем и их компонентов, например,

только что перечисленные, которые можно рассматривать, как находящиеся в каждый момент времени в одном из конечного числа “состояний”. Назначением каждого состояния является запоминание определенного момента истории системы. Поскольку число состояний конечно, то запомнить всю историю системы невозможно, а значит, необходимо строить систему тщательно, чтобы хранить только действительно важную информацию и забывать несущественную. Преимущество конечности числа состояний заключается в том, что систему можно реализовать, имея ограниченные ресурсы. Например, ее можно реализовать “в железе” как схему или же в виде простой программы, принимающей решения на основе ограниченного количества данных или текущей команды машинного кода.

Пример 1.1. Простейшим нетривиальным конечным автоматом является переключатель “включено-выключено”. Это устройство помнит свое текущее состояние, и от этого состояния зависит результат нажатия кнопки. Из состояния “выключено” нажатие кнопки переводит переключатель в состояние “включено”, и наоборот.

На рис. 1.1 представлена конечноавтоматная модель переключателя. Как и для всех конечных автоматов, состояния обозначены кружками. В данном случае они отмечены как “*вкл.*” и “*выкл.*”. Дуги между состояниями отмечены “входными символами”, задающими внешние воздействия на систему. В нашем случае это *Нажатие*, что означает нажатие на кнопку переключателя. Стрелки указывают, что всякий раз при *Нажатии* система переходит из одного состояния в другое.

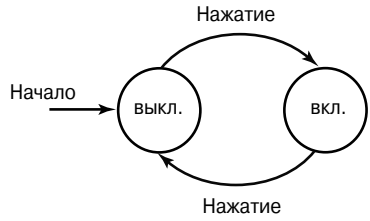


Рис. 1.1. Конечный автомат, моделирующий переключатель

Одно из состояний является так называемым “начальным состоянием”. Это состояние (в нашем случае — “*выкл.*”), в котором система находится изначально. На рисунке оно отмечено словом *Начало* и стрелкой, указывающей на это состояние.

Часто необходимо выделять одно или несколько “заключительных”, или “допускающих”, состояний. Попав в одно из них в результате реализации некоторой последовательности входных воздействий, мы можем считать такую входную последовательность в определенном смысле “хорошей”. Так, например, состояние “*вкл.*” на рис. 1.1 можно рассматривать как допускающее, поскольку если переключатель находится в этом состоянии, то устройство, управляемое им, находится в рабочем режиме. Допускающие состояния принято обозначать двойным кружком, хотя мы не использовали это обозначение на рис 1.1. □

Пример 1.2. Иногда состояние автомата запоминает гораздо более сложную информацию, чем выбор "*вкл.–выкл.*". На рис. 1.2 представлен конечный автомат, который может служить частью лексического анализатора. Его функцией является распознавание ключевого слова `then`. Этот автомат должен иметь пять различных состояний, каждое из которых представляет позицию в слове `then`, достигнутую на данный момент. Эти позиции соответствуют префиксам слова, от пустой цепочки (никакая позиция в слове еще не достигнута) до целого слова.

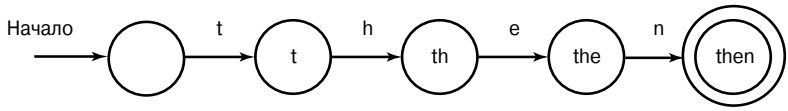


Рис. 1.2. Конечный автомат, моделирующий распознавание слова `then`

На рис. 1.2 каждое из пяти состояний обозначено частью слова, прочитанной на данном шаге. Входным сигналам соответствуют буквы. Мы можем считать, что данный лексический анализатор всякий раз просматривает по одному символу компилируемой программы. Каждый следующий символ рассматривается как входной сигнал для данного автомата. Начальное состояние автомата соответствует пустой цепочке, и каждое состояние имеет переход по очередной букве слова `then` в состояние, соответствующее следующему префиксу. Состояние, обозначенное словом "then", достигается, когда по буквам введено все данное слово. Поскольку функция автомата заключается в распознавании слова `then`, то последнее состояние будем считать единственным допускающим. □

1.1.2. Структурные представления

Следующие системы записи не являются автоматными, но играют важную роль в теории автоматов и ее приложениях.

1. *Грамматики*. Они являются полезными моделями при проектировании программного обеспечения, обрабатывающего данные рекурсивной структуры. Наиболее известный пример — "синтаксический анализатор". Этот компонент компилятора работает с такими рекурсивно вложенными конструкциями в типичных языках программирования, как выражения: арифметические, условные и т.п. Например, грамматическое правило типа $E \Rightarrow E + E$ означает, что выражение может быть получено соединением любых двух выражений с помощью знака "плюс". Это типичное правило построения выражений в существующих языках программирования. Ниже, в главе 5, будут определены так называемые контекстно-свободные грамматики.
2. *Регулярные выражения*. Они также задают структуру данных, в частности, текстовых цепочек. Как будет показано в главе 3, шаблоны описываемых ими цепочек представляют собой то же самое, что задают конечные автоматы. Стиль этих выражений суще-

ственно отличается от стиля, используемого в грамматиках. Ограничимся простым примером. Регулярное выражение в стиле UNIX `'[A-z][a-z]*[ ][A-Z][A-Z]'` представляет множество слов, начинающихся с прописной буквы, за которыми следуют пробел и две прописные буквы. В тексте такое выражение задает шаблоны, которые могут быть названиями города и штата, например: `Ithaca NY` (Итака, штат Нью-Йорк). В этом выражении не учтено, что название города может состоять из нескольких слов, к примеру: `Palo Alto CA` (Пало-Альто, штат Калифорния). Чтобы учесть этот случай, приходится использовать более сложное выражение: `'([A-Z][a-z]*[ ])*[ ][A-Z][A-Z]'`. Для интерпретации подобных выражений достаточно знать лишь то, что `[A-Z]` означает последовательность прописных букв английского алфавита от “A” до “Z” (т.е. любую прописную букву), а `[ ]` означает одиночный пробел. Кроме того, * значит “любое число вхождений” предшествующего выражения. Круглые скобки используются для группировки членов выражения и не являются символами описываемого текста.

1.1.3. Автоматы и сложность

Автоматы являются существенным инструментом исследования пределов вычислимости. Как мы уже упоминали в начале главы, существуют следующие две важные проблемы.

1. Что компьютер вообще может? Это вопрос “разрешимости”, а задачи, которые могут быть решены с помощью компьютера, называют “разрешимыми”. Этой теме посвящена глава 9.
2. Что компьютер может делать эффективно? Это вопрос “труднорешаемости” задач. Те задачи, на решение которых компьютеру требуется время, зависящее от размера входных данных как некоторая медленно растущая функция, называют “легко разрешимыми”. “Медленно растущими” функциями чаще всего считаются полиномиальные, а функции, которые растут быстрее, чем любой полином, считают растущими слишком быстро. Этот круг вопросов изучается в главе 10.

1.2. Введение в теорию формальных доказательств

Если вам довелось изучать планиметрию в школе до начала 1990-х, то, вероятнее всего, вам приходилось проводить подробные “дедуктивные доказательства” шаг за шагом, последовательно и детально обосновывая истинность некоторого утверждения. Поскольку геометрия имеет свою практическую сторону (например, если вы хотите приобрести нужное количество коврового покрытия для комнаты, то вам нужно знать правило вычисления площади прямоугольника), постольку и изучение общих методик формального доказательства в школе считалось весьма важным.

В 1990-х годах в США стало модным учить доказательствам, основанным на субъективных ощущениях относительно истинности утверждения. Конечно, неплохо уметь чувствовать истинность необходимого вам утверждения, однако плохо то, что в школе теперь не изучают весьма важные методики доказательств. А между тем, всякий, кто занимается информатикой, должен уметь понимать доказательства. Некоторые специалисты в области информатики придерживаются крайней точки зрения, полагая, что формальное доказательство корректности программы должно идти рука об руку с ее написанием. Продуктивность такого подхода вызывает сомнение. С другой стороны, есть и такие, которые считают, что в программировании как дисциплине нет места доказательствам. Среди них популярен девиз: "Если ты не уверен в правильности своей программы — запусти ее и убедись".

Мы занимаем промежуточную позицию по отношению к этим полярным подходам. Тестирование программ, безусловно, важно. Однако тестирование проходит до поры до времени, поскольку вы не в состоянии проверить работу вашей программы для всех возможных входных данных. Важнее, что, если ваша программа достаточно сложна (скажем, какая-нибудь хитрая рекурсия или итерация), то, не зная точно, что происходит при входе в цикл или рекурсивном вызове некоторой функции, вы вряд ли сумеете верно записать код. Получив сообщение об ошибке, вам все равно придется искать правильное решение.

Чтобы правильно выполнить рекурсию или итерацию, необходимо предварительно построить некую индуктивную гипотезу, причем полезно обосновать формально или неформально, что эта гипотеза остается истинной при рекурсии или итерации. В сущности, процедура уяснения механизма работы правильно написанной программы — это то же самое, что процедура доказательства по индукции теоремы. Поэтому наряду с моделями, полезными для различных видов программного обеспечения, в курсе теории автоматов традиционно изучают и методы формальных доказательств. Теория автоматов, пожалуй, в большей степени, чем все остальные предметы, лежащие в основе информатики, прибегает к естественным и содержательным доказательствам, как *дедуктивным* (состоящим из последовательности обоснованных шагов), так и *индуктивным*. Последние представляют собой рекурсивные доказательства параметризованных утверждений, в которых используется само утверждение с "меньшими" значениями параметра.

1.2.1. Дедуктивные доказательства

Как указывалось выше, дедуктивное доказательство состоит из последовательности утверждений, истинность которых следует из некоторого исходного утверждения, называемого *гипотезой*, или *данным утверждением* (*данными утверждениями*), или *посылкой*. Конечное утверждение этой цепочки называется *заключением*. Каждый шаг дедуктивного доказательства делается в соответствии с некоторыми допустимыми логически-

ми принципами на основании либо известных фактов, либо предыдущих утверждений, либо комбинации тех и других.

Исходная гипотеза может быть истинной или ложной. Обычно это зависит от значений входящих в нее параметров. Часто гипотеза содержит несколько независимых утверждений, связанных логическим союзом "И". В таких случаях каждое из этих утверждений считается гипотезой или данным утверждением.

Теорема, которую мы доказываем, переходя от гипотезы H к заключению C, является утверждением вида "если H, то C". Мы говорим, что *C логически (дедуктивно) следует из H*. Следующая теорема служит иллюстрацией утверждения данного типа.

Теорема 1.3. Если $x \geq 4$, то $2^x \geq x^2$. □

Формальное доказательство теоремы 1.3, основанное на индукции, мы рассмотрим в примере 1.17. Неформальное же доказательство этой теоремы особых усилий не требует. Для начала отметим, что гипотезой H является утверждение "$x \geq 4$". Эта гипотеза зависит от параметра x, а потому не является ни истинной, ни ложной. Истинность H зависит от значения параметра x: так, например, при $x = 6$ она верна, а при $x = 2$ — ложна.

Точно так же заключение C — это утверждение "$2^x \geq x^2$". Данное утверждение также зависит от параметра x и является истинным при одних значениях параметра и ложным — при других. Например, C ложно при $x = 3$, поскольку $2^3 = 8$ не превышает $3^2 = 9$. С другой стороны, C истинно при $x = 4$, так как $2^4 = 4^2 = 16$. Для $x = 5$ утверждение также истинно, поскольку $2^5 = 32$ не меньше, чем $5^2 = 25$.

Интуиция наверняка вам подсказывает, что утверждение $2^x \geq x^2$ истинно при $x \geq 4$. В его истинности при $x = 4$ мы уже убедились. Если $x > 4$ и x возрастает, то 2^x удваивается всякий раз, когда значение x увеличивается на 1. При этом выражение x^2, стоящее в правой части, увеличивается в $((x + 1)/x)^2$ раз. Если $x \geq 4$, то $((x + 1)/x)$ не превышает 1.25, следовательно, $((x + 1)/x)^2$ не превышает 1.5625. Поскольку $1.5625 < 2$, то при $x > 4$ с ростом x левая часть 2^x растет быстрее, чем правая x^2. Таким образом, если, начиная со значения x, при котором неравенство $2^x \geq x^2$ выполняется, например для $x = 4$, мы увеличиваем x, то неравенство остается верным.

Мы завершили неформальное, но вполне аккуратное доказательство теоремы 1.3. К более строгому доказательству этой теоремы мы еще вернемся в примере 1.17, после того как познакомимся с "индуктивными" доказательствами.

Как и все содержательные теоремы, теорема 1.3 охватывает бесконечное число связанных между собой фактов. В данном случае для всех целых x имеем "если $x \geq 4$, то $2^x \geq x^2$". На самом деле, необязательно предполагать, что x — целое. Но, поскольку в доказательстве теоремы говорится лишь об x, возрастающих на единицу, начиная с $x = 4$, то в действительности теорема доказана только для целых x.

Из теоремы 1.3 можно вывести ряд других теорем. В следующем примере мы рассмотрим полное дедуктивное доказательство простой теоремы, использующее теорему 1.3.

Теорема 1.4. Если x является суммой квадратов четырех положительных целых чисел, то $2^x \geq x^2$.

Доказательство. На интуитивном уровне идея доказательства состоит в том, что если предположение данной теоремы верно для некоторого x, то, поскольку x — сумма квадратов четырех положительных целых чисел, значение x по крайней мере не меньше 4. Но тогда выполнено условие теоремы 1.3, а поскольку мы считаем, что она верна, то мы можем утверждать, что и заключение этой теоремы справедливо для x. Рассуждения можно представить в виде последовательности шагов, каждый из которых является либо гипотезой доказываемой теоремы, либо ее частью, либо утверждением, которое следует из одного или нескольких предыдущих утверждений.

Под словом "следует" мы подразумеваем, что, если предыдущим утверждением является гипотеза какой-либо теоремы, то заключение этой теоремы верно и может быть записано, как утверждение в нашем доказательстве. Это логическое правило часто называют правилом *modus ponens*. Оно гласит, что, если известно, что утверждение H истинно и утверждение "если H, то C" истинно, то можно заключить, что C также истинно. Кроме того, при выводе утверждений из одного или нескольких предыдущих утверждений допустимы и другие логические шаги. Так, например, если A и B — два предыдущих утверждения, то мы можем вывести и записать утверждение "A и B".

На рис. 1.3 приведена вся последовательность утверждений, необходимых для доказательства теоремы 1.4. Отметим, что мы не собираемся доказывать теоремы в такой стилизованной форме, хотя она помогает представить доказательство в виде явного перечня строго обоснованных утверждений. На шаге (1) мы повторяем одну из посылок теоремы: x есть сумма квадратов четырех целых чисел. В доказательствах часто оказывается полезным как-то обозначать величины. Здесь четыре целых числа обозначены как a, b, c и d.

	Утверждение	Обоснование
1.	$x = a^2 + b^2 + c^2 + d^2$	Посылка
2.	$a \geq 1$; $b \geq 1$; $c \geq 1$; $d \geq 1$	Посылка
3.	$a^2 \geq 1$; $b^2 \geq 1$; $c^2 \geq 1$; $d^2 \geq 1$	(2) и свойства арифметики
4.	$x \geq 4$	(1), (3) и свойства арифметики
5.	$2^x \geq x^2$	(4) и теорема 1.3

Рис. 1.3. Формальное доказательство теоремы 1.4

На шаге 2 записана еще одна часть предположения теоремы: каждое из чисел, возводимых в квадрат, не меньше 1. Технически, это утверждение содержит в себе четыре отдельных утверждения для каждого из данных четырех целых чисел. Затем, на шаге 3, за-

мечаем, что квадрат числа, не меньшего 1, также не меньше 1. В качестве обоснования используется истинность утверждения 2 и "свойства арифметики", т.е. мы предполагаем, что читатель знает или может самостоятельно вывести простые утверждения с неравенствами, такие как: "если $y \geq 1$, то $y^2 \geq 1$".

На шаге 4 используются утверждения 1 и 3. В первом из них говорится, что x есть сумма четырех квадратов, а во втором — что каждый из квадратов не меньше 1. Воспользовавшись известными свойствами арифметики, заключаем, что минимальное значение x равно 1 + 1 + 1 + 1, т.е. 4.

На последнем шаге 5 используем утверждение 4, которое является гипотезой теоремы 1.3. Теорема же сама по себе есть основание, благодаря которому мы можем выписать ее заключение, поскольку предыдущее утверждение является ее предположением. Так как утверждение 5, т.е. заключение теоремы 1.3, является также заключением теоремы 1.4, то теорема 1.4 доказана. Таким образом, исходя из предположения теоремы, нам удалось вывести ее заключение. □

1.2.2. Сведение к определениям

В двух предыдущих теоремах использовались такие хорошо известные понятия, как целые числа, сложение и умножение. Во многих других теоремах, в том числе во многих теоремах теории автоматов, понятия, используемые в утверждениях, могут быть не столь очевидными. Для многих доказательств оказывается полезным следующее правило.

- Если вы не знаете, с чего начать доказательство, то замените все понятия, входящие в гипотезу, их определениями.

Приведем пример теоремы, которую легко доказать, переписав ее утверждение в элементарных терминах. В ней используются следующие определения.

1. Множество S называется *конечным*, если существует целое число n, и S содержит ровно n элементов. Мы пишем $||S|| = n$, где $||S||$ обозначает число элементов во множестве S. Если множество S не является конечным, то его называют *бесконечным*. Интуитивно бесконечное множество можно представлять как множество, число элементов которого больше любого целого числа.

   - Если множества S и T являются подмножествами некоторого множества U, то говорят, что T есть *дополнение* S (относительно множества U), если $S \bigcup T = U$ и $S \bigcap T = \varnothing$. Таким образом, всякий элемент U содержится в одном, и только в одном, из множеств S или T. Другими словами, в T содержатся те, и только те, элементы U, которые не содержатся в S.

Теорема 1.5. Пусть S — конечное подмножество бесконечного множества U, и пусть T — дополнение S относительно U. Тогда T — бесконечное множество.

Доказательство. Интуитивно утверждение теоремы гласит, что, если имеется бесконечный запас чего-либо (U), и из него изымается конечное количество (S), то оставшееся

содержимое по-прежнему бесконечно. Для начала перепишем положения теоремы так, как показано на рис. 1.4.

Исходное утверждение	Новое утверждение
S конечно	Существует целое n и $\|S\| = n$
U бесконечно	Не существует целого p, при котором $\|U\| = p$
T является дополнением S	$S \cup T = U$ и $S \cap T = \varnothing$

Рис. 1.4. Переформулировка положений теоремы 1.5

Чтобы сдвинуться с места в нашем доказательстве, мы должны применить общий метод, называемый "доказательством от противного". Применяя этот метод, который обсуждается в разделе 1.3.3, мы предполагаем, что заключение теоремы ложно. Затем, основываясь на этом предположении и отдельных утверждениях гипотезы теоремы, доказываем утверждение, являющееся отрицанием какого-либо из утверждений гипотезы. Этим мы показываем, что невозможно, чтобы одновременно все части гипотезы были истинными, а заключение — ложным. Остается единственная возможность — сделать вывод, что заключение истинно, если истинна гипотеза, т.е. что теорема верна.

В теореме 1.5 отрицанием заключения является "T — конечное множество". Предположим, что T — конечно, вместе с утверждением гипотезы о конечности S, т.е. $\|S\| = n$ при некотором целом n. Наше предположение можно переформулировать в виде "$\|T\| = m$ для некоторого целого числа m".

Одна из посылок утверждает, что $S \cup T = U$ и $S \cap T = \varnothing$, т.е. каждый элемент U принадлежит в точности одному из множеств S или T. Но тогда в U должно содержаться $n + m$ элементов. Поскольку $n + m$ — целое число, и, как показано, $\|U\| = n + m$, то U — конечное множество. Точнее, мы показали, что число элементов U есть целое число, что соответствует определению конечного множества. Но утверждение, что U — конечно, противоречит условию теоремы, согласно которому U — бесконечное множество. Таким образом, предположив противное заключению теоремы, мы получили противоречие одному из данных утверждений ее гипотезы. Согласно принципу "доказательства от противного" приходим к выводу, что теорема верна. □

Доказательства не обязательно должны быть столь подробными. Зная идею доказательства, мы можем теперь записать его в несколько строк.

Доказательство (теоремы 1.5). Известно, что $S \cup T = U$ и множества S и T не пересекаются, а потому $\|S\| + \|T\| = \|U\|$. Поскольку S — конечное множество, то $\|S\| = n$ для некоторого целого числа n, а из того, что U — бесконечное множество, следует, что не существует такого целого числа p, для которого $\|U\| = p$. Допустим, что множество T — конечное, т.е. $\|T\| = m$ для некоторого целого m. Тогда $\|U\| = \|S\| + \|T\| = m + n$, что противоречит посылке — утверждению, что не существует целого числа, равного $\|U\|$. □

1.2.3. Другие формы теорем

Во многих разделах математики наиболее распространены теоремы вида “если-то”. Однако приходится доказывать в виде теорем и другие типы утверждений. В этом разделе мы изучим основные схемы таких утверждений и способы их доказательств.

Разновидности утверждений типа „если-то“

Существует множество видов теорем, утверждения которых, на первый взгляд, отличаются от простого “если H, то C”, но на самом деле означают то же самое, а именно: если гипотеза H верна при данном значении параметра (параметров), то при этом значении верно и заключение C. Вот несколько возможных способов записи утверждений типа “если H, то C”.

1. H влечет C.
2. H только тогда, когда C.
3. C, если H.
4. Из H следует C.

Форма 4 имеет множество разновидностей, например: “если H, следовательно, C” или “если верно H, то C также верно”.

Пример 1.6. Утверждение теоремы 1.3 может быть записано в следующих четырех формах.

1. $x \geq 4$ влечет $2^x \geq x^2$.
2. $x \geq 4$ только тогда, когда $2^x \geq x^2$.
3. $2^x \geq x^2$, если $x \geq 4$.
4. Из $x \geq 4$ следует $2^x \geq x^2$.

□

Утверждения с кванторами

Во многих теоремах присутствуют утверждения с *кванторами* “для всех” и “существует” или их вариациями, например, “для каждого” вместо “для всех”. От того, в каком порядке кванторы входят в утверждение, зависит его смысл. Часто оказывается полезным представлять утверждения с кванторами как “игру”, в которой участвуют два игрока — “Для всех” и “Существует”. Они по очереди определяют значения параметров теоремы. Игрок “Для всех” должен рассматривать все существующие возможности, поэтому параметры, на которые он действует, всегда остаются переменными. Игроку “Существует”, напротив, достаточно выбрать лишь одно значение, которое может зависеть от значений параметров, выбранных игроками ранее. Право первого хода определяется порядком вхождения кванторов в данное утверждение. Утверждение верно, если игрок, делающий ход последним, всегда может выбрать подходящее значение параметра.

В качестве примера рассмотрим альтернативное определение “бесконечного множества”: множество S *бесконечно* тогда и только тогда, когда для каждого целого числа n существует подмножество T множества S, содержащее ровно n элементов. Здесь “Для каждого” предшествует “Существует”, поэтому мы должны рассматривать произвольное целое n. Затем “Существует”, используя информацию об n, выбирает подмножество T. Так, если S — множество всех целых чисел, то “Существует” может выбрать подмножество $T = \{1, 2, \ldots, n\}$, удовлетворив таким образом требование независимо от n. Тем самым доказано, что множество целых чисел бесконечно.

Следующее утверждение, похожее на определение бесконечного множества, *некорректно*, поскольку кванторы в него входят в обратном порядке: “существует подмножество T множества S, при котором для всякого n множество T содержит ровно n элементов”. Теперь нам дано множество S, например множество целых чисел, и игрок “Существует” может выбрать произвольное его подмножество T, например $\{1, 2, 5\}$. Относительно данного множества “Для всех” должен доказать, что при любом n оно содержит n элементов. Но это невозможно, поскольку при $n = 4$, и вообще, для любого $n \neq 3$ это утверждение ложно.

В дополнение скажем, что в формальной логике вместо оборота “если-то” часто используется оператор $\rightarrow$. Таким образом, вместо выражения “если H, то C” в математической литературе встречается запись $H \rightarrow C$. Далее в книге этот оператор используется для других целей.

Утверждения типа „тогда и только тогда“

Иногда мы встречаемся с выражениями вида “A тогда и только тогда, когда B”. Разновидностями этого утверждения являются “A эквивалентно B”, “A равносильно B” или “A в точности тогда, когда B”[1]. На самом деле такое утверждение содержит два утверждения типа “если-то”: “если A, то B” и “если B, то A”. Чтобы доказать, что “A тогда и только тогда, когда B”, необходимо доказать оба эти утверждения.

1. *Достаточность* (B для A), или “если”-часть: “если B, то A”, т.е. “A тогда, когда B”.
2. *Необходимость* (B для A), или “только-если”-часть: “если A, то B”, т.е. “A только тогда, когда B”.

Какими должны быть формальные доказательства?

Ответить на этот вопрос не так просто. Все доказательства имеют одну цель — убедить кого-либо, будь то человек, проверяющий вашу работу, или вы сами, в кор-

[1] В англоязычной математической литературе вместо выражения “if and only if” часто используется его краткое обозначение — “iff”. Это утверждение еще называют *эквивалентностью*. — *Прим. ред.*

ректности выбранной вами стратегии. Если доказательство убедительно, то этого вполне достаточно. Если же оно вызывает сомнения у "потребителя", значит оно недостаточно подробно.

Неопределенность в степени подробности доказательств обусловлена различными уровнями знаний у его возможного потребителя. Так, при доказательстве теоремы 1.4 мы предполагали, что читатель знает арифметику, и что такое утверждение, как "если $y \geq 1$, то $y^2 \geq 1$", не вызовет у него сомнений. Если бы он не был знаком с арифметикой, то нам бы пришлось доказывать это утверждение и, соответственно, добавлять в наше дедуктивное доказательство несколько дополнительных пунктов.

Существуют, однако, определенные требования к доказательству, которыми никак нельзя пренебрегать, иначе оно будет некорректным. Например, нельзя считать корректным дедуктивное доказательство, если оно содержит утверждение, не доказанное исходя из предыдущих или данных утверждений. Затем при доказательстве утверждений типа "тогда и только тогда" мы, конечно же, должны проводить их и в одну, и в другую сторону. Наконец, в индуктивных доказательствах (обсуждаемых в разделе 1.4) мы должны доказывать базисное утверждение и индуктивную часть.

Доказательство можно проводить в любом порядке. Во многих теоремах одна часть значительно проще другой. Обычно ее доказывают вначале, чтобы потом на нее не отвлекаться.

В формальной логике для обозначения утверждений типа "тогда и только тогда" встречаются операторы $\leftrightarrow$ и $\equiv$. Следовательно, запись $A \leftrightarrow B$ или $A \equiv B$ означает то же, что и "*A* тогда и только тогда, когда *B*".

Доказывая утверждения типа "тогда и только тогда", важно помнить, что следует доказывать обе его части — и необходимость, и достаточность. Иногда оказывается полезным разбить его на ряд нескольких эквивалентностей. Таким образом, чтобы доказать, что "*A* тогда и только тогда, когда *B*", вы можете вначале доказать что "*A* тогда и только тогда, когда *C*", а затем, что "*C* тогда и только тогда, когда *B*". Еще раз подчеркнем, что, применяя этот метод, обязательно нужно доказывать и необходимость, и достаточность. Доказав подобное утверждение лишь в одну сторону, мы тем самым оставляем доказательство незавершенным.

Приведем простой пример доказательства теоремы типа "тогда и только тогда". Введем следующие обозначения.

1. $\lfloor x \rfloor$ обозначает наибольшее целое число, меньшее или равное вещественному числу *x*.
2. $\lceil x \rceil$ обозначает наименьшее целое число, которое больше или равно вещественному числу *x*.

Теорема 1.7. Пусть *x* — вещественное число. $\lfloor x \rfloor = \lceil x \rceil$ тогда и только тогда, когда *x* — целое.

Доказательство. (*Необходимость*) В этой части мы предполагаем, что $\lfloor x \rfloor = \lceil x \rceil$, и попытаемся доказать, что x — целое число. Заметим, что по определению $\lfloor x \rfloor \leq x$ и $\lceil x \rceil \geq x$. Нам дано, что $\lfloor x \rfloor = \lceil x \rceil$. Поэтому мы можем изменить в первом неравенстве $\lfloor x \rfloor$ на $\lceil x \rceil$. Поскольку верны оба неравенства $\lceil x \rceil \leq x$ и $\lceil x \rceil \geq x$, то согласно свойствам арифметических неравенств заключаем, что $\lceil x \rceil = x$. Поскольку число $\lceil x \rceil$ всегда целое, x тоже целое.

(*Достаточность*) Предположим теперь, что x — целое, и попытаемся доказать, что $\lfloor x \rfloor = \lceil x \rceil$. Эта часть доказывается легко. По определению, $\lfloor x \rfloor$ и $\lceil x \rceil$ при целом x оба равны x, а следовательно, равны между собой. □

1.2.4. Теоремы без гипотезы

Иногда встречаются теоремы, которые, на первый взгляд, не имеют гипотезы. Пример хорошо известен из тригонометрии.

Теорема 1.8. $\sin^2 \theta + \cos^2 \theta = 1$. □

На самом деле у этой теоремы *есть* гипотеза, состоящая из тех утверждений, которые необходимо знать, чтобы понять смысл этого утверждения. В частности, здесь неявно предполагается, что θ — это некоторый угол, и потому функции синус и косинус имеют обычный смысл. Исходя из определений членов этого равенства и теоремы Пифагора (в прямоугольном треугольнике квадрат гипотенузы равен сумме квадратов двух других сторон), вы можете доказать эту теорему. На самом деле, она имеет вид утверждения типа "если-то": "если θ — угол, то $\sin^2 \theta + \cos^2 \theta = 1$".

1.3. Дополнительные схемы доказательств

В этом разделе мы рассмотрим следующие дополнительные вопросы, касающиеся методов доказательств.

1. Доказательства утверждений о множествах.
2. Доказательства методом "от противного".
3. Доказательства с помощью контрпримера.

1.3.1. Доказательства эквивалентностей, связанных с множествами

В теории автоматов нам нередко приходится доказывать, что два множества, записанные разными способами, на самом деле равны. Часто эти множества состоят из цепочек символов и являются так называемыми "языками". Но в данном разделе природа множеств не будет играть роли. Если E и F — выражения, представляющие некоторые множества, то утверждение $E = F$ означает, что эти множества равны. Точнее, каждый элемент множества, представленного E, принадлежит множеству, представленному F, и наоборот.

Пример 1.9. *Коммутативный закон объединения множеств* утверждает, что, объединяя два множества, мы можем делать это в любом порядке, т.е. $R \cup S = S \cup R$. В дан-

ном случае в качестве E выступает выражение $R \cup S$, а в качестве F — $S \cup R$. Согласно коммутативному закону $E = F$. □

Равенство $E = F$ можно переписать, как необходимое и достаточное условие: произвольный элемент x принадлежит множеству E тогда и только тогда, когда x принадлежит F. Следовательно, доказательство для равенств множеств имеет такую же структуру, как и для утверждений типа "тогда и только тогда".

1. Доказать, что если x принадлежит E, то x принадлежит и F.
2. Доказать, что если x принадлежит F, то x принадлежит и E.

В качестве примера рассмотрим доказательство *закона дистрибутивности объединения относительно пересечения*.

Теорема 1.10. $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$.

Доказательство. Мы имеем дело со следующими выражениями: $E = R \cup (S \cap T)$ и $F = (R \cup S) \cap (R \cup T)$.

Докажем по очереди обе части теоремы. Для доказательства необходимости предположим, что x принадлежит E, и покажем, что тогда x принадлежит F. Эта часть доказательства представлена на рис. 1.5. При этом мы используем определения объединения и пересечения множеств, предполагая, что читатель с ними знаком.

	Утверждение	Обоснование
1.	x принадлежит $R \cup (S \cap T)$	Посылка
2.	x принадлежит R или x принадлежит $S \cap T$	(1) и определение объединения
3.	x принадлежит R или x принадлежит как S, так и T	(1) и определение пересечения
4.	x принадлежит $R \cup S$	(3) и определение объединения
5.	x принадлежит $R \cup T$	(3) и определение объединения
6.	x принадлежит $(R \cup S) \cap (R \cup T)$	(4), (5) и определение пересечения

Рис. 1.5. Доказательство необходимости теоремы 1.10

Затем нужно доказать достаточность. Тут мы предполагаем, что x принадлежит F, и показываем, что тогда x принадлежит E. Эта часть доказательства представлена на рис. 1.6. Доказав обе части утверждения (и необходимость, и достаточность), мы тем самым доказали закон дистрибутивности объединения относительно пересечения. □

	Утверждение	Обоснование
1.	x принадлежит $(R \cup S) \cap (R \cup T)$	Посылка
2.	x принадлежит $R \cup S$	(1) и определение пересечения
3.	x принадлежит $R \cup T$	(1) и определение пересечения
4.	x принадлежит R или x принадлежит как S, так и T	(2), (3) и рассуждения о множествах
5.	x принадлежит R или x принадлежит $S \cap T$	(4) и определение пересечения
6.	x принадлежит $R \cup (S \cap T)$	(5) и определение объединения

Рис. 1.6. Доказательство достаточности теоремы 1.10

1.3.2. Контрапозиция

Всякое утверждение типа “если-то” может быть записано в эквивалентной форме, что подчас облегчает его доказательство. Утверждение “если не C, то не H” является *обратным противоположному* для утверждения “если H, то C”, или его *контрапозицией*. Утверждение и его контрапозиция либо оба истинны, либо оба ложны. Поэтому, доказав одно из них, мы доказываем и другое.

Чтобы показать, почему именно утверждения “если H, то C” и “если не C, то не H” логически равносильны, рассмотрим следующие четыре случая.

1. H и C оба истинны.
2. H истинно, а C ложно.
3. C истинно, а H ложно.
4. H и C оба ложны.

“Тогда и только тогда” для множеств

Как мы уже упоминали, теоремы, которые устанавливают эквивалентность выражений для множеств, являются утверждениями типа “тогда и только тогда”. Таким образом, утверждение теоремы 1.10 может быть записано в виде: “элемент x принадлежит $R \cup (S \cap T)$ тогда и только тогда, когда x принадлежит $(R \cup S) \cap (R \cup T)$”.

Эквивалентность множеств можно выразить иначе с помощью оборота “все те, и только те”. Например, утверждение теоремы 1.10 может быть записано в таком виде: “элементами множества $R \cup (S \cap T)$ являются все те, и только те элементы, которые принадлежат $(R \cup S) \cap (R \cup T)$”.

Утверждение типа “если-то” может быть ложным лишь в одном случае — когда гипотеза истинна, а заключение ложно, что соответствует случаю (2). В остальных трех случаях, включая и случай (4), в котором заключение ложно, данное утверждение типа “если-то” остается истинным.

Теперь рассмотрим, когда ложно утверждение, обратное противоположному, т.е. “если не C, то не H”. Для того чтобы это утверждение было ложным, его гипотеза (“не C”) должна быть истинной, а заключение (“не H”) — ложным. Но “не C” истинно именно тогда, когда C — ложно, а “не H” ложно именно тогда, когда H — истинно. А это и есть случай (2). Последнее означает, что в каждом из четырех возможных случаев утверждение и его контрапозиция одновременно либо истинны, либо ложны, т.е. они логически эквивалентны.

Обратное утверждение (конверсия)

Не следует путать понятия “утверждение, обратное противоположному” (контрапозиция) и “обратное утверждение” (конверсия). *Конверсия* утверждения типа “если-то” (или *обратное утверждение*) есть то же утверждение, прочитанное “в обратную сторону”. Следовательно, конверсией утверждения “если H, то C” является утверждение “если C, то H”. В отличие от контрапозиции, конверсия не является логически эквивалентной исходному утверждению. На самом деле, части утверждения типа “тогда и только тогда” всегда представляют собой некоторое утверждение и его конверсию.

Пример 1.11. Вспомним теорему 1.3, в которой утверждалось: “если $x \geq 4$, то $2^x \geq x^2$”. Для данного утверждения обратное противоположному есть: “если не $2^x \geq x^2$, то не $x \geq 4$”. Говоря обычным языком и приняв во внимание, что “не $a \geq b$” означает $a < b$, можно записать эту контрапозицию таким образом: “если $2^x < x^2$, то $x < 4$”. □

Использование контрапозиции при доказательстве теорем типа “тогда и только тогда” дает нам несколько дополнительных возможностей. Представьте, например, что необходимо доказать эквивалентность множеств $E = F$. Тогда в утверждении “x принадлежит E тогда и только тогда, когда x принадлежит F” мы можем одну из частей заменить ее контрапозицией. Одна из возможных форм такова.

- Если x принадлежит E, то x принадлежит F, а если x не принадлежит E, то x не принадлежит F.

В последнем утверждении E и F можно также поменять местами.

1.3.3. Доказательство методом „от противного“

Еще один способ доказать утверждение типа “если-то” состоит в том, чтобы доказать утверждение

- “H и не C влечет ложь”.

Следовательно, следует вначале предположить, что верны гипотеза *H* и отрицание заключения *C*. Затем нужно доказать, что из *H* и "не *C*" следует некоторое заведомо ложное утверждение. Эта схема доказательства называется доказательством "от противного".

Пример 1.12. Вспомним теорему 1.5, в которой было доказано утверждение типа "если-то" с гипотезой *H* = "*U* — бесконечное множество, *S* — конечное подмножество *U*, *T* — дополнение *S* относительно *U*" и заключением *C* = "*T* — бесконечное множество". Мы доказывали эту теорему методом "от противного": предполагая "не *C*", т.е., что *T* — конечное множество, пытались вывести некоторое ложное утверждение. Мы показали, что если *S* и *T* — оба конечны, то и *U* должно быть конечным. Но, поскольку согласно гипотезе *H U* — бесконечное множество, и быть одновременно конечным и бесконечным множество не может, то полученное утверждение ложно. Выражаясь терминами логики, мы имеем некоторое утверждение *p* (*U* — конечно) и его отрицание "*не p*" (*U* — бесконечно). Затем используем тот факт, что утверждение "*p* и *не p*" всегда ложно. □

Обоснуем теперь корректность метода доказательства "от противного" с точки зрения логики. Вспомним раздел 1.3.2, где мы показали, что из четырех возможных комбинаций значений истинности *H* и *C* только во втором случае утверждение "если *H*, то *C*" ложно. Из ложности *H* и *не C* следует, что случай 2 невозможен. Таким образом, возможны лишь оставшиеся три комбинации, и для каждой из них утверждение "если *H*, то *C*" истинно.

1.3.4. Контрпримеры

В обыденной жизни нам не приходится доказывать теорем. Однако довольно часто мы сталкиваемся с чем-то, что кажется нам верным — например, со стратегией реализации программы, и мы вынуждены задумывается над истинностью такого рода "теоремы". Чтобы решить эту проблему, можно попытаться сначала доказать ее истинность, а если это не удастся, то обосновать ее ложность.

Теоремы, вообще говоря, являются утверждениями, включающими бесконечное число случаев, соответствующих множеству значений входящих в них параметров. В математике существует строгое соглашение о том, что утверждение можно назвать "теоремой" лишь тогда, когда оно описывает бесконечное число случаев. Те же утверждения, в которые параметры либо вовсе не входят, либо они могут принимать лишь конечное число значений, называют *наблюдениями*. Для того чтобы доказать, что предполагаемая теорема неверна, достаточно показать, что она ложна хотя бы в одном случае. Схожая ситуация имеет место и с программами, поскольку мы обычно считаем, что программа содержит ошибку, если она неверно обрабатывает хотя бы один набор входных данных, с которым она должна работать.

Часто бывает легче доказать, что утверждение не является теоремой, чем доказать, что оно — теорема. Мы уже упоминали о том, что, если *S* — некоторое утверждение, то

утверждение “S — не теорема” само по себе уже не содержит параметров, а потому является наблюдением, а не теоремой.

В следующих двух примерах первое утверждение, очевидно, не является теоремой, а второе лишь похоже на теорему, однако требуется некоторое дополнительное исследование, чтобы доказать, что оно — не теорема.

Мнимая теорема 1.13. Все простые числа нечетны. (Более строго: если целое число x является простым, то x — число нечетное.)

Опровержение. 2 — простое число, но 2 четно. □

Обсудим теперь “теорему”, в которой используются элементы арифметической теории делимости. Но сначала дадим следующее важное определение.

Если a и b — натуральные числа, то $a \bmod b$ — это остаток от деления a на b, т.е. такое целое число r между 0 и $b - 1$, что $a = qb + r$, где q — некоторое целое число. Например, $8 \bmod 3 = 2$, а $9 \bmod 3 = 0$. Докажем, что следующая теорема неверна.

Мнимая теорема 1.14. Не существует такой пары целых чисел a и b, для которой $a \bmod b = b \bmod a$. □

Оперируя парами объектов, как a и b в данном случае, мы зачастую можем упростить отношения между объектами, используя симметричность. Так, мы можем подробно рассмотреть лишь случай $a < b$, поскольку если $a > b$, то a и b можно просто поменять местами. При этом в теореме 1.14 получим то же равенство. Но необходимо быть осторожным и не забыть о третьем случае, когда $a = b$. Именно здесь кроется подвох в попытке доказательства теоремы.

Итак, предположим, что $a < b$. Тогда $a \bmod b = a$, поскольку в определении остатка $a \bmod b$ для этого случая имеем $q = 0$ и $r = a$, т.е. $a = 0 \times b + a$, когда $a < b$. Но $b \bmod a < a$, так как любой остаток от деления на a есть число между 0 и $a - 1$. Таким образом, если $a < b$, то $b \bmod a < a \bmod b$, так что равенство $a \bmod b = b \bmod a$ невозможно. Используя теперь приведенные выше рассуждения о симметричности, получаем, что $a \bmod b \neq b \bmod a$ и в случае, когда $b < a$.

Однако есть еще третий случай, когда $a = b$. Поскольку для любого целого x выполнено $x \bmod x = 0$, то при $a = b$ имеем $a \bmod b = b \bmod a$. Этим предполагаемая теорема опровергнута.

Опровержение мнимой теоремы 1.14. Пусть $a = b = 2$, тогда $a \bmod b = b \bmod a = 0$. □

В процессе поиска контрпримера мы на самом деле нашли точные условия, при выполнении которых предполагаемая теорема верна. Ниже приведена правильная формулировка этой теоремы и ее доказательство.

Теорема 1.15. $a \bmod b = b \bmod a$ тогда и только тогда, когда $a = b$.

Доказательство. (*Достаточность*) Предположим, что $a = b$. Тогда, как было отмечено выше, $x \bmod x = 0$ для любого целого x. Таким образом, $a \bmod b = b \bmod a = 0$, если $a = b$.

(*Необходимость*) Предположим теперь, что $a \bmod b = b \bmod a$. Лучше всего в данном случае применить метод "от противного", и предположить, что заключение неверно, т.е. что $a \neq b$. Тогда, поскольку вариант $a = b$ исключается, остается рассмотреть случаи $a < b$ и $b < a$.

Выше мы выяснили, что если $a < b$, то $a \bmod b = a$ и $b \bmod a < a$. Эти утверждения совместно с гипотезой $a \bmod b = b \bmod a$ приводят к противоречию. По свойству симметричности, если $b < a$, то $b \bmod a = b$ и $a \bmod b < b$. Снова получаем противоречие. Таким образом, необходимость также доказана. Итак, теорема верна, поскольку мы доказали ее в обе стороны. □

1.4. Индуктивные доказательства

При оперировании с рекурсивно определенными объектами мы сталкиваемся с необходимостью использовать в доказательствах особый метод, называемый "индуктивным". Большинство известных индуктивных доказательств оперирует с целыми числами, но в теории автоматов нам приходится индуктивно доказывать утверждения о таких рекурсивно определяемых понятиях, как деревья и разного рода выражения, например, регулярные, о которых мы уже вкратце упоминали в разделе 1.1.2. В этом разделе будут рассмотрены индуктивные доказательства сначала на примере "простых" индукций для целых чисел. Затем мы покажем, как проводить "структурную" индукцию для любых рекурсивно определяемых понятий.

1.4.1. Индукция по целым числам

Пусть требуется доказать некое утверждение $S(n)$, которое зависит от целого числа n. Существует общий подход к доказательствам такого рода. Доказательство разбивается на следующие два этапа.

1. *Базис*. На этом этапе мы показываем, что $S(i)$ верно для некоторого частного целого значения i. Обычно полагают $i = 0$ или $i = 1$, но существуют примеры, где выбирается большее начальное значение, например, когда для всех меньших значений i утверждение S ложно.
2. *Индуктивный переход*. Здесь мы предполагаем, что $n \geq i$, где i — целое число из базиса индукции, и доказываем, что из истинности $S(n)$ следует истинность $S(n + 1)$, т.е. доказываем утверждение "если $S(n)$, то $S(n + 1)$".

На уровне здравого смысла данное доказательство убеждает нас в том, что $S(n)$ на самом деле верно для всякого целого n, большего или равного базисному i. Обосновать это можно следующим образом. Предположим, что $S(n)$ ложно для одного или нескольких таких целых n. Тогда среди этих значений n существует наименьшее, скажем, j, причем $j \geq i$. Значение j не может быть равным i, так как на основании базиса индукции $S(i)$

истинно. Поэтому j должно быть строго больше i. Таким образом, мы знаем, что $j - 1 \geq i$ и $S(j - 1)$ истинно.

Но во второй части доказательства показано, что если $n \geq i$, то $S(n)$ влечет $S(n + 1)$. Предположим, что $n = j - 1$. Тогда, совершая индуктивный переход, из $S(j - 1)$ выводим $S(j)$. Следовательно, из истинности $S(j - 1)$ следует истинность $S(j)$.

Но мы предполагали, что справедливо отрицание доказываемого утверждения, а именно: $S(j)$ ложно для некоторого $j \geq i$. Придя к противоречию, мы тем самым доказали методом "от противного", что $S(n)$ истинно для всех $n \geq i$.

К сожалению, в приведенных рассуждениях присутствует трудноуловимый логический изъян. Дело в том, что первый пункт нашего предположения о том, что мы можем выбрать наименьшее $j \geq i$, для которого $S(j)$ ложно, зависит от того, принимаем ли мы на веру справедливость принципа индукции. Это значит, что по сути единственный способ доказать существование такого j есть индуктивное доказательство. Но с позиций здравого смысла приведенное нами "доказательство" вполне осмысленно, и соответствует нашему пониманию мира. В связи с этим мы присоединим к нашей логической системе следующий закон.

- *Принцип индукции*: если доказано, что $S(i)$ верно, и что при всех $n \geq i$ из $S(n)$ следует $S(n + 1)$, значит, $S(n)$ истинно при всех $n \geq i$.

Покажем на следующих двух примерах, как принцип индукции используется при доказательстве теорем о целых числах.

Теорема 1.16. Для всех $n \geq 0$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \tag{1.1}$$

Доказательство. Доказательство будет состоять из двух частей: базиса и индуктивного шага. Докажем их по очереди.

Базис. В качестве базисного значения выберем $n = 0$. В левой части равенства (1.1) стоит $\sum_{i=1}^{0}$, поэтому может показаться, что при $n = 0$ утверждение теоремы не имеет смысла. Однако существует общий принцип, согласно которому, если верхний предел суммы (в данном случае 0) меньше нижнего предела (здесь 1), то сумма не содержит слагаемых и равна 0. Это значит, что $\sum_{i=1}^{0} i^2 = 0$.

Правая часть равенства (1.1) также равна нулю, поскольку $0 \times (0 + 1) \times (2 \times 0 + 1)/6 = 0$. Таким образом, при $n = 0$ равенство (1.1) выполняется.

Индукция. Предположим теперь, что $n \geq 0$. Мы должны совершить индуктивный переход, т.е. доказать, что, изменив в равенстве (1.1) n на $n + 1$, мы получим также правильное соотношение. Оно имеет следующий вид.

$$\sum_{i=1}^{[n+1]} i^2 = \frac{[n+1]([n+1]+1)(2[n+1]+1)}{6} \tag{1.2}$$

Можно упростить равенства (1.1) и (1.2), раскрыв скобки в правых частях. Равенства примут следующий вид.

$$\sum_{i=1}^{n} i^2 = (2n^3 + 3n^2 + n)/6 \tag{1.3}$$

$$\sum_{i=1}^{n+1} i^2 = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.4}$$

Нам нужно доказать (1.4), используя (1.3). Эти равенства соответствуют утверждениям $S(n + 1)$ и $S(n)$ из принципа индукции. “Фокус” заключается в том, чтобы разбить сумму для $n + 1$ в левой части (1.4) на сумму для n плюс $(n + 1)$-й член. После чего мы сможем убедиться, что (1.4) верно, заменив сумму первых n членов левой частью равенства (1.3). Запишем эти действия.

$$\left(\sum_{i=1}^{n} i^2\right) + (n+1)^2 = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.5}$$

$$(2n^3 + 3n^2 + n)/6 + (n^2 + 2n + 1) = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.6}$$

Наконец, чтобы убедиться в справедливости (1.6), нужно преобразовать левую часть этого равенства в правую с помощью несложных алгебраических действий. □

Пример 1.17. В этом примере мы докажем теорему 1.3 из пункта 1.2.1. Напомним, в этой теореме утверждается следующее: если $x \geq 4$, то $2^x \geq x^2$. Мы уже приводили неформальное доказательство этой теоремы, основной идеей которого являлось то, что отношение $x^2/2^x$ уменьшается с ростом x, когда $x > 4$. Мы придадим строгость нашим рассуждениям, доказав утверждение $2^x \geq x^2$ по индукции, начиная с базисного значения $x = 4$. Отметим, кстати, что при $x < 4$ утверждение неверно.

Базис. Если $x = 4$, то и 2^x, и x^2 равны 16. Следовательно, нестрогое неравенство $2^x \geq x^2$ выполнено.

Индукция. Предположим, что $2^x \geq x^2$ для некоторого $x \geq 4$. Приняв это утверждение в качестве гипотезы, мы должны доказать, что верно и следующее утверждение $2^{(x+1)} \geq (x + 1)^2$, где вместо x стоит $x + 1$. Эти два утверждения соответствуют $S(x)$ и $S(x + 1)$ в принципе индукции. Тот факт, что мы в качестве параметра используем не n, а x, не имеет значения, это всего лишь обозначение локальной переменной.

Как и в теореме 1.16, мы должны переписать $S(x + 1)$ так, чтобы можно было использовать $S(x)$. В данном случае мы можем записать $2^{(x+1)}$ как 2×2^x. $S(x)$ утверждает, что $2^x \geq x^2$, поэтому можно заключить, что $2^{x+1} = 2\times2^x \geq 2x^2$.

Но нам нужно показать нечто иное, а именно: $2^{(x+1)} \geq (x + 1)^2$. Это можно сделать, доказав, например, что $2x^2 \geq (x + 1)^2$, а затем, использовав транзитивность отношения $\geq$, показать, что $2^{(x+1)} \geq 2x^2 \geq (x + 1)^2$. Доказывая, что

$$2x^2 \geq (x + 1)^2, \tag{1.7}$$

можно использовать предположение, что $x \geq 4$. Для начала упростим (1.7):

$$x^2 \geq 2x + 1. \tag{1.8}$$

Разделив обе части (1.8) на x, получим:

$$x \geq 2 + \frac{1}{x}. \tag{1.9}$$

Поскольку $x \geq 4$, то $1/x \leq 1/4$. Таким образом, минимальное значение выражения в левой части (1.9) равно 4, а максимальное значение выражения в правой — 2.25. Итак, истинность (1.9) доказана. Следовательно, верны (1.8) и (1.7). В свою очередь, (1.7) влечет $2x^2 \geq (x + 1)^2$ при $x \geq 4$, что позволяет доказать утверждение $S(x + 1)$, т.е. $2^{x+1} \geq (x + 1)^2$. □

Целые числа как рекурсивно определяемые понятия

Мы уже упоминали о том, что индуктивные доказательства особенно полезны при работе с рекурсивно определяемыми объектами. Но в первых же примерах мы столкнулись с индукцией по целым числам, которые нами, как правило, не воспринимаются как объекты, определяемые рекурсивно. Однако существует естественное рекурсивное определение неотрицательного целого числа. Это определение вполне соответствует индуктивной процедуре по целым числам: переходу от ранее определенных объектов к тем, что определяются позже.

Базис. 0 есть целое число.

Индукция. Если n — целое число, то $n + 1$ — тоже целое число.

1.4.2. Более общие формы целочисленных индуктивных доказательств

В разделе 1.4.1 мы описали схему индуктивного доказательства по целым числам, где утверждение S доказывается вначале для некоторого базисного значения, а затем доказывается "если $S(n)$, то $S(n + 1)$". Однако иногда индуктивное доказательство возможно лишь при использовании более общих схем. Рассмотрим следующие два важных обобщения.

1. Можно использовать несколько базисных значений. Это значит, что мы доказываем $S(i)$, $S(i + 1)$, …, $S(j)$ для некоторого $j > i$.
2. При доказательстве $S(n + 1)$ можно использовать истинность не только $S(n)$, но также и всех утверждений $S(i)$, $S(i + 1)$, …, $S(n)$. Кроме того, если доказана истинность S для базисных значений вплоть до j, то можно предполагать не только $n \geq i$, но и $n \geq j$.

На основании доказательств такого базиса и индуктивного шага можно заключить, что $S(n)$ истинно для всех $n \geq i$.

Пример 1.18. Возможности обоих описанных выше принципов проиллюстрируем на примере следующего утверждения $S(n)$: "если $n \geq 8$, то n можно представить в виде суммы троек и пятерок". Отметим, между прочим, что 7 нельзя представить в таком виде.

Базис. В качестве базисных утверждений выберем $S(8)$, $S(9)$ и $S(10)$. Доказательства их соответственно таковы: $8 = 3 + 5$, $9 = 3 + 3 + 3$ и $10 = 5 + 5$.

Индукция. Предположим, что $n \geq 10$ и $S(8)$, $S(9)$, …, $S(n)$ истинны. Используя эти факты, мы должны доказать, что истинно $S(n + 1)$. Для этого вычтем сначала 3 из $n + 1$, заметим, что полученная разность должна быть представима в виде суммы троек и пятерок, а затем прибавим к этой сумме 3 и получим сумму для $n + 1$.

Более строго вышесказанное выглядит так. Поскольку $n - 2 \geq 8$, то можно сделать предположение об истинности $S(n - 2)$, т.е. $n - 2 = 3a + 5b$ для некоторых целых a и b. Но тогда $n + 1 = 3 + 3a + 5b$, и, значит, мы можем записать $n + 1$ в виде суммы $a + 1$ троек и b пятерок. Это позволяет нам завершить индуктивный шаг и доказывает истинность утверждения $S(\mathrm{n} + 1)$. □

1.4.3. Структурная индукция

В теории автоматов используется несколько рекурсивно определяемых понятий, относительно которых нам будет необходимо доказывать те или иные утверждения. Важными примерами таких понятий являются деревья и выражения. Подобно индукции, все рекурсивные определения включают базис, где определяется одна или несколько элементарных структур, и индуктивный шаг, с помощью которого более сложные структуры определяются через структуры, определенные ранее.

Пример 1.19. Рассмотрим рекурсивное определение дерева.

Базис. Одиночный узел есть дерево, и этот узел является *корнем* дерева.

Индукция. Если T_1, T_2, …, T_k — деревья, то можно построить новое дерево следующим образом.

1. Возьмем в качестве корня новый узел N.
2. Возьмем по одному экземпляру деревьев T_1, T_2,…, T_k.
3. Добавим ребра, соединяющие корень N, с корнями каждого из деревьев T_1, T_2, …, T_k.

Индуктивное построение дерева с корнем N из k меньших деревьев представлено на рис. 1.7. □

Пример 1.20. Определим рекурсивно понятие выражения, использующего арифметические операции + и *. В качестве его операндов могут выступать как числа, так и переменные.

Базис. Всякое число или буква (т.е. переменная) есть выражение.

Индукция. Пусть E и F — некоторые выражения, тогда $E + F$, $E * F$ и (E) также являются выражениями.

Например, 2 и x являются выражениями согласно базису. Индуктивный шаг позволяет утверждать, что $x + 2$, $(x + 2)$ и $2*(x + 2)$ — тоже выражения. Отметим, что каждое из них состоит из частей, в свою очередь являющихся выражениями. □

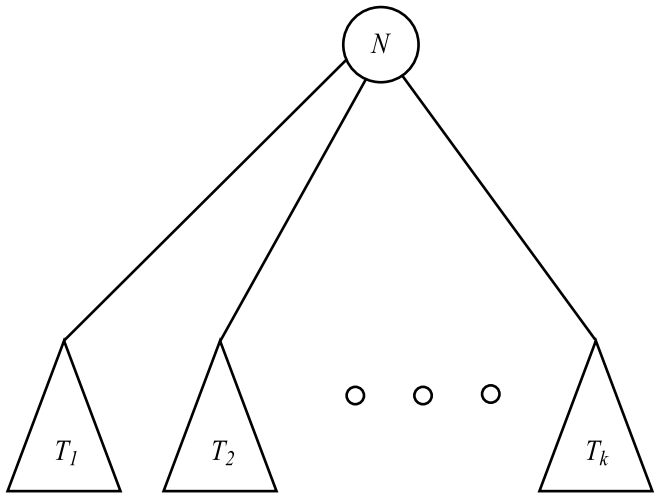


Рис. 1.7. Индуктивное построение дерева

Неформальное обоснование структурной индукции

Можно неформально обосновать правомочность структурной индукции как метода доказательства. Представим, что в процессе рекурсивного определения мы последовательно вводим некоторые структуры X_1, X_2, …. Первыми в этой последовательности идут базисные элементы, и структура X_i определена, если только все ее подструктуры предшествуют X_i в этой последовательности. С этой точки зрения структурная индукция — ни что иное, как обычная индукция по n для утверждения $S(X_n)$. В частности, это может быть и обобщенная индукция, описанная в разделе 1.4.2, т.е. в ней могут использоваться базисные утверждения, а индуктивный переход может опираться на все предыдущие утверждения. Однако следует помнить, что, как и в разделе 1.4.1, данное апеллирующее к интуиции пояснение не является формальным доказательством. Фактически, мы должны допустить справедливость этого принципа, так же, как в том разделе допустили справедливость исходного принципа индукции.

Если у нас имеется рекурсивное определение некоторого понятия, то теоремы относительно этого понятия можно доказывать с помощью следующего метода, называемого *структурной индукцией*. Пусть $S(X)$ — утверждение о структурах X, определяемых рекурсивно.

1. В качестве базиса докажем утверждение $S(X)$ для базисной структуры (или структур) X.
2. Для индуктивного перехода возьмем структуру X, определенную рекурсивно через $Y_1, Y_2, \ldots, Y_k$. Предполагая, что верны утверждения $S(Y_1), S(Y_2), \ldots, S(Y_k)$, докажем с их помощью $S(X)$.

Отсюда заключаем, что $S(X)$ истинно для всех X. В следующих двух примерах мы доказываем теоремы о деревьях и выражениях.

Теорема 1.21. В любом дереве число узлов на единицу больше числа ребер.

Доказательство. Запишем утверждение $S(T)$, которое нам требуется доказать методом структурной индукции в формальном виде: "Если T — дерево, и T содержит n узлов и e ребер, то $n = e + 1$".

Базис. В качестве базисного выберем случай, когда дерево T состоит из одного узла. Тогда $n = 1$ и $e = 0$, а потому соотношение $n = e + 1$ выполнено.

Индукция. Пусть T — дерево, построенное по индуктивному определению из корневого узла N и k меньших деревьев $T_1, T_2, \ldots, T_k$. Предположим, что утверждение $S(T_i)$ истинно при $i = 1, 2, ..., k$, т.е., если дерево T_i содержит n_i узлов и e_i ребер, то $n_i = e_i + 1$.

Узлы дерева T — это узел N и все узлы деревьев T_i. Таким образом, T содержит $1 + n_1 + n_2 + \ldots + n_k$ узлов. Ребра T — это k ребер, которые мы добавили на индуктивном шаге определения, плюс все ребра деревьев T_i. Значит, T содержит

$$k + e_1 + e_2 + ... + e_k \tag{1.10}$$

ребер. Заменив n_i на $e_i + 1$ в выражении для числа узлов T, получим, что оно равно

$$1 + [e_1 + 1] + [e_2 + 1] + ... + [e_k + 1]. \tag{1.11}$$

Поскольку в сумме (1.11) содержится k слагаемых вида "+1", ее можно перегруппировать таким образом.

$$k + 1 + e_1 + e_2 + ... + e_k \tag{1.12}$$

Это выражение равно на единицу больше, чем значение выражения (1.10), показывающего число ребер T. Таким образом, число узлов на единицу больше числа ребер. □

Теорема 1.22. Всякое выражение содержит поровну правых и левых скобок.

Доказательство. Говоря формально, необходимо доказать такое утверждение $S(G)$ относительно выражения G, определенного рекурсивно в примере 1.20: левых и правых скобок в G поровну.

Базис. Если G — базисное выражение, то это либо число, либо переменная. В этих выражениях 0 левых и 0 правых скобок, т.е. поровну.

Индукция. По определению индуктивного шага существует три способа построения выражения G.

1. $G = E + F$.
2. $G = E * F$.

3. $G = (E)$.

Мы предполагаем, что утверждения $S(E)$ и $S(F)$ истинны, т.е. что каждое из выражений E и F содержит поровну правых и левых скобок, по n и m, соответственно. Тогда в каждом из трех случаев мы можем подсчитать число входящих в G правых и левых скобок.

1. Если $G = E + F$, то G содержит по $n + m$ скобок каждого сорта: по n левых и правых скобок из выражения E, и по m — из F.
2. Если $G = E * F$, то G также содержит по $n + m$ скобок каждого сорта, как и в случае (1).
3. Если же $G = (E)$, то G содержит $n + 1$ левых скобок — одну мы видим явно, а n находятся в E. Аналогично G содержит $n + 1$ правых скобок (одна явная и n в E).

Итак, в каждом из этих трех случаев число правых и левых скобок в G одинаково, а это значит, что теорема доказана. □

1.4.4. Совместная индукция

Иногда мы не можем доказать по индукции некоторое отдельно взятое утверждение, и вместо этого нам нужно доказать совместно целую группу утверждений $S_1(n)$, $S_2(n)$, …, $S_k(n)$ с помощью индукции по n. В теории автоматов такая ситуация встречается довольно часто. Так, в примере 1.23 мы рассмотрим общую ситуацию, в которой действие автомата описывается группой утверждений, по одному для каждого из состояний. В этих утверждениях говорится, какие последовательности входных сигналов приводят автомат в каждое из его состояний.

Строго говоря, доказательство группы утверждений не отличается от доказательства *конъюнкции* (логическое “И”) всех этих утверждений. Например, группу утверждений $S_1(n)$, $S_2(n)$, ..., $S_k(n)$ можно заменить одним утверждением $S_1(n)$ **И** $S_2(n)$ **И** … **И** $S_k(n)$. Однако, если необходимо доказывать несколько действительно независимых утверждений, то проще рассматривать их отдельно и для каждого из них доказывать свой базис и индуктивный шаг. Этот тип доказательства назовем *совместной индукцией*. В следующем примере мы покажем, какие основные этапы должно содержать такое доказательство.

Пример 1.23. Вернемся к примеру 1.1, где в виде автомата был представлен переключатель состояний “*вкл.-выкл.*”. Сам автомат еще раз воспроизведен на рис. 1.8. Нажатие кнопки переводит автомат из одного состояния в другое, а в качестве начального выбрано состояние “*выкл.*”, поэтому можно предполагать, что поведение переключателя описывается системой из следующих двух утверждений.

$S_1(n)$: после n нажатий кнопки автомат находится в состоянии “*выкл.*” тогда и только тогда, когда n — четное число.

$S_2(n)$: после n нажатий кнопки автомат находится в состоянии “*вкл.*” тогда и только тогда, когда n — нечетное число.

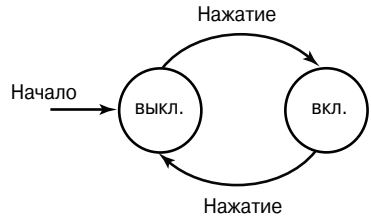


Рис. 1.8. Повтор автомата с рис. 1.1

Поскольку число n не может быть одновременно и четным, и нечетным, то можно предположить, что из S_1 следует S_2, и наоборот. Однако, вообще говоря, то, что некоторый автомат находится в одном и только в одном состоянии, не всегда верно. На самом деле автомат на рис. 1.8 находится всегда лишь в одном из состояний, но этот факт нужно доказывать как часть совместной индукции.

Ниже приводятся доказательства базисов и индуктивных шагов для утверждений $S_1(n)$ и $S_2(n)$. В доказательствах используются некоторые свойства четных и нечетных чисел, а именно: если к четному числу прибавить 1 или вычесть из него 1, то получим число нечетное, а если то же самое проделать с нечетным числом, то получим четное.

Базис. В качестве базисного значения выберем $n = 0$. Поскольку требуется доказать два утверждения типа "тогда и только тогда", причем каждое из них в обе стороны, то фактически нужно рассмотреть четыре базисных случая и четыре шага индукции.

1. [S_1; *достаточность*] Поскольку 0 — четное число, мы должны показать, что после 0 нажатий автомат на рис. 1.8 находится в состоянии "*выкл.*". Но это действительно так, поскольку начальное состояние автомата — "*выкл.*".
2. [S_1; *необходимость*] После 0 нажатий автомат находится в состоянии "*выкл.*", поэтому нам нужно показать, что 0 — четное число. Но тут и доказывать нечего, поскольку 0 есть четное число по определению.
3. [S_2; *достаточность*] Гипотеза достаточности S_2 состоит в том, что 0 — нечетное число. Очевидно, она ложна. А поскольку гипотеза H ложна, то, как показано в разделе 1.3.2, всякое утверждение вида "если H, то C" истинно. Таким образом, и эта часть базиса верна.
4. [S_2; *необходимость*] Гипотеза о том, что после 0 нажатий автомат находится в состоянии "*вкл.*" также ложна, так как в это состояние мы попадаем только по стрелке "*Нажатие*", а это означает, что на кнопку нажали хотя бы один раз. Отсюда, как и ранее, приходим к выводу, что утверждение типа "если-то" истинно, поскольку его гипотеза ложна.

Индукция. Предположим теперь, что $S_1(n)$ и $S_2(n)$ истинны, и докажем утверждения $S_1(n + 1)$ и $S_2(n + 1)$. Это доказательство также разделяется на следующие четыре части.

1. [$S_1(n+1)$; *достаточность*] Гипотезой для этой части является то, что $n+1$ — четное число, т.е. n — нечетное. В этом случае из достаточности утверждения $S_2(n)$ следует, что после n нажатий автомат находится в состоянии "*вкл.*". Дуга из "*вкл.*" в "*выкл.*", обозначенная сигналом "*Нажатие*", указывает, что $(n+1)$-е нажатие переводит автомат в состояние "*выкл.*". Таким образом, доказана достаточность утверждения S_1.
2. [$S_1(n+1)$; *необходимость*] Предположим, что после $(n+1)$-го нажатия автомат находится в состоянии "*выкл.*". Тогда, анализируя автомат на рис. 1.8, можно сделать вывод, что в состояние "*выкл.*" мы попадаем, находясь в состоянии "*вкл.*" и получая на вход "*Нажатие*". Таким образом, если после $(n+1)$-го нажатия мы находимся в состоянии "*выкл.*", то после n нажатий мы находились в состоянии "*вкл.*". Но тогда из необходимости утверждения $S_2(n)$ заключаем, что n — нечетное число. Следовательно, $n+1$ — число четное, а это и есть нужное нам заключение "необходимости" в утверждении $S_1(n+1)$.
3. [$S_2(n+1)$; *достаточность*] Для доказательства этой части достаточно в пункте (1) поменять местами утверждения S_2 и S_1, а также слова "четное" и "нечетное". Не сомневаемся, что читатель самостоятельно восстановит это доказательство.
4. [$S_2(n+1)$; *необходимость*] Для этого доказательства достаточно в п. 2 поменять местами S_2 и S_1, а также "нечетное" и "четное".

□

На основе примера 1.23 можно сделать следующие общие выводы относительно совместных индуктивных доказательств.

- Для каждого утверждения нужно отдельно доказать и базис, и индуктивный шаг.
- Если утверждения имеют вид "тогда и только тогда", то при доказательстве и базиса, и шага индукции утверждения нужно доказывать в обе стороны.

1.5. Основные понятия теории автоматов

В этом разделе вводятся наиболее важные из понятий, которыми оперирует теория автоматов: "алфавит" (множество символов), "цепочка" (последовательность символов некоторого алфавита) и "язык" (множество цепочек в одном и том же алфавите).

1.5.1. Алфавиты

Алфавитом называют конечное непустое множество символов. Условимся обозначать алфавиты символом Σ. Наиболее часто используются следующие алфавиты.

1. $\Sigma = \{0,1\}$ — *бинарный* или двоичный алфавит.
2. $\Sigma = \{a, b, \ldots, z\}$ — множество строчных букв английского алфавита.

3. Множество ASCII-символов или множество всех печатных ASCII-символов.

1.5.2. Цепочки

Цепочка, или иногда *слово*, — это конечная последовательность символов некоторого алфавита. Например, 01101 — это цепочка в бинарном алфавите $\Sigma = \{0, 1\}$. Цепочка 111 также является цепочкой в этом алфавите.

Пустая цепочка

Пустая цепочка — это цепочка, не содержащая ни одного символа. Эту цепочку, обозначаемую ε, можно рассматривать как цепочку в любом алфавите.

Длина цепочки

Часто оказывается удобным классифицировать цепочки по их *длине*, т.е. по числу позиций для символов в цепочке. Например, цепочка 01101 имеет длину 5. Обычно говорят, что длина цепочки — это "число символов" в ней. Это определение широко распространено, но не вполне корректно. Так, в цепочке 01101 всего 2 символа, но число *позиций* в ней — пять, поэтому она имеет длину 5. Все же следует иметь в виду, что часто пишут "число символов", имея в виду "число позиций".

Длину некоторой цепочки w обычно обозначают $|w|$. Например, $|011| = 3$, а $|\varepsilon| = 0$.

Степени алфавита

Если Σ — некоторый алфавит, то можно выразить множество всех цепочек определенной длины, состоящих из символов данного алфавита, используя знак степени. Определим Σ^k как множество всех цепочек длины k, состоящих из символов алфавита Σ.

Пример 1.24. Заметим, что $\Sigma^0 = \{\varepsilon\}$ независимо от алфавита Σ, т.е. ε — единственная цепочка длины 0.

Если $\Sigma = \{0, 1\}$, то $\Sigma^1 = \{0, 1\}$, $\Sigma^2 = \{00, 01, 10, 11\}$, $\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$ и так далее. Отметим, что между Σ и Σ^1 есть небольшое различие. Дело в том, что Σ есть алфавит, и его элементы 0 и 1 являются символами, а Σ^1 является множеством цепочек, и его элементы — это цепочки 0 и 1, каждая длиной 1. Мы не будем вводить разные обозначения для этих множеств, полагая, что из контекста будет понятно, является $\{0, 1\}$ или подобное ему множество алфавитом или же множеством цепочек. □

Соглашения о символах и цепочках

Как правило, строчными буквами из начальной части алфавита (или цифрами) мы будем обозначать символы, а строчными буквами из конца алфавита, например w, x, y или z — цепочки. Руководствуясь этим соглашением, вы легко сможете понять, элементы какого типа рассматриваются в том или ином случае.

Множество всех цепочек над алфавитом Σ принято обозначать Σ^*. Так, например, $\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$. По-другому это множество можно записать в виде

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$$

Иногда нам будет необходимо исключать из множества цепочек пустую цепочку. Множество всех непустых цепочек в алфавите Σ обозначают через Σ^+. Таким образом, имеют место следующие равенства:

- $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$
- $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.

Конкантенация цепочек

Пусть x и y — цепочки. Тогда xy обозначает их *конкатенацию* (соединение), т.е. цепочку, в которой последовательно записаны цепочки x и y. Более строго, если x — цепочка из i символов: $x = a_1a_2\ldots a_i$, а y — цепочка из j символов: $y = b_1b_2\ldots b_j$, то xy — это цепочка длины $i + j$: $xy = a_1a_2\ldots a_ib_1b_2\ldots b_j$.

Пример 1.25. Пусть x = 01101 и y = 110. Тогда xy = 01101110, а yx = 11001101. Для любой цепочки w справедливы равенства $\varepsilon w = w\varepsilon = w$. Таким образом, ε является *единицей* (*нейтральным элементом*) *относительно операции конкантенации*, поскольку результат ее конкатенации с любой цепочкой дает ту же самую цепочку (аналогично тому как 0, нейтральный элемент относительно сложения, при сложении с любым числом x дает число x). □

1.5.3. Языки

Множество цепочек, каждая из которых принадлежит Σ^*, где Σ — некоторый фиксированный алфавит, называют *языком*[2]. Если Σ — алфавит, и $L \subseteq \Sigma^*$, то L — это *язык над* Σ, или *в* Σ. Отметим, что язык в Σ не обязательно должен содержать цепочки, в которые входят все символы Σ. Поэтому, если известно, что L является языком в Σ, то можно утверждать, что L — это язык над любым алфавитом, содержащим Σ.

Термин “язык” может показаться странным. Однако оправданием служит то, что и обычные языки можно рассматривать как множества цепочек. Возьмем в качестве примера английский язык, где набор всех литературных английских слов есть множество цепочек в алфавите (английских же букв). Еще один пример — язык программирования C или любой другой язык программирования, в котором правильно написанные программы представляют собой подмножество множества всех возможных цепочек, а цепочки состоят из символов алфавита данного языка. Этот алфавит является подмножеством символов ASCII. Алфавиты для разных языков программирования могут быть раз-

[2] В таком определении язык часто называется *формальным*. — *Прим. ред.*

личными, хотя обычно они состоят из прописных и строчных букв, цифр, знаков пунктуации и математических символов.

Существует, однако, множество других языков, изучаемых в теории автоматов. Приведем несколько примеров.

1. Язык, состоящий из всех цепочек, в которых n единиц следуют за n нулями для некоторого $n \geq 0$:{ε, 01, 0011, 000111, …}.
2. Множество цепочек, состоящих из 0 и 1 и содержащих поровну тех и других: {ε, 01, 10, 0011, 1001, …}.
3. Множество двоичных записей простых чисел: {10, 11, 101, 111,1011, …}.
4. Σ^* — язык для любого алфавита Σ.
5. $\varnothing$ — пустой язык в любом алфавите.
6. {ε} — язык, содержащий одну лишь пустую цепочку. Он также является языком в любом алфавите. Заметим, что $\varnothing \neq \{\varepsilon\}$; первый не содержит вообще никаких цепочек, а второй состоит из одной цепочки.

Единственное существенное ограничение для множеств, которые могут быть языками, состоит в том, что все алфавиты конечны. Таким образом, хотя языки и могут содержать бесконечное число цепочек, но эти цепочки должны быть составлены из символов некоторого фиксированного конечного алфавита.

1.5.4. Проблемы

В теории автоматов *проблема* — это вопрос о том, является ли данная цепочка элементом определенного языка. Как мы вскоре выясним, все, что называется “проблемой” в более широком смысле слова, может быть выражено в виде проблемы принадлежности некоторому языку. Точнее, если Σ — некоторый алфавит, и L — язык в Σ, то проблема L формулируется следующим образом.

- Дана цепочка w из Σ^*, требуется выяснить, принадлежит цепочка w языку L, или нет.

Пример 1.26. Задачу проверки простоты данного числа можно выразить в терминах принадлежности языку L_p, который состоит из всех двоичных цепочек, выражающих простые числа. Таким образом, ответ “да” соответствует ситуации, когда данная цепочка из нулей и единиц является двоичным представлением простого числа. В противном случае ответом будет “нет”. Для некоторых цепочек принять решение довольно просто. Например, цепочка 0011101 не может быть представлением простого числа по той причине, что двоичное представление всякого целого числа, за исключением 0, начинается с 1. Однако решение данной проблемы для цепочки 11101 не так очевидно и требует значительных затрат таких вычислительных ресурсов, как время и/или объем памяти. □

Описание множеств как способ определения языков

Языки часто задаются с помощью конструкции, описывающей множество:
$\{w \mid$ сведения о $w\}$.
Читается данное выражение так: “множество слов w, соответствующих тому, что сказано о w справа от вертикальной черты”. Например:

1. $\{w \mid w$ содержит поровну нулей и единиц$\}$.
2. $\{w \mid w$ есть двоичное представление простого числа$\}$.
3. $\{w \mid w$ есть синтаксически правильная программа на языке C$\}$.

Кроме того, часто вместо w пишут некоторое выражение, зависящее от параметров, и описывают цепочки языка, накладывая на эти параметры определенные условия. В первом из следующих примеров в качестве параметра фигурирует n, а во втором — параметры i и j.

1. $\{0^n 1^n \mid n \geq 1\}$. Читается: “множество цепочек из n нулей и n единиц, где n больше или равно 1”. Этот язык содержит цепочки $\{01, 0011, 000111, \ldots\}$. Заметим, что, как и для алфавита, мы можем определить n-ю степень одиночного символа как цепочку из n копий данного символа.
2. $\{0^i 1^j \mid 0 \leq i \leq j\}$. Этот язык состоит из цепочек, у которых вначале идет некоторое (возможно, нулевое) число нулей, а затем некоторое число единиц, причем число последних не меньше числа нулей.

В приведенном нами определении “проблемы” есть одно слабое место. Дело в том, что обычно под проблемами подразумевают не вопросы разрешения (истинно нечто, или нет), а запросы на обработку или преобразование некоторых входных данных (желательно, наилучшим способом). Например, задача анализатора в компиляторе языка C — определить, принадлежит ли данная цепочка символов ASCII множеству L_C всех правильных программ на C — в точности соответствует нашему определению. Но в задачи анализатора входят также формирование дерева синтаксического анализа, заполнение таблицы имен и, возможно, другие действия. Хуже того, компилятор в целом решает задачу перевода C-программы в объектный код для некоторой машины, и эта задача весьма далека от простого ответа “да” или “нет” на вопрос о правильности такой программы.

Тем не менее, определение “проблем” как языков выдержало проверку временем и позволяет нам успешно решать многие задачи, возникающие в теории сложности. В рамках этой теории мы отыскиваем нижние границы сложности определенных задач. Особенно важны тут методы доказательства того, что определенные типы задач не могут быть решены за время, меньшее по количеству, чем экспонента от размера их входных данных. Оказывается, что в этом смысле задача, сформулированная в терминах теории языков (т.е. требующая ответа “да” или “нет”), так же трудна, как и исходная задача, требующая “найти решение”.

Таким образом, если мы можем доказать, что трудно определить, принадлежит ли данная цепочка множеству L_X всех правильных программ на языке X, следовательно, переводить программы с языка X в объектный код, по крайней мере, не легче. В самом деле, если было бы легко генерировать код, то мы могли бы попросту запустить транслятор и, когда он успешно выработал бы объектный код, заключить, что входная цепочка является правильной программой, принадлежащей L_X. Поскольку последний шаг определения, выработан ли объектный код, не может быть сложным, то с помощью быстрого алгоритма генерации кода мы могли бы эффективно решать задачу о принадлежности цепочки множеству L_X. Но так мы приходим к противоречию с предположением о том, что определить принадлежность цепочки языку L_X трудно. Итак, мы доказали методом от противного утверждение "если проверка принадлежности языку L_X трудна, то и компиляция программ, написанных на языке X, также трудна".

Этот метод, позволяющий показать трудность одной задачи с использованием предполагаемого эффективного алгоритма ее решения для эффективного решения другой, заведомо сложной задачи, называется "сведением" второй задачи к первой. Это мощный инструмент в исследованиях сложности проблем, причем его применение значительно упрощается в силу нашего замечания о том, что проблемами, по сути, являются лишь вопросы о принадлежности некоторому языку.

Что это — язык или проблема?

На самом деле, язык и проблема — это одно и то же. Употребление терминов зависит лишь от нашей точки зрения. Когда нас интересуют цепочки сами по себе, например, множество $\{0^n1^n \mid n \geq 1\}$, мы склонны видеть в этом множестве цепочек некоторый язык. В последних главах этой книги мы будем больше интересоваться "семантикой" цепочек, т.е. рассматривать цепочки как закодированные графы, логические выражения или целые числа. В этих случаях нас будут больше интересовать не сами цепочки, а те объекты, которые они представляют. И тогда мы будем склонны рассматривать множество цепочек как некую проблему.

Резюме

- *Конечные автоматы*. Конечные автоматы включают набор состояний и переходов между ними, зависящих от входных данных. Они весьма полезны при построении различных систем программного обеспечения, включая лексические анализаторы компиляторов и системы проверки корректности схем или протоколов.
- *Регулярные выражения*. Это структурные записи для описания некоторых шаблонов, представимых конечными автоматами. Используются во многих компонентах

программного обеспечения, например, в программах поиска по шаблону в текстах или среди файловых имен.

- *Контекстно-свободные грамматики*. Эта важная система описания структуры языков программирования и связанных с ними множеств цепочек используется при построении такого компонента компилятора, как анализатор.
- *Машины Тьюринга*. Автоматы, моделирующие все возможности реальных вычислительных машин. Позволяют изучать разрешимость, т.е. вопрос о том, что можно и чего нельзя сделать с помощью компьютера. Кроме того, они позволяют отличать задачи, разрешимые за полиномиальное время, от неразрешимых.
- *Дедуктивные доказательства*. Основной метод доказательства, состоящий из цепочки утверждений, которые либо даны как истинные, либо логически следуют из предыдущих утверждений.
- *Доказательства утверждений типа "если-то"*. Многие теоремы имеют вид: "если (нечто одно), то (нечто другое)". Утверждение (или утверждения), стоящее в скобках после "если", является гипотезой, а утверждение, стоящее после слова "то", — заключением. Цепочка дедуктивных доказательств утверждений этого типа начинается с гипотезы, из которой последовательно логически выводятся новые утверждения до тех пор, пока на каком-то шаге одно из них не совпадет с заключением.
- *Доказательства утверждений типа "тогда и только тогда"*. Существуют теоремы вида "(нечто одно) тогда и только тогда, когда (нечто другое)". Доказываются такие утверждения в одну и другую стороны как утверждения типа "если-то". Этот вид теорем очень близок к утверждениям о равенстве двух множеств, записанных разными способами. Для их доказательства нужно показать, что каждое из этих множеств содержится в другом.
- *Доказательство контрапозиции*. Доказать утверждение типа "если *H*, то *C*" иногда бывает легче путем доказательства эквивалентного утверждения "если не *C*, то не *H*", которое называют контрапозицией исходного.
- *Доказательство методом "от противного"*. В некоторых случаях удобнее вместо утверждения "если *H*, то *C*" доказывать утверждение "если *H* и не *C*, то (нечто заведомо ложное)". Этот метод доказательства называется доказательством от противного.
- *Контрпримеры*. Иногда необходимо доказывать, что некоторое утверждение не является истинным. Если это утверждение содержит один или несколько параметров, то можно доказать его ложность в целом, приведя всего один контрпример, т.е. подобрав такие значения параметров, при которых это утверждение ложно.

- *Индуктивные доказательства*. Утверждения, содержащие целый параметр n, очень часто могут быть доказаны по индукции. Для этого мы доказываем, что данное утверждение истинно для базиса, конечного числа случаев определенных значений n. Затем доказываем, что если утверждение истинно для n, то оно истинно и для $n + 1$.
- *Структурная индукция*. Довольно часто в этой книге возникает ситуация, когда в теореме, требующей индуктивного доказательства, речь идет о таких рекурсивно определяемых понятиях, как деревья. Тогда теорему о построенных таким способом объектах можно доказывать индукцией по числу шагов, использованных при их построении. Этот тип индукции называют структурной индукцией.
- *Алфавиты*. Алфавитом является некоторое конечное множество символов.
- *Цепочки*. Цепочкой называют конечную последовательность символов.
- *Языки и проблемы*. Язык есть (вообще говоря, бесконечное) множество цепочек, состоящих из символов некоторого фиксированного алфавита. Если цепочки языка должны быть проинтерпретированы некоторым способом, вопрос о принадлежности определенной цепочки этому языку иногда называют проблемой.

Литература

Для углубленного изучения материала этой главы, посвященной основополагающим математическим понятиям информатики, мы рекомендуем книгу [1].

1. A. V. Aho and J. D. Ullman, *Foundations of Computer Science*, Computer Science Press, New York, 1994.

ГЛАВА 2

Конечные автоматы

В этой главе мы введем класс языков, известных как “регулярные”. Это языки, которые могут быть описаны конечными автоматами. Последние мы уже обсудили вкратце в разделе 1.1.1. Перед тем как формально определить конечные автоматы, рассмотрим развернутый пример, из которого станет ясной мотивация последующего изучения этих объектов.

Как указывалось ранее, конечный автомат состоит из множества состояний и “управления”, переводящего из одного состояния в другое в зависимости от получаемых извне “входных данных”. Классы автоматов существенно различаются по типу этого управления. Управление может быть “детерминированным” в том смысле, что автомат может находиться в каждый момент времени не более чем в одном состоянии, и “недетерминированным”, т.е. автомат может одновременно находиться в нескольких состояниях. Мы выясним, что добавление недетерминизма не позволяет определять языки, которые нельзя было бы определить с помощью детерминированных конечных автоматов. Тем не менее, недетерминированные автоматы оказываются весьма эффективными в приложениях. Именно недетерминизм позволяет нам “программировать” решение задач, используя языки высокого уровня. В этой главе рассматривается алгоритм “компиляции” недетерминированного конечного автомата в детерминированный, который затем может быть “выполнен” на обычной вычислительной машине.

В заключительной части главы изучается расширенный недетерминированный автомат, который имеет дополнительную возможность переходить из состояния в состояние спонтанно, т.е. по пустой цепочке в качестве входа. Эти автоматы также описывают только регулярные языки. Тем не менее, они окажутся совершенно необходимыми в главе 3 при изучении регулярных выражений и доказательстве их эквивалентности автоматам.

Изучение регулярных языков продолжается в главе 3. Там представлен еще один важный способ их описания посредством такой алгебраической нотации, как регулярное выражение. Мы изучим регулярные выражения и покажем их эквивалентность конечным автоматам, чтобы в главе 4 использовать и те, и другие как инструменты для доказательства некоторых важных свойств регулярных языков. Например, свойства “замкнутости”, позволяющие утверждать, что некоторый язык является регулярным, на том основании, что один или несколько других языков регулярны. Еще один пример — “разрешимые” свойства, т.е. наличие алгоритмов, позволяющих ответить на вопросы, касающиеся автомата или регулярного выражения, например, представляют ли два различных автомата или регулярных выражения один и тот же язык.

2.1. Неформальное знакомство с конечными автоматами

В этом разделе рассматривается развернутый пример реальной проблемы, в решении которой важную роль играют конечные автоматы. Мы изучим протоколы, поддерживающие операции с "электронными деньгами" — файлами, которые клиент использует для платы за товары в Internet, а продавец получает с гарантией, что "деньги" — настоящие. Для этого продавец должен знать, что эти файлы не были подделаны или скопированы и отосланы продавцу, хотя клиент сохраняет копию этого файла и вновь использует ее для оплаты.

Невозможность подделки файла должна быть гарантирована банком и стратегией шифрования. Таким образом, третий участник, банк, должен выпускать и шифровать "денежные" файлы так, чтобы исключить возможность подделки. Но у банка есть и другая важная задача: хранить в своей базе данных информацию о всех выданных им деньгах, годных к платежу. Это нужно для того, чтобы банк мог подтвердить, что полученный магазином файл представляет реальные деньги и может быть переведен на счет магазина. Мы не будем останавливаться на криптографическом аспекте проблемы, а также на том, каким образом банк может хранить и обрабатывать биллионы "электронных денежных счетов". Весьма маловероятно, чтобы эти проблемы привели к каким-нибудь долговременным затруднениям в концепции электронных денег, тем более что они используются в относительно небольших масштабах с конца 1990-х годов.

Однако для того, чтобы использовать электронные деньги, необходимо составить протоколы, позволяющие производить с этими деньгами различные действия в зависимости от желания пользователя. Поскольку в монетарных системах всегда возможно мошенничество, нужно проверять правильность использования денег, какая бы система шифрования ни применялась. Иными словами, нужно гарантировать, что произойти могут только предусмотренные события. Это не позволит нечистому на руку пользователю украсть деньги у других или их "напечатать". В конце раздела приводится очень простой пример (плохого) протокола расчета электронными деньгами, моделируемого конечными автоматами, и показывается, как конструкции на основе автоматов можно использовать для проверки протоколов (или, как в нашем случае, для поиска в протоколе изъянов).

2.1.1. Основные правила

Есть три участника: клиент, магазин и банк. Для простоты предположим, что есть всего один "денежный" файл ("деньги"). Клиент может принять решение передать этот файл магазину, который затем обменяет его в банке (точнее, потребует, чтобы банк взамен его выпустил новый файл, принадлежащий уже не клиенту, а магазину) и доставит товар клиенту. Кроме того, клиент имеет возможность отменить свой файл, т.е. попросить банк вернуть деньги на свой счет, причем они уже не могут быть израсхо-

дованы. Взаимодействие трех участников ограничено, таким образом, следующими пятью событиями.

1. Клиент может совершить *оплату* (*pay*) товара, т.е. переслать денежный файл в магазин.
2. Клиент может выполнить *отмену* (*cancel*) денег. Они отправляются в банк вместе с сообщением о том, что их сумму следует добавить к банковскому счету клиента.
3. Магазин может произвести *доставку* (*ship*) товара клиенту.
4. Магазин может совершить *выкуп* (*redeem*) денег. Они отправляются в банк вместе с требованием передать их сумму магазину.
5. Банк может выполнить *перевод* (*transfer*) денег, создав новый, надлежащим образом зашифрованный, файл и переслав его магазину.

2.1.2. Протокол

Во избежание недоразумений участники должны вести себя осторожно. В нашем случае мы резонно полагаем, что клиенту доверять нельзя. Клиент, в частности, может попытаться скопировать денежный файл и после этого уплатить им несколько раз или уплатить и отменить его одновременно, получая, таким образом, товар бесплатно.

Банк должен вести себя ответственно, иначе он не банк. В частности, он должен проверять, не посылают ли на выкуп два разных магазина один и тот же денежный файл, и не допускать, чтобы одни и те же деньги и отменялись, и выкупались. Магазин тоже должен быть осторожен. Он, например, не должен доставлять товар, пока не убедится, что получил за него деньги, действительные к оплате.

Протоколы такого типа можно представить в виде конечных автоматов. Каждое состояние представляет ситуацию, в которой может находиться один из участников. Таким образом, состояние “помнит”, что одни важные события произошли, а другие — еще нет. Переходы между состояниями в нашем случае совершаются, когда происходит одно из пяти описанных выше событий. События мы будем считать “внешними” по отношению к автоматам, представляющим трех наших участников, несмотря на то, что каждый из них может инициировать одно или несколько из этих событий. Оказывается, важно не то, кому именно позволено вызывать эти события, а то, какие последовательности событий могут произойти.

На рис. 2.1 участники представлены автоматами. На диаграмме показаны лишь те события, которые влияют на того или иного участника. Например, действие *оплата* влияет лишь на клиента и магазин. Банк не знает о том, что клиент отправил деньги в магазин; он узнает об этом, когда магазин выполняет действие *выкуп*.

Рассмотрим вначале автомат (*в*), изображающий банк. Его начальное состояние — это состояние 1. Оно соответствует ситуации, когда банк выпустил денежный файл, о котором идет речь, но еще не получил требования на его *выкуп* или *отмену*. Если клиент

посылает в банк запрос на *отмену*, то банк восстанавливает деньги на счету клиента и переходит в состояние 2. Оно представляет ситуацию, в которой деньги возвращены клиенту. Поскольку банк в ответе за свои действия, то, попав в состояние 2, он уже не покидает его. Этим он не позволит клиенту вернуть на свой счет с помощью *отмены* или израсходовать те же самые деньги.[1]

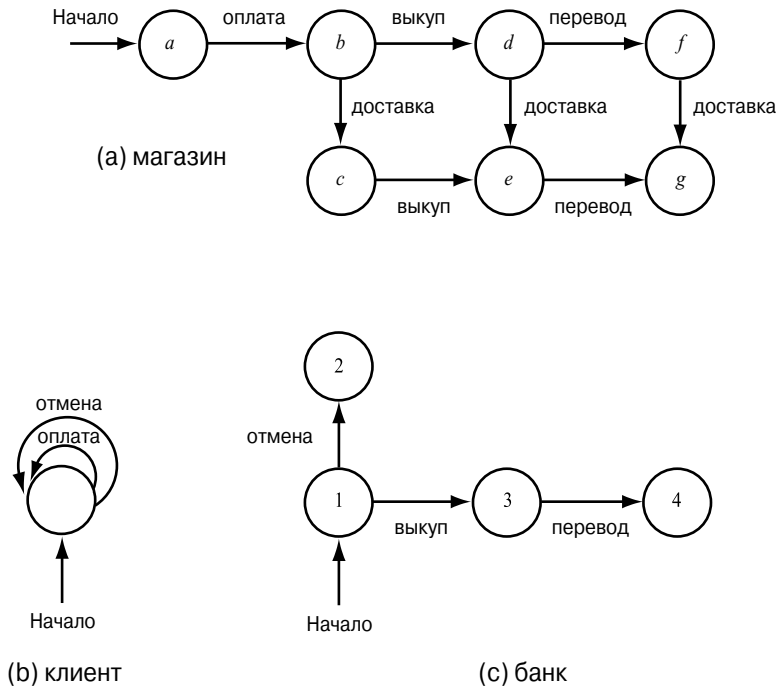


Рис. 2.1. Конечные автоматы, представляющие клиента, магазин и банк

С другой стороны, находясь в состоянии 1, банк может получить от магазина требование *выкупа* денег. В этом случае он переходит в состояние 3 и тут же отправляет магазину сообщение о *переводе*, в котором содержится новый денежный файл, принадлежащий уже магазину. Отослав это сообщение, банк переходит в состояние 4. В этом состоянии он не принимает запросов на *отмену* или *выкуп* денег, равно как не совершает никаких других действий с этим конкретным денежным файлом.

Рассмотрим теперь автомат на рис. 2.1, *a*, представляющий действия магазина. В то время как банк всегда работает безукоризненно, в системе работы магазина есть изъяны. Представьте, что доставка товара и финансовые операции совершаются отдельно друг от

[1] Нужно помнить, что тут речь идет об одном денежном файле. На самом деле, банк будет работать по такому же протоколу с большим количеством единиц электронных денег. При этом всякий раз протокол работы с каждой такой единицей — один и тот же, поэтому мы можем рассматривать данную проблему так, как если бы существовала всего одна единица электронных денег.

друга. Тогда доставка может быть выполнена до, во время или после выкупа электронных денег. Придерживаясь такой политики, магазин рискует попасть в ситуацию, когда товар уже доставлен, а деньги, как выясняется, поддельные.

Магазин начинает в состоянии *a*. Когда покупатель заказывает товар, выполняя *оплату*, магазин переходит в состояние *b*. В этом состоянии магазин начинает одновременно два процесса: *доставку* товара и *выкуп* денег. Если первым заканчивается процесс доставки, то магазин переходит в состояние *c*, в котором он еще должен осуществить *выкуп* денег и получить *перевод* эквивалентного денежного файла из банка. Магазин также может сначала отправить в банк запрос на *выкуп* денег и перейти в состояние *d*. В состоянии *d* магазин либо доставит товар и перейдет в состояние *e*, либо получит из банка денежный перевод и перейдет в состояние *f*. Следует ожидать, что в состоянии *f* магазин в конце концов доставит товар и перейдет в состояние *g*. В последнем случае сделка завершена, и ничего больше не происходит. В состоянии *e* магазин ожидает *перевода* денег из банка. К сожалению, может получиться, что магазину не повезет: товар он доставит, а денежного перевода так никогда и не получит.

Рассмотрим, наконец, автомат на 2.1, *б*, изображающий клиента. У этого автомата есть только одно состояние, отражающее тот факт, что клиент может делать все, что угодно. Он может выполнять *оплату* или *отмену* сколько угодно раз и в любом порядке. При этом после каждого действия он остается в своем единственном состоянии.

2.1.3. Возможность игнорирования автоматом некоторых действий

Тройка автоматов на рис. 2.1 отображает поведение участников независимо друг от друга, однако некоторые переходы в автоматах пропущены. Так, сообщение об *отмене* денег не затрагивает магазин, и если клиент отменяет деньги, то магазин должен оставаться в том же состоянии, в котором находился. Однако согласно формальному определению конечного автомата, которое мы рассмотрим в разделе 2.2, если на вход автомата подается *X*, то он должен совершить переход по дуге с меткой *X* из текущего состояния в некоторое новое. Следовательно, к каждому состоянию автомата для магазина нужно добавить еще одну дугу с меткой *отмена*, ведущую в то же состояние. Тогда при выполнении *отмены* автомат, изображающий магазин, может совершить “переход”, который состоит в том, что автомат остается в том же состоянии, в котором и был. Если бы этих дополнительных дуг не было, то автомат, изображающий магазин, “умирал” бы, т.е. он не находился бы ни в каком состоянии, и любые его последующие действия были бы невозможны.

Еще одна потенциальная проблема кроется в том, что участники могут, умышленно или случайно, отправить сообщение, не предусмотренное протоколом, и мы не хотим, чтобы это повлекло “смерть” одного из автоматов. Представим, например, что клиент выполнил действие *оплата* во второй раз, когда магазин находился в состоянии *e*. Поскольку это состояние не имеет выходящей дуги с меткой *оплата*, то автомат, изображающий магазин, умрет прежде, чем получит *перевод* из банка. Итак, к некоторым со-

стояниям автоматов на рис. 2.1 нужно добавить петли с метками, обозначающими действия, которые следует проигнорировать в этих состояниях. Дополненные таким образом автоматы изображены на рис. 2.2. Для экономии места дуги с разными метками, имеющие начало и конец в одном и том же состоянии, объединяются в одну дугу с несколькими метками. Игнорироваться должны следующие два типа действий.

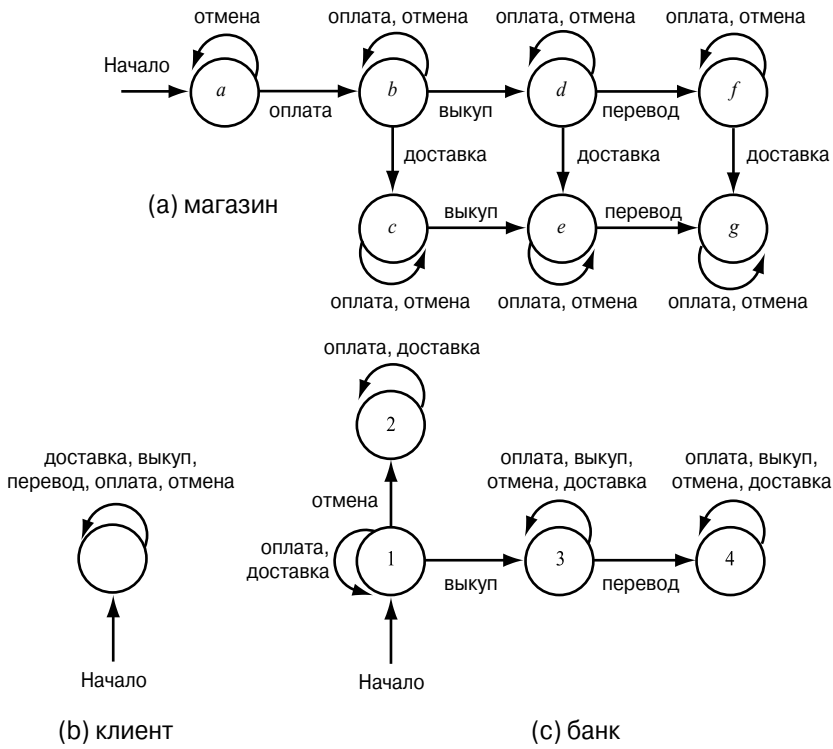


Рис. 2.2. Полные множества переходов для трех автоматов

1. *Действия, не затрагивающие данного участника*. Как мы видели, для магазина единственным таким действием является *отмена*, поэтому каждое его состояние имеет петлю с меткой *отмена*. К банку не имеют отношения ни *оплата*, ни *доставка*, а потому к каждому из его состояний добавляется петля с метками *оплата* и *доставка*. Клиента не затрагивают *доставка*, *выкуп* и *перевод*, и мы добавляем дуги с такими метками. В итоге, какая бы последовательность действий ни была подана на вход, он остается в своем единственном состоянии. Поэтому на операции, совершаемые системой в целом, автомат клиента не влияет. Безусловно, клиент остается участником, так как именно он инициирует действия *оплата* и *отмена*. Но, как мы уже говорили, для поведения автоматов неважно, кто именно инициирует те или иные действия.
2. *Действия, которые не следует допускать во избежание смерти автомата*. Как упоминалось ранее, чтобы не убить автомат магазина, нельзя позволять покупателю повторно выполнять *оплату*. С этой целью добавлены петли с меткой *оплата* ко

всем его состояниям, за исключением состояния *a* (в котором действие *оплата* уместно и ожидаемо). Кроме того, добавлены петли с меткой *отмена* к состояниям 3 и 4 банка для того, чтобы не допустить смерти автомата банка, если покупатель попытается отменить деньги, которые уже были выкуплены. Банк с полным правом игнорирует такое требование. Точно так же состояния 3 и 4 имеют петли с меткой *выкуп*. Магазин не должен пытаться дважды *выкупить* одни и те же деньги. Но если он все же делает это, то банк совершенно справедливо игнорирует второе требование.

2.1.4. Система в целом как автомат

Обычный способ изучения взаимодействия подобных автоматов состоит в построении так называемого *автомата-произведения*. Состояниями этого автомата являются пары состояний, первое из которых есть состояние магазина, а второе — состояние банка. Например, состояние автомата-произведения (3, *d*) представляет ситуацию, когда банк находится в состоянии 3, а магазин — в состоянии *d*. Поскольку магазин имеет четыре состояния, а банк — семь, то число состояний автомата-произведения равно 4×7 = 28.

Данный автомат-произведение изображен на рис. 2.3. Для ясности все 28 состояний расположены в виде массива, строки которого соответствуют состояниям банка, а столбцы — состояниям магазина. Кроме того, в целях экономии места дуги помечены буквами, каждая из которых соответствует определенному действию: *P* — *оплата* (*pay*), *S* — *доставка* (*ship*), *C* — *отмена* (*cancel*), *R* — *выкуп* (*redeem*), *T* — *перевод* (*transfer*).

Чтобы правильно построить дуги в автомате-произведении, нужно проследить “параллельную” работу автоматов банка и магазина. Каждый из двух компонентов автомата-произведения совершает, в зависимости от входных действий, различные переходы. Важно отметить, что если, получив на вход некоторое действие, один из этих двух автоматов не может совершить переход ни в какое состояние, то автомат-произведение “умирает”, поскольку также не может перейти ни в какое состояние.

Придадим строгость правилу переходов из одного состояния в другое. Рассмотрим автомат-произведение в состоянии (*i*, *x*). Это состояние соответствует ситуации, когда банк находится в состоянии *i*, а магазин — в состоянии *x*. Пусть *Z* означает одно из входных действий. Мы смотрим, имеет ли автомат банка переход из состояния *i* с меткой *Z*. Предположим, что такой переход есть, и ведет он в состояние *j* (которое может совпадать с *i*, если банк, получив на вход *Z*, остается в том же состоянии). Затем, глядя на автомат магазина, мы выясняем, есть ли у него дуга с меткой *Z*, ведущая в некоторое состояние *y*. Если *j* и *y* существуют, то автомат-произведение содержит дугу из состояния (*i*, *x*) в состояние (*j*, *y*) с меткой *Z*. Если же либо состояния *j*, либо состояния *y* нет (по той причине, что банк или магазин для входного действия *Z* не имеет, соответственно, перехода из состояния *i* или *x*), то не существует и дуги с меткой *Z*, выходящей из состояния (*i*, *x*).

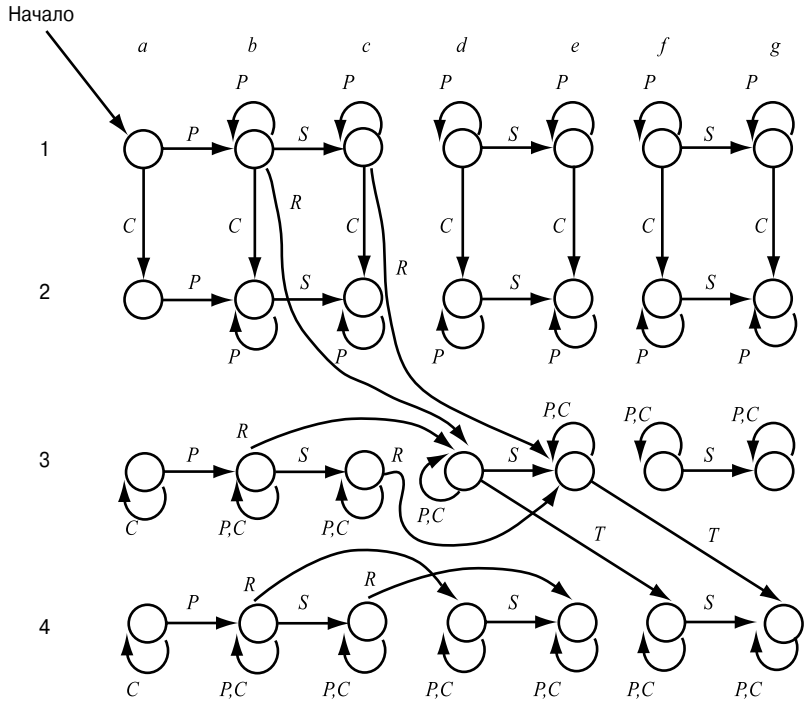


Рис. 2.3. Автомат-произведение для магазина и банка

Теперь ясно, каким образом выбраны дуги на рис. 2.3. Например, получая на вход действие *оплата*, магазин совершает переход из состояния *a* в состояние *b*, а из любого другого состояния — в него же. Банк же, получая на вход действие *оплата*, в любом случае сохраняет свое состояние, поскольку это действие его не затрагивает. Эти наблюдения объясняют, как были построены четыре дуги с меткой *P* в четырех столбцах рис. 2.3 и петли с меткой *P* для других состояний.

Еще один пример выбора дуг мы получим, рассмотрев действие *выкуп*. Если банк получит запрос на *выкуп*, находясь в состоянии 1, то он перейдет в состояние 3, а если в состоянии 3 или 4 — останется в том же состоянии. Если же он получит это действие на вход, находясь в состоянии 2, то "умрет", т.е. не сможет никуда перейти. С другой стороны, магазин, получив на вход *выкуп*, может из состояния *b* перейти в *d* или из *c* в *e*. На рис. 2.3 мы видим шесть дуг с меткой *выкуп*, соответствующих шести комбинациям из трех состояний банка и двух состояний магазина, имеющих выходящие дуги с меткой *R*. Например, из состояния (1, *b*) дуга с меткой *R* переводит автомат в состояние (3, *d*), так как *выкуп* переводит банк из состояния 1 в 3, а магазин — из *b* в *d*. Еще один пример: существует дуга с меткой *R*, ведущая из состояния (4, *c*) в (4, *e*), поскольку *выкуп* переводит банк из состояния 4 снова в состояние 4, а магазин — из *c* в *e*.

2.1.5. Проверка протокола с помощью автомата-произведения

Из рис. 2.3 можно узнать кое-что интересное. Так, из начального состояния (1, *a*) — комбинации начальных состояний банка и магазина — можно попасть только в десять из всех 28 состояний. Заметим, что такие состояния, как (2, *e*) и (4, *d*), не являются *достижимыми*, т.е. пути, ведущего к ним из начального состояния, не существует. Нет надобности включать в автомат недостижимые состояния, и в нашем случае это сделано исключительно для последовательности изложения.

Однако реальной целью анализа протоколов, подобных данному, с помощью автоматов является ответ на вопрос: "возможна ли ошибка данного типа?". Простейший пример: нас может интересовать, возможно ли, что магазин доставит товар, а оплаты за него так и не получит, т.е. может ли автомат-произведение попасть в состояние, в котором магазин уже завершил доставку (и находится в одном из состояний в столбцах *c*, *e* или *g*), и при этом перехода, соответствующего входу *T*, никогда ранее не было и не будет.

К примеру, в состоянии (3, *e*) товар уже доставлен, но переход в состояние (4, *g*), соответствующий входу *T*, в конце концов произойдет. В терминах действий банка это означает, что если банк попал в состояние 3, то он уже получил запрос на *выкуп* и обработал его. Значит, он находился в состоянии 1 перед получением этого запроса, не получал требования об *отмене* и будет игнорировать его в будущем. Таким образом, в конце концов банк переведет деньги магазину.

Однако в случае состояния (2, *e*) мы сталкиваемся с проблемой. Состояние достижимо, но единственная выходящая дуга ведет в него же. Это состояние соответствует ситуации, когда банк получил сообщение об *отмене* раньше, чем запрос на *выкуп*. Но магазин получил сообщение об *оплате*, т.е. наш пройдоха-клиент одни и те же деньги и потратил, и отменил. Магазин же, по глупости, доставил товар прежде, чем попытался выкупить деньги. Теперь, если магазин выполнит запрос на *выкуп*, то банк даже не подтвердит получение соответствующего сообщения, так как после *отмены*, находясь в состоянии 2, банк не будет обрабатывать запрос на *выкуп*.

2.2. Детерминированные конечные автоматы

Теперь пора ввести формальное понятие конечного автомата и уточнить рассуждения и описания из разделов 1.1.1 и 2.1. Начнем с рассмотрения формализма детерминированного конечного автомата, который, прочитав любую последовательность входных данных, может находиться только в одном состоянии. Термин "детерминированный" говорит о том, что для всякой последовательности входных символов существует лишь одно состояние, в которое автомат может перейти из текущего. В противоположность детерминированному, "недетерминированный" конечный автомат, который рассматривается в разделе 2.3, может находиться сразу в нескольких состояниях. Под термином "конечный автомат" далее подразумевается автомат детерминированного типа. Но

обычно для того, чтобы напомнить читателю, автомат какого типа рассматривается, употребляется слово “детерминированный” или сокращение ДКА (DFA — Deterministic Finite Automaton).

2.2.1. Определение детерминированного конечного автомата

Детерминированный конечный автомат состоит из следующих компонентов.

1. Конечное множество *состояний*, обозначаемое обычно как Q.
2. Конечное множество *входных символов*, обозначаемое обычно как Σ.
3. *Функция переходов*, аргументами которой являются текущее состояние и входной символ, а значением — новое состояние. Функция переходов обычно обозначается как δ. Представляя нестрого автомат в виде графа, мы изображали δ отмеченными дугами, соединяющими состояния. Если q — состояние и a — входной символ, то $\delta(q, a)$ — это состояние p, для которого существует дуга, отмеченная символом a и ведущая из q в p.[2]
4. *Начальное состояние*, одно из состояний в Q.
5. Множество *заключительных*, или *допускающих*, состояний F. Множество F является подмножеством Q.

В дальнейшем детерминированный конечный автомат часто обозначается как *ДКА*. Наиболее компактное представление ДКА — это список пяти вышеуказанных его компонентов. В доказательствах ДКА часто трактуется как пятерка

$$A = (Q, \Sigma, \delta, q_0, F),$$

где A — имя ДКА, Q — множество состояний, Σ — множество входных символов, δ — функция переходов, q_0 — начальное состояние и F — множество допускающих состояний.

2.2.2. Как ДКА обрабатывает цепочки

Первое, что следует выяснить о ДКА, — это понять, каким образом ДКА решает, “допускать” ли последовательность входных символов. “Язык” ДКА — это множество всех его допустимых цепочек. Пусть $a_1a_2\ldots a_n$ — последовательность входных символов. ДКА начинает работу в начальном состоянии q_0. Для того чтобы найти состояние, в которое A перейдет после обработки первого символа a_1, мы обращаемся к функции переходов δ. Пусть, например, $\delta(q_0, a_1) = q_1$. Для следующего входного символа a_2 находим $\delta(q_1, a_2)$. Пусть это будет состояние q_2. Аналогично находятся и последующие состояния $q_3, q_4, \ldots, q_n$, где $\delta(q_{i-1}, a_i) = q_i$ для каждого i. Если q_n принадлежит множеству F, то вход-

[2] Точнее говоря, граф есть изображение некоторой функции переходов δ, а дуги этого графа отображают переходы, определяемые функцией δ.

ная последовательность $a_1a_2\ldots a_n$ допускается, в противном случае она “отвергается” как недопустимая.

Пример 2.1. Определим формально ДКА, допускающий цепочки из 0 и 1, которые содержат в себе подцепочку 01. Этот язык можно описать следующим образом:

$\{w \mid w$ имеет вид $x01y$, где x и y — цепочки, состоящие только из 0 и 1$\}$.

Можно дать и другое, эквивалентное описание, содержащее x и y слева от вертикальной черты:

$\{x01y \mid x$ *и* y *— некоторые цепочки, состоящие из* 0 *и* 1$\}$.

Примерами цепочек этого языка являются цепочки 01, 11010 и 1000111. В качестве примеров цепочек, *не принадлежащих* данному языку, можно взять цепочки ε, 0 и 111000.

Что мы знаем об автомате, допускающем цепочки данного языка L? Во-первых, что алфавитом его входных символов является $\Sigma = \{0, 1\}$. Во-вторых, имеется некоторое множество Q состояний этого автомата. Один из элементов этого множества, скажем, q_0, является его начальным состоянием. Для того чтобы решить, содержит ли входная последовательность подцепочку 01, автомат A должен помнить следующие важные факты относительно прочитанных им входных данных.

1. Была ли прочитана последовательность 01? Если это так, то всякая читаемая далее последовательность допустима, т.е. с этого момента автомат будет находиться лишь в допускающих состояниях.
2. Если последовательность 01 еще не считана, то был ли на предыдущем шаге считан символ 0? Если это так, и на данном шаге читается символ 1, то последовательность 01 будет прочитана, и с этого момента автомат будет находиться только в допускающих состояниях.
3. Действительно ли последовательность 01 еще не прочитана, и на предыдущем шаге на вход либо ничего не подавалось (состояние начальное), либо был считан символ 1? В этом случае A не перейдет в допускающее состояние до тех пор, пока им не будут считаны символы 0 и сразу за ним 1.

Каждое из этих условий можно представить как некоторое состояние. Условию (3) соответствует начальное состояние q_0. Конечно, находясь в самом начале процесса, нужно последовательно прочитать 0 и 1. Но если в состоянии q_0 читается 1, то это нисколько не приближает к ситуации, когда прочитана последовательность 01, поэтому нужно оставаться в состоянии q_0. Таким образом, $\delta(q_0, 1) = q_0$.

Однако если в состоянии q_0 читается 0, то мы попадаем в условие (2), т.е. 01 еще не прочитаны, но уже прочитан 0. Пусть q_2 обозначает ситуацию, описываемую условием (2). Переход из q_0 по символу 0 имеет вид $\delta(q_0, 0) = q_2$.

Рассмотрим теперь переходы из состояния q_2. При чтении 0 мы попадаем в ситуацию, которая не лучше предыдущей, но и не хуже. 01 еще не прочитаны, но уже прочитан 0, и

теперь ожидается 1. Эта ситуация описывается состоянием q_2, поэтому определим $\delta(q_2, 0) = q_2$. Если же в состоянии q_2 читается 1, то становится ясно, что во входной последовательности непосредственно за 0 следует 1. Таким образом, можно перейти в допускающее состояние, которое обозначается q_1 и соответствует приведенному выше условию (1), т.е. $\delta(q_2, 1) = q_1$.

Наконец, нужно построить переходы в состоянии q_1. В этом состоянии уже прочитана последовательность 01, и, независимо от дальнейших событий, мы будем находиться в этом же состоянии, т.е. $\delta(q_1, 0) = \delta(q_1, 1) = q_1$.

Таким образом, $Q = \{q_0, q_1, q_2\}$. Ранее упоминалось, что q_0 — начальное, а q_1 — единственное допускающее состояние автомата, т.е. $F = \{q_1\}$. Итак, полное описание автомата A, допускающего язык L цепочек, содержащих 01 в качестве подцепочки, имеет вид

$$A = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

где δ — функция, описанная выше. □

2.2.3. Более простые представления ДКА

Определение ДКА как пятерки объектов с детальным описанием функции переходов слишком сухое и неудобочитаемое. Существует два более удобных способа описания автоматов.

1. *Диаграмма переходов*, которая представляет собой граф (его пример приведен в разделе 2.1).
2. *Таблица переходов*, дающая табличное представление функции δ. Из нее очевидны состояния и входной алфавит.

Диаграммы переходов

Диаграмма переходов для ДКА вида $A = (Q, \Sigma, \delta, q_0, F)$ есть граф, определяемый следующим образом:

а) всякому состоянию из Q соответствует некоторая вершина;

б) пусть $\delta(q, a) = p$ для некоторого состояния q из Q и входного символа a из Σ. Тогда диаграмма переходов должна содержать дугу из вершины q в вершину p, отмеченную a. Если существует несколько входных символов, переводящих автомат из состояния q в состояние p, то диаграмма переходов может содержать одну дугу, отмеченную списком этих символов;

в) диаграмма содержит стрелку в начальное состояние, отмеченную как *Начало*. Эта стрелка не выходит ни из какого состояния;

г) вершины, соответствующие допускающим состояниям (состояниям из F), отмечаются двойным кружком. Состояния, не принадлежащие F, изображаются простым (одинарным) кружком.

Пример 2.2. На рис. 2.4 изображена диаграмма переходов для ДКА, построенного в примере 2.1. На диаграмме видны три вершины, соответствующие трем состояниям, стрелка *Начало*, ведущая в начальное состояние q_0, и одно допускающее состояние q_1, отмеченное двойным кружком. Из каждого состояния выходят две дуги: одна отмечена 0, вторая — 1 (для состояния q_1 дуги объединены в одну). Каждая дуга соответствует одному из фактов для δ, построенных в примере 2.1. □

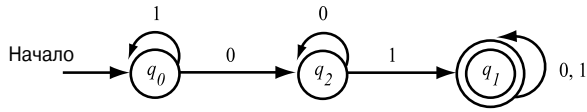


Рис. 2.4. Диаграмма переходов для ДКА, допускающего все цепочки, которые содержат подцепочку 01

Таблицы переходов

Таблица переходов представляет собой обычное табличное представление функции, подобной δ, которая двум аргументам ставит в соответствие одно значение. Строки таблицы соответствуют состояниям, а столбцы — входным символам. На пересечении строки, соответствующей состоянию q, и столбца, соответствующего входному символу a, находится состояние $\delta(q, a)$.

Пример 2.3. На рис. 2.5 представлена таблица переходов, соответствующая функции δ из примера 2.1. Кроме того, здесь показаны и другие особенности таблицы переходов. Начальное состояние отмечено стрелкой, а допускающее — звездочкой. Поскольку прописные символы строк и столбцов задают множества состояний и символов, у нас есть вся информация, необходимая для однозначного описания данного конечного автомата. □

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

Рис. 2.5. Таблица переходов для ДКА из примера 2.1

2.2.4. Расширение функции переходов на цепочки

Ранее было нестрого обосновано утверждение о том, что всякий ДКА определяет некоторый язык, а именно: множество всех цепочек, приводящих автомат из начального состояния в одно из допускающих. В терминах диаграмм переходов язык ДКА — это множество меток вдоль всех путей, ведущих из начального состояния в любое допускающее.

Теперь дадим строгое определение языка ДКА. С этой целью определим *расширенную функцию переходов*, которая описывает ситуацию, при которой мы, начиная с произвольного состояния, отслеживаем произвольную последовательность входных символов. Если δ — наша функция переходов, то расширенную функцию, построенную по δ, обозначим $\hat{\delta}$. Расширенная функция переходов ставит в соответствие состоянию q и цепочке w состояние p, в которое автомат попадает из состояния q, обработав входную последовательность w. Определим $\hat{\delta}$ индукцией по длине входной цепочки следующим образом.

Базис. $\hat{\delta}(q, \varepsilon) = q$, т.е., находясь в состоянии q и не читая вход, мы остаемся в состоянии q.

Индукция. Пусть w — цепочка вида xa, т.е. a — последний символ в цепочке, а x — цепочка, состоящая из всех символов цепочки w, за исключением последнего.[3] Например, $w = 1101$ разбивается на $x = 110$ и $a = 1$. Тогда

$$\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a) \qquad (2.1)$$

Выражение (2.1) может показаться довольно громоздким, но его идея проста. Для того чтобы найти $\hat{\delta}(q, w)$, мы вначале находим $\hat{\delta}(q, x)$ — состояние, в которое автомат попадает, обработав все символы цепочки w, кроме последнего. Предположим, что это состояние p, т.е. $\hat{\delta}(q, x) = p$. Тогда $\hat{\delta}(q, w)$ — это состояние, в которое автомат переходит из p при чтении a — последнего символа w. Таким образом, $\hat{\delta}(q, w) = \delta(p, a)$.

Пример 2.4. Построим ДКА, допустимым для которого является язык

$L = \{w \mid w$ содержит четное число 0 и четное число 1$\}$.

Вполне естественно, что состояния данного ДКА используются для подсчета числа нулей и единиц. При этом подсчет ведется по модулю 2, т.е. состояния “запоминают”, четное или нечетное число 0 или 1 было прочитано на данный момент. Итак, существуют четыре состояния, которые можно описать следующим образом.

q_0 : Прочитано четное число 0 и четное число 1.

q_1 : Прочитано четное число 0 и нечетное число 1.

q_2 : Прочитано четное число 1 и нечетное число 0.

q_3 : Прочитано нечетное число 0 и нечетное число 1.

Состояние q_0 одновременно является и начальным, и единственным допускающим. Начальным оно является потому, что до того, как будут прочитаны какие-либо входные данные, количество прочитанных и 0, и 1 равно нулю, а нуль — число четное. Это же со-

[3] Напомним, что мы условились обозначать символы буквами из начальной части алфавита, а цепочки — буквами из конца алфавита. Это соглашение необходимо для того, чтобы понять смысл выражения “цепочка вида xa”.

стояние — единственное допускающее, поскольку в точности описывает условие принадлежности последовательности из 0 и 1 языку L.

Теперь мы знаем почти все, что нужно для определения ДКА, соответствующего языку L. Это автомат

$$A = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0\}),$$

где функция переходов δ изображается диаграммой переходов на рис. 2.6. Заметим, что по каждому символу 0 совершается переход через горизонтальную пунктирную линию. Таким образом, после чтения четного числа символов 0 мы находимся над горизонтальной линией, в состоянии q_0 или q_1, а после нечетного числа — под ней, в состоянии q_2 или q_3. Аналогично, символ 1 заставляет нас пересечь вертикальную пунктирную линию. Следовательно, после чтения четного числа единиц мы находимся слева от вертикальной линии, в состоянии q_0 или q_2, а после чтения нечетного — справа, в состоянии q_1 или q_3. Эти наблюдения представляют собой нестрогое доказательство того, что данные четыре состояния интерпретируются правильно. Хотя можно и формально, как в примере 1.23, доказать корректность наших утверждений о состояниях, используя совместную индукцию.

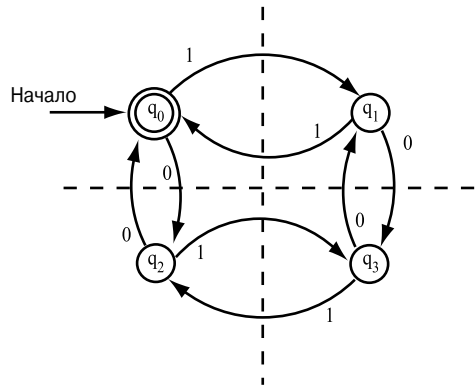


Рис. 2.6. Диаграмма переходов для ДКА из примера 2.4

Данный ДКА можно также представить таблицей переходов, изображенной на рис. 2.7. Но нам нужно не просто построить ДКА. Мы хотим с его помощью показать, как строится функция $\hat{\delta}$ по функции переходов δ. Допустим, на вход подается цепочка 110101. Она содержит четное число 0 и 1, поэтому принадлежит данному языку. Таким образом, мы ожидаем, что $\hat{\delta}(q_0, 110101) = q_0$, так как q_0 — единственное допускающее состояние. Проверим это утверждение.

Для проверки требуется найти $\hat{\delta}(q_0, w)$ для всех постепенно нарастающих, начиная с ε, префиксов w цепочки 110101. Результат этих вычислений выглядит следующим образом.

	0	1
$*\rightarrow q_0$	q_2	q_1
$*q_1$	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Рис. 2.7. Таблица переходов *для ДКА из примера 2.4*

- $\hat{\delta}(q_0, \varepsilon) = q_0$.
- $\hat{\delta}(q_0, 1) = \delta(\hat{\delta}(q_0, \varepsilon), 1) = \delta(q_0, 1) = q_1$.
- $\hat{\delta}(q_0, 11) = \delta(\hat{\delta}(q_0, 1), 1) = \delta(q_1, 1) = q_0$.
- $\hat{\delta}(q_0, 110) = \delta(\hat{\delta}(q_0, 11), 0) = \delta(q_0, 0) = q_2$.
- $\hat{\delta}(q_0, 1101) = \delta(\hat{\delta}(q_0, 110), 1) = \delta(q_2, 1) = q_3$.
- $\hat{\delta}(q_0, 11010) = \delta(\hat{\delta}(q_0, 1101), 0) = \delta(q_3, 0) = q_1$.
- $\hat{\delta}(q_0, 110101) = \delta(\hat{\delta}(q_0, 11010), 1) = \delta(q_1, 1) = q_0$.

□

Стандартные обозначения и локальные переменные

По прочтении этого раздела может сложиться впечатление, что нужно обязательно пользоваться введенными здесь обозначениями, т.е. функцию переходов обозначать δ, ДКА — буквой A и т.д. Действительно, во всех примерах мы стараемся использовать одинаковые переменные для обозначения однотипных объектов. Делается это для того, чтобы легче было вспомнить, о каком типе переменных идет речь. Так, в программах i почти всегда обозначает переменную целого типа. Однако в выборе обозначений для компонентов автомата (или чего-либо другого) мы совершенно свободны. Например, при желании мы можем обозначить ДКА буквой M, а его функцию переходов — буквой T.

Более того, нет ничего странного в том, что одна и та же переменная обозначает, в зависимости от контекста, разные объекты. Например, функции переходов в примерах 2.1 и 2.4 обозначены буквой δ. Но эти две функции являются локальными переменными и относятся только к своим примерам. Они значительно отличаются друг от друга и никак не связаны.

2.2.5. Язык ДКА

Теперь можно определить *язык* ДКА вида $A = (Q, \Sigma, \delta, q_0, F)$. Этот язык обозначается $L(A)$ и определяется как

$L(A) = \{ w \mid \hat{\delta}(q_0, w)$ принадлежит $F\}$.

Таким образом, язык — множество цепочек, приводящих автомат из состояния q_0 в одно из допускающих состояний. Если язык L есть $L(A)$ для некоторого ДКА A, то говорят, что L является *регулярным языком*.

Пример 2.5. Ранее упоминалось, что если A — ДКА из примера 2.1, то $L(A)$ — множество цепочек из 0 и 1, содержащих подцепочку 01. Если же A — ДКА из примера 2.4, то $L(A)$ — множество всех цепочек из 0 и 1, содержащих четное число 0 и четное число 1. □

2.2.6. Упражнения к разделу 2.2

2.2.1. На рис. 2.8 изображена игра "катящиеся шарики". Мраморный шарик бросается в точке A или B. Направляющие рычаги x_1, x_2 и x_3 заставляют шарик катиться влево или вправо. После того как шарик, столкнувшись с рычагом, проходит его, рычаг поворачивается в противоположную сторону, так что следующий шарик покатится уже в другую сторону.

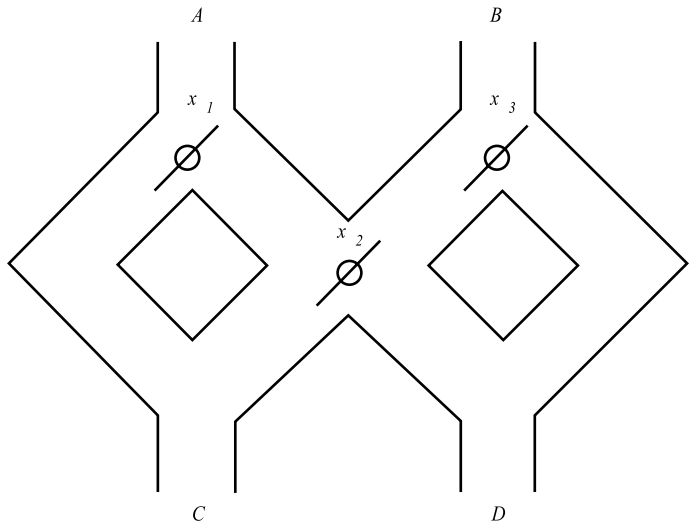


Рис. 2.8. Игра "катящиеся шарики"

Выполните следующее:

а) (*) постройте конечный автомат, моделирующий данную игру. Пусть A и B обозначают входы — те места, куда бросается шарик. Допустимость соответствует попаданию шарика в точку D, а недопустимость — в точку C;

б) (!) дайте нестрогое описание этого автомата;

в) предположим, что рычаги поворачиваются *до того*, как шарик проходит через них. Как это обстоятельство повлияет на ответ в частях (а) и (б)?

2.2.2. (***!**) Мы определяли $\hat{\delta}$, разбивая цепочку на цепочку и следующий за ней символ (в индуктивной части определения, уравнение 2.1). Однако неформально $\hat{\delta}$ представляется как описание событий вдоль пути с определенной цепочкой отметок переходов. Поэтому не должно иметь значения, как именно разбивать входную цепочку в определении $\hat{\delta}$. Покажите, что в действительности $\hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q, x), y)$ для всякого состояния q и цепочек x и y. *Указание*. Проведите индукцию по $|y|$.

2.2.3. (**!**) Покажите, что $\hat{\delta}(q, ax) = \hat{\delta}(\hat{\delta}(q, a), x)$ для всякого состояния q, цепочки x и входного символа a. *Указание*. Используйте упражнение 2.2.2.

2.2.4. Опишите ДКА, которые допускают следующие языки над алфавитом {0, 1}:

а) (*****) множество всех цепочек, оканчивающихся на 00;

б) множество всех цепочек, содержащих три нуля подряд;

в) множество цепочек, содержащих в качестве подцепочки 011.

2.2.5. (**!**) Опишите ДКА, допускающие такие языки над алфавитом {0, 1}:

а) множество всех цепочек, в которых всякая подцепочка из пяти последовательных символов содержит хотя бы два 0;

б) множество всех цепочек, у которых на десятой позиции справа стоит 1;

в) множество цепочек, которые начинаются или оканчиваются (или и то, и другое) последовательностью 01;

г) множество цепочек, в которых число нулей делится на пять, а число единиц — на три.

2.2.6. (**!!**) Опишите ДКА, которые допускают следующие языки над алфавитом {0, 1}:

а) (*****) множество всех цепочек, начинающихся с 1, и если рассматривать их как двоичное представление целого числа, то это число кратно 5. Например, цепочки 101, 1010 и 1111 принадлежат этому языку, а цепочки 0, 100 и 111 — нет;

б) множество всех цепочек, запись которых в *обратном порядке* образует двоичное представление целого числа, кратного 5. Примерами цепочек этого языка являются цепочки 0, 10011, 1001100 и 0101.

2.2.7. Пусть A есть некоторый ДКА и q — его состояние, у которого $\delta(q, a) = q$ для любого символа ввода a. Докажите индукцией по длине входной цепочки w, что $\hat{\delta}(q, w) = q$ для всякой входной цепочки w.

2.2.8. Пусть A — некоторый ДКА и a — входной символ, причем $\delta(q, a) = q$ для всех состояний q автомата A:

а) докажите индукцией по n, что $\hat{\delta}(q, a^n) = q$ для всякого $n \geq 0$, где a^n — цепочка, состоящая из n символов a;

б) покажите, что либо $\{a\}^* \subseteq L(A)$, либо $\{a\}^* \cap L(A) = \varnothing$.

2.2.9. (***!**) Пусть $A = (Q, \Sigma, \delta, q_0, \{q_f\})$ — ДКА. Предположим, что для всех a из Σ имеет место равенство $\delta(q_0, a) = \delta(q_f, a)$:

а) покажите, что для всех $w \neq \varepsilon$ верно равенство $\hat{\delta}(q_0, w) = \hat{\delta}(q_f, a)$;

б) покажите, что если x — непустая цепочка из $L(A)$, то для всех $k > 0$, x^k (т.е. цепочка x, записанная k раз) также принадлежит $L(A)$.

2.2.10. (***!**) Рассмотрим ДКА со следующей таблицей переходов:

	0	1
$\rightarrow A$	A	B
$*B$	B	A

Опишите неформально язык, допустимый данным ДКА, и докажите индукцией по длине входной цепочки, что ваше описание корректно. *Указание*. Устанавливая индуктивную гипотезу, разумно сформулировать утверждение о том, какие входные последовательности приводят в каждое состояние, а не только в допускающее.

2.2.11. (**!**) Выполните задание упражнения 2.2.10 со следующей таблицей переходов.

	0	1
$\rightarrow *A$	B	A
$*B$	C	A
C	C	C

2.3. Недетерминированные конечные автоматы

"Недетерминированный" конечный автомат, или НКА (NFA — Nondeterministic Finite Automaton), обладает свойством находиться в нескольких состояниях одновременно. Эту особенность часто представляют как свойство автомата делать "догадки" относительно его входных данных. Так, если автомат используется для поиска определенных цепочек символов (например, ключевых слов) в текстовой строке большой длины, то в начале поиска полезно "догадаться", что мы находимся в начале одной из этих цепочек, а затем использовать некоторую последовательность состояний для простой проверки того, что символ за символом появляется данная цепочка. Пример приложения такого типа приведен в разделе 2.4.

Прежде, чем перейти к приложениям, нужно определить недетерминированные конечные автоматы и показать, что всякий такой автомат допускает язык, допустимый некоторым ДКА, т.е. НКА допускают регулярные языки точно так же, как и ДКА. Однако есть причины рассматривать и НКА. Они зачастую более компактны и легче строятся, чем ДКА. Кроме того, хотя НКА всегда можно преобразовать в ДКА, последний может иметь экспоненциально больше состояний, чем НКА. К счастью, такие случаи довольно редки.

2.3.1. Неформальное описание недетерминированного конечного автомата

НКА, как и ДКА, имеют конечное множество состояний, конечное множество входных символов, одно начальное состояние и множество допускающих состояний. Есть также функция переходов, которая, как обычно, обозначается через δ. Различие между ДКА и НКА состоит в типе функции δ. В НКА δ есть функция, аргументами которой являются состояние и входной символ (как и в ДКА), а значением — множество, состоящее из нуля, одного или нескольких состояний (а не одно состояние, как в ДКА). Прежде чем дать строгое определение НКА, рассмотрим пример.

Пример 2.6. На рис. 2.9 изображен недетерминированный конечный автомат, допускающий те и только те цепочки из 0 и 1, которые оканчиваются на 01. Начальным является состояние q_0, и можно считать, что автомат находится в этом состоянии (а также, возможно, и в других состояниях) до тех пор, пока не "догадается", что на входе началась замыкающая подцепочка 01. Всегда существует вероятность того, что следующий символ не является начальным для замыкающей подцепочки 01, даже если это символ 0. Поэтому состояние q_0 может иметь переходы в себя как по 1, так и по 0.

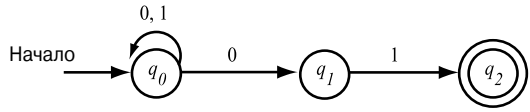


Рис. 2.9. НКА, допускающий цепочки, которые оканчиваются на 01

Если очередной входной символ — 0, то НКА может предположить, что уже началась замыкающая подпоследовательность 01. Таким образом, дуга с меткой 0 ведет из состояния q_0 в q_1. Заметим, что из q_0 выходят две дуги, отмеченные символом 0. НКА может перейти как в q_0, так и в q_1, и в действительности переходит в оба эти состояния. Мы убедимся в этом, когда дадим строгое определение НКА. В состоянии q_1 наш НКА проверяет, является ли следующий входной символ единицей. Если это так, то он переходит в состояние q_2 и считает цепочку допустимой.

Заметим, что из состояния q_1 дуга, отмеченная нулем, не выходит, а состояние q_2 вообще не имеет выходящих дуг. В этих состояниях пути НКА "умирают", хотя другие пути попрежнему существуют. В то время как ДКА имеет в каждом состоянии ровно одну выхо-

дящую дугу для каждого входного символа, для НКА такого ограничения нет. На примере рис. 2.9 мы убедились, что числом дуг может быть, например, нуль, один или два.

На рис. 2.10 видно, как НКА обрабатывает цепочку. Показано, что происходит, когда автомат (см. рис. 2.9) получает на вход последовательность 00101. Движение начинается из единственного начального состояния q_0. Когда прочитан первый символ 0, НКА может перейти в состояние либо q_0, либо q_1, а потому переходит в оба эти состояния. Эти два пути представлены на рис. 2.10 вторым столбцом.

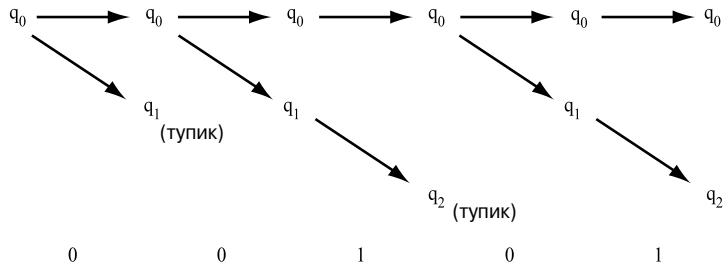


Рис. 2.10. Состояния, в которых находится НКА в процессе обработки цепочки 00101

Затем читается второй символ 0. Из состояния q_0 вновь можно перейти в q_0 и q_1. Однако состояние q_1 не имеет переходов по символу 0, и поэтому оно "умирает". Когда появляется третий входной символ, 1, нужно рассмотреть переходы из обоих состояний q_0 и q_1. Из состояния q_0 по 1 есть переход только в q_0, а из q_1 — только в q_2. Таким образом, прочитав цепочку 001, НКА находится в состояниях q_0 и q_2. НКА допускает ее, поскольку q_2 — допускающее состояние.

Однако чтение входных данных еще не завершено. Четвертый входной символ 0 приводит к тому, что ветвь, соответствующая q_2, отмирает, в то время как q_0 переходит в q_0 и q_1. По последнему символу 1 происходит переход из q_0 в q_0, а из q_1 — в q_2. Поскольку мы вновь попали в допускающее состояние, то цепочка 00101 допустима. □

2.3.2. Определение недетерминированного конечного автомата

Теперь мы определим формально понятия, связанные с недетерминированными конечными автоматами, выделив по ходу различия между ДКА и НКА. Структура НКА в основном повторяет структуру ДКА:

$$A = (Q, \Sigma, \delta, q_0, F).$$

Эти обозначения имеют следующий смысл.

1. Q есть конечное множество *состояний*.
2. Σ есть конечное множество *входных символов*.
3. q_0, один из элементов Q, — *начальное состояние*.
4. F, подмножество Q, — множество *заключительных* (или *допускающих*) состояний.

5. δ, *функция переходов*, — это функция, аргументами которой являются состояние из Q и входной символ из Σ, а значением — некоторое подмножество множества Q. Заметим, что единственное различие между НКА и ДКА состоит в типе значений функции δ. Для НКА — это множество состояний, а для ДКА — одиночное состояние.

Пример 2.7. НКА на рис. 2.9 можно формально задать, как

$(\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$,

где функция переходов δ задается таблицей на рис. 2.11. □

	0	1
$\rightarrow q_0$	$\{q_2, q_1\}$	$\{q_0\}$
q_1	$\varnothing$	$\{q_2\}$
$^*q_2$	$\varnothing$	$\varnothing$

Рис. 2.11. Таблица переходов *для ДКА, допускающего цепочки, которые оканчиваются на 01*

Заметим, что, как и для ДКА, функцию переходов НКА можно задавать таблицей. Единственное различие состоит в том, что в таблице для НКА на пересечениях строк и столбцов стоят множества, хотя, возможно, и *одноэлементные*, т.е. содержащие один элемент (*singleton*). Заметим также, что, когда из некоторого состояния по определенному символу перехода нет, на пересечении соответствующих строки и столбца должно стоять $\varnothing$ — пустое множество.

2.3.3. Расширенная функция переходов

Для НКА, так же, как и для ДКА, нам потребуется расширить функцию δ до функции $\hat{\delta}$, аргументами которой являются состояние q и цепочка входных символов w, а значением — множество состояний, в которые НКА попадает из состояния q, обработав цепочку w. Эта идея проиллюстрирована на рис. 2.10. По сути, $\hat{\delta}(q, w)$ есть столбец состояний, которые получаются при чтении цепочки w, при условии, что q — единственное состояние в первом столбце. Так, на рис. 2.10 видно, что $\hat{\delta}(q_0, 001) = \{q_0, q_2\}$. Формально $\hat{\delta}$ для НКА определяется следующим образом.

Базис. $\hat{\delta}(q, \varepsilon) = \{q\}$, т.е., не прочитав никаких входных символов, НКА находится только в том состоянии, в котором начинал.

Индукция. Предположим, цепочка w имеет вид $w = xa$, где a — последний символ цепочки w, а x — ее оставшаяся часть. Кроме того, предположим, что $\hat{\delta}(q, x) = \{p_1, p_2, \ldots, p_k\}$.

Пусть

$$\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, \ldots, r_m\}.$$

Тогда $\hat{\delta}(q, w) = \{r_1, r_2, \ldots, r_m\}$. Говоря менее формально, для того, чтобы найти $\hat{\delta}(q, w)$, нужно найти $\hat{\delta}(q, x)$, а затем совершить из всех полученных состояний все переходы по символу a.

Пример 2.8. Используем $\hat{\delta}$ для описания того, как НКА на рис. 2.9 обрабатывает цепочку 00101.

1. $\hat{\delta}(q_0, \varepsilon) = \{q_0\}$.
2. $\hat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$.
3. $\hat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \varnothing = \{q_0, q_1\}$.
4. $\hat{\delta}(q_0, 001) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$.
5. $\hat{\delta}(q_0, 0010) = \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \varnothing = \{q_0, q_1\}$.
6. $\hat{\delta}(q_0, 00101) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$.

Строка (1) повторяет основное правило. Строку (2) получаем, применяя δ к единственному состоянию q_0 из предыдущей строки. В результате получаем множество $\{q_0, q_1\}$. Строка (3) получается объединением двух состояний предыдущего множества и применением к каждому из них δ с входным символом 0, т.е. $\delta(q_0, 0) = \{q_0, q_1\}$, и $\delta(q_1, 0) = \varnothing$. Для того чтобы получить строку (4), берется объединение $\delta(q_0, 1) = \{q_0\}$ и $\delta(q_1, 1) = \{q_2\}$. Строки (5) и (6) получены так же, как строки (3) и (4). □

2.3.4. Язык НКА

По нашему описанию НКА допускает цепочку w, если в процессе чтения этой цепочки символов можно выбрать хотя бы одну последовательность переходов в следующие состояния так, чтобы прийти из начального состояния в одно из допускающих. Тот факт, что при другом выборе последовательности переходов по символам цепочки w мы можем попасть в недопускающее состояние или вообще не попасть ни в какое (т.е. последовательность состояний “умирает”), отнюдь не означает, что w не является допустимой для НКА в целом. Формально, если $A = (Q, \Sigma, \delta, q_0, F)$ — некоторый НКА, то

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F = \varnothing\}.$$

Таким образом, $L(A)$ есть множество цепочек w из Σ^*, для которых среди состояний $\hat{\delta}(q_0, w)$ есть хотя бы одно допускающее.

Пример 2.9. В качестве примера докажем формально, что НКА на рис. 2.9 допускает язык $L = \{w \mid w$ оканчивается на 01$\}$. Доказательство представляет собой совместную индукцию следующих трех утверждений, характеризующих три состояния.

1. $\hat{\delta}(q_0, w)$ содержит q_0 для всякой цепочки w.
2. $\hat{\delta}(q_0, w)$ содержит q_1 тогда и только тогда, когда w оканчивается на 0.
3. $\hat{\delta}(q_0, w)$ содержит q_2 тогда и только тогда, когда w оканчивается на 01.

Чтобы доказать эти утверждения, нужно рассмотреть, каким образом A может попасть в каждое из этих состояний, т.е. каким был последний входной символ, и в каком состоянии находился A непосредственно перед тем, как прочитал этот символ.

Поскольку язык этого автомата есть множество цепочек w, для которых $\hat{\delta}(q_0, w)$ содержит q_2 (так как q_2 — единственное допускающее состояние), то доказательство этих трех утверждений, в частности, доказательство 3, гарантирует, что язык данного НКА есть множество цепочек, оканчивающихся на 01. Доказательство этой теоремы представляет собой индукцию по $|w|$, длине цепочки w, начиная с нуля.

Базис. Если $|w| = 0$, то $w = \varepsilon$. В утверждении 1 говорится, что $\hat{\delta}(q_0, \varepsilon)$ содержит q_0. Но это действительно так в силу базисной части определения $\hat{\delta}$. Рассмотрим теперь утверждение 2. Мы знаем, что ε не заканчивается на 0, и, кроме того, опять же в силу базисной части определения $\hat{\delta}(q_0, \varepsilon)$ не содержит q_1. Таким образом, гипотезы утверждения 2 типа "тогда и только тогда" в обе стороны ложны. Поэтому само утверждение является истинным в обе стороны. Доказательство утверждения 3, по сути, повторяет доказательство утверждения 2.

Индукция. Допустим, что $w = xa$, где a есть символ 0 или 1. Можно предположить, что утверждения 1–3 выполняются для x. Нужно доказать их для w, предположив, что $|w| = n + 1$, а $|x| = n$. Предполагая, что гипотеза индукции верна для n, докажем ее для $n + 1$.

1. Нам известно, что $\hat{\delta}(q_0, x)$ содержит q_0. Поскольку по обоим входным символам 0 и 1 существуют переходы из q_0 в себя, то $\hat{\delta}(q_0, w)$ также содержит q_0. Таким образом, утверждение 1 доказано для w.
2. (*Достаточность*) Предположим, что w оканчивается на 0, т.е. $a = 0$. Применяя утверждение 1 к x, получаем, что $\hat{\delta}(q_0, x)$ содержит q_0. Поскольку по символу 0 существует переход из q_0 в q_1, заключаем, что $\hat{\delta}(q_0, w)$ содержит q_1.

   (*Необходимость*) Допустим, что $\hat{\delta}(q_0, w)$ содержит q_1. По диаграмме на рис. 2.9 видно, что единственная возможность попасть в состояние q_1 реализуется, когда w имеет вид $x0$. Это доказывает необходимость в утверждении 2.
3. (*Достаточность*) Предположим, что w оканчивается на 01. Тогда если $w = xa$, то мы знаем, что $a = 1$, а x оканчивается на 0. Применив утверждение 2 к x, получим, что $\hat{\delta}(q_0, x)$ содержит q_1. Поскольку по символу 1 существует переход из q_1 в q_2, приходим к выводу, что $\hat{\delta}(q_0, w)$ содержит q_2.

   (*Необходимость*) Допустим, что $\hat{\delta}(q_0, w)$ содержит q_2. По диаграмме на рис. 2.9 видно, что в состояние q_2 можно попасть тогда и только тогда, когда w имеет вид $x1$

и $\hat{\delta}(q_0, x)$ содержит q_1. Применяя утверждение 2 к x, получим, что x оканчивается на 0. Таким образом, w оканчивается на 01, и утверждение 3 доказано.

□

2.3.5. Эквивалентность детерминированных и недетерминированных конечных автоматов

Для многих языков, в частности, для языка цепочек, оканчивающихся на 01 (пример 2.6), построить соответствующий НКА гораздо легче, чем ДКА. Несмотря на это, всякий язык, который описывается некоторым НКА, можно также описать и некоторым ДКА. Кроме того, этот ДКА имеет, как правило, примерно столько же состояний, сколько и НКА, хотя часто содержит больше переходов. Однако в худшем случае наименьший ДКА может содержать 2^n состояний, в то время как НКА для того же самого языка имеет всего n состояний.

В доказательстве того, что ДКА обладают всеми возможностями НКА, используется одна важная конструкция, называемая *конструкцией подмножеств*, поскольку включает построение всех подмножеств множества состояний НКА. Вообще, в доказательствах утверждений об автоматах часто по одному автомату строится другой. Для нас конструкция подмножеств важна в качестве примера того, как один автомат описывается в терминах состояний и переходов другого автомата без знания специфики последнего.

Построение подмножеств начинается, исходя из НКА $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Целью является описание ДКА $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$, у которого $L(N) = L(D)$. Отметим, что входные алфавиты этих двух автоматов совпадают, а начальное состояние D есть множество, содержащее только начальное состояние N. Остальные компоненты D строятся следующим образом.

- Q_D есть множество всех подмножеств Q_N, или *булеан* множества Q_N. Отметим, что если Q_N содержит n состояний, то Q_D будет содержать уже 2^n состояний. Однако часто не все они достижимы из начального состояния автомата D. Такие недостижимые состояния можно “отбросить”, поэтому фактически число состояний D может быть гораздо меньше, чем 2^n.

- F_D есть множество подмножеств S множества Q_N, для которых $S \cap F_N \neq \varnothing$, т.е. F_D состоит из всех множеств состояний N, содержащих хотя бы одно допускающее состояние N.

- Для каждого множества $S \subseteq Q_N$ и каждого входного символа a из Σ

  $$\delta_D(S, a) = \bigcup_{p \text{ из } S} \delta_N(p, a).$$

  Таким образом, для того, чтобы найти $\delta_D(S, a)$, мы рассматриваем все состояния p из S, ищем те состояния N, в которые можно попасть из состояния p по

символу a, а затем берем объединение множеств найденных состояний по всем состояниям p.

Пример 2.10. Пусть N — автомат на рис. 2.9, допускающий цепочки, которые оканчиваются на 01. Поскольку множество состояний N есть $\{q_0, q_1, q_2\}$, то конструкция подмножеств дает ДКА с $2^3 = 8$ состояниями, отвечающими всем подмножествам, составленным из этих трех состояний. На рис. 2.12 приведена таблица переходов для полученных восьми состояний. Объясним вкратце, как были получены элементы этой таблицы.

	0	1
$\varnothing$	$\varnothing$	$\varnothing$
$\to \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\varnothing$	$\{q_2\}$
$^*\{q_2\}$	$\varnothing$	$\varnothing$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$^*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$^*\{q_1, q_2\}$	$\varnothing$	$\{q_2\}$
$^*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Рис. 2.12. Полная конструкция подмножеств для автомата на рис. 2.9

Заметим, что данная таблица, элементами которой являются множества, соответствует детерминированному конечному автомату, поскольку состояния построенного ДКА *сами являются множествами*. Для ясности можно переобозначить состояния. Например, $\varnothing$ обозначить как A, $\{q_0\}$ — как B и т.д. Таблица переходов для ДКА на рис. 2.13 определяет в точности тот же автомат, что и на рис. 2.12, и из ее вида понятно, что элементами таблицы являются одиночные состояния ДКА.

	0	1
A	A	A
$\to B$	E	B
C	A	D
$^*D$	A	A
E	E	F
$^*F$	E	B
$^*G$	A	D
$^*H$	E	F

Рис. 2.13. Переименование состояний на рис. 2.12

Начиная в состоянии B, из всех восьми состояний мы можем попасть только в состояния B, E и F. Остальные пять состояний из начального недостижимы, и поэтому их можно исключить из таблицы. Часто можно избежать построения элементов таблицы переходов для всех подмножеств, что требует экспоненциального времени. Для этого выполняется следующее “ленивое вычисление” подмножеств.

Базис. Мы точно знаем, что одноэлементное множество, состоящее из начального состояния N, является достижимым.

Индукция. Предположим, мы установили, что множество состояний S является достижимым. Тогда для каждого входного символа a нужно найти множество состояний $\delta_D(S, a)$. Найденные таким образом множества состояний также будут достижимы.

Элементарный пример: нам известно, что $\{q_0\}$ есть одно из состояний ДКА D. Находим, что $\delta_D(\{q_0\}, 0) = \{q_0, q_1\}$ и $\delta_D(\{q_0\}, 1) = \{q_0\}$. Оба эти факта следуют из диаграммы переходов для автомата на рис. 2.9; как видно, по символу 0 есть переходы из q_0 в q_0 и q_1, а по символу 1 — только в q_0. Таким образом, получена вторая строка таблицы переходов ДКА на рис. 2.12.

Одно из найденных множеств, $\{q_0\}$, уже рассматривалось. Но второе, $\{q_0, q_1\}$, — новое, и переходы для него нужно найти: $\delta_D(\{q_0, q_1\}, 0) = \{q_0, q_1\}$ и $\delta_D(\{q_0, q_1\}, 1) = \{q_0, q_2\}$. Проследить последние вычисления можно, например, так:

$$\delta_D(\{q_0, q_1\}, 1) = \delta_N(q_0, 1) \cup \delta_N(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}.$$

Теперь получена пятая строка таблицы на рис. 2.12 и одно новое состояние $\{q_0, q_2\}$. Аналогичные вычисления показывают, что

$$\delta_D(\{q_0, q_2\}, 0) = \delta_N(q_0, 0) \cup \delta_N(q_2, 0) = \{q_0, q_1\} \cup \varnothing = \{q_0, q_1\},$$
$$\delta_D(\{q_0, q_2\}, 1) = \delta_N(q_0, 1) \cup \delta_N(q_2, 1) = \{q_0\} \cup \varnothing = \{q_0\}.$$

Эти вычисления дают шестую строку таблицы на рис. 2.12, но при этом не получено ни одного нового множества состояний.

Итак, конструкция подмножеств сошлась; известны все допустимые состояния и соответствующие им переходы. Полностью ДКА показан на рис. 2.14. Заметим, что он имеет лишь три состояния. Это число случайно оказалось равным числу состояний НКА на рис. 2.9, по которому строился этот ДКА. Но ДКА на рис. 2.14 имеет шесть переходов, а автомат на рис. 2.9 — лишь четыре. □

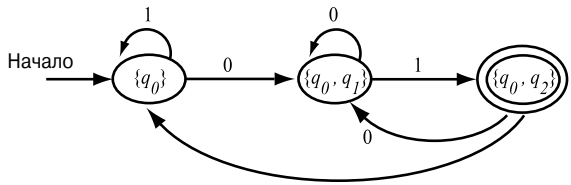


Рис. 2.14. ДКА, построенный по НКА на рис. 2.9

На последнем примере мы убедились, что конструкция подмножеств успешно работает. Теперь докажем это формально. По прочтении последовательности символов w построенный нами ДКА находится в состоянии, представляющем собой множество состояний НКА, в которые тот попадает, прочитав эту цепочку. Но допускающие состояния ДКА — это состояния, которые содержат хотя бы одно допускающее состояние НКА, а допустимыми для НКА являются цепочки, приводящие его хотя бы в одно из допускающих состояний. Поэтому можно заключить, что ДКА и НКА допускают в точности одни и те же цепочки. Следовательно, они допускают один и тот же язык.

Теорема 2.11. Если ДКА $D = (Q_D, \Sigma, \delta_D, q_0, F_D)$ построен по НКА $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ посредством конструкции подмножеств, то $L(D) = L(N)$.

Доказательство. Вначале с помощью индукции по $|w|$ покажем, что

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w).$$

Заметим, что как для одной, так и для другой функции $\hat{\delta}$, значением является множество состояний из Q_N. При этом $\hat{\delta}_D$ интерпретирует его как состояние из Q_D (являющегося булеаном Q_N), а $\hat{\delta}_N$ — как подмножество Q_N.

Базис. Пусть $|w| = 0$, т.е. $w = \varepsilon$. Из базисных частей определений $\hat{\delta}$ для ДКА и НКА имеем, что как $\hat{\delta}_D(\{q_0\}, \varepsilon)$, так и $\hat{\delta}_N(q_0, \varepsilon)$ равны $\{q_0\}$.

Индукция. Пусть w имеет длину $n + 1$, и предположим, что утверждение справедливо для цепочки длины n. Разобьем w на $w = xa$, где a — последний символ цепочки w. Согласно гипотезе индукции $\hat{\delta}_D(\{q_0\}, x) = \hat{\delta}_N(q_0, x)$. Пусть оба эти множества состояний N представляют собой $\{p_1, p_2, \ldots, p_k\}$.

По индуктивной части определения для НКА

$$\hat{\delta}_N(q_0, w) = \bigcup_{i=1}^{k} \delta_N(p_i, a). \tag{2.2}$$

С другой стороны, конструкция подмножеств дает

$$\delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a). \tag{2.3}$$

Теперь, подставляя (2.3) в индуктивную часть определения для ДКА и используя тот факт, что $\hat{\delta}_D(\{q_0\}, x) = \{p_1, p_2, \ldots, p_k\}$, получаем:

$$\hat{\delta}_D(\{q_0\}, w) = \delta_D(\hat{\delta}_D(q_0, x), a)) = \delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a). \tag{2.4}$$

Таким образом, из уравнений (2.2) и (2.4) видно, что $\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$. Далее, замечая, что и D, и N допускают w тогда и только тогда, когда $\hat{\delta}_D(\{q_0\}, w)$ или $\hat{\delta}_N(q_0, w)$ соответственно содержат некоторое состояние из F_N, получаем полное доказательство того, что $L(D) = L(N)$. □

Теорема 2.12. *Язык L допустим некоторым ДКА тогда и только тогда, когда он допускается некоторым НКА.*

Доказательство. *Достаточность* следует из конструкции подмножеств и теоремы 2.11.

(*Необходимость*) Доказательство этой части не представляет трудности; нам нужно лишь перейти от ДКА к идентичному НКА. Диаграмму переходов для некоторого ДКА можно рассматривать неформально как диаграмму переходов для некоторого НКА, причем последний имеет по любому входному символу лишь один переход. Точнее, пусть $D = (Q, \Sigma, \delta_D, q_0, F)$ есть некоторый ДКА. Определим $N = (Q, \Sigma, \delta_N, q_0, F)$ как эквивалентный ему НКА, где δ_N определена следующим правилом.

- Если $\delta_D(q, a) = p$, то $\delta_N(q, a) = \{p\}$.

Индукцией по $|w|$ легко показать, что, если $\hat{\delta}_D(q, w) = p$, то

$$\hat{\delta}_N(q_0, w) = \{p\}.$$

Доказательство предоставляется читателю. Как следствие, D допускает w тогда и только тогда, когда N допускает w, т.е. $L(D) = L(N)$. □

2.3.6. Плохой случай для конструкции подмножеств

В примере 2.10 мы обнаружили, что число состояний ДКА и число состояний НКА одинаково. Как мы уже говорили, ситуация, когда количества состояний НКА и построенного по нему ДКА примерно одинаковы, на практике встречается довольно часто. Однако при переходе от НКА к ДКА возможен и экспоненциальный рост числа состояний, т.е. все 2^n состояний, которые могут быть построены по НКА, имеющему n состояний, оказываются достижимыми. В следующем примере мы немного не дойдем до этого предела, но будет ясно, каким образом наименьший ДКА, построенный по НКА с $n + 1$ состояниями, может иметь 2^n состояний.

Пример 2.13. Рассмотрим НКА на рис. 2.15. $L(N)$ есть множество всех цепочек из 0 и 1, у которых n-м символом с конца является 1. Интуиция подсказывает, что ДКА D, допускающий данный язык, должен помнить последние n прочитанных символов. Поскольку всего имеется 2^n последовательностей, состоящих из последних n символов, то при числе состояний D меньше 2^n нашлось бы состояние q, в которое D попадает по прочтении двух разных последовательностей, скажем, $a_1a_2\ldots a_n$ и $b_1b_2\ldots b_n$.

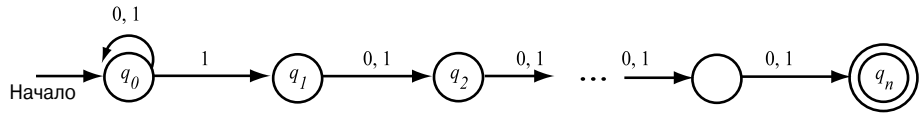


Рис. 2.15. Этот НКА не имеет эквивалентного ДКА, число состояний которого меньше 2^n

Поскольку последовательности различны, они должны различаться символом в некоторой позиции, например, $a_i \neq b_i$. Предположим (с точностью до симметрии), что $a_i = 1$ и $b_i = 0$. Если $i = 1$, то состояние q должно быть одновременно и допускающим, и недо-

пускающим, поскольку последовательность $a_1a_2\ldots a_n$ допустима (n-й символ с конца есть 1), а $b_1b_2\ldots b_n$ — нет. Если же $i > 1$, то рассмотрим состояние p, в которое D попадает из состояния q по прочтении цепочки из $i - 1$ нулей. Тогда p вновь должно одновременно и быть, и не быть допускающим, так как цепочка $a_ia_{i+1}\ldots a_n00\ldots0$ допустима, а $b_ib_{i+1}\ldots b_n00\ldots0$ — нет.

Теперь рассмотрим, как работает НКА N на рис. 2.15. Существует состояние q_0, в котором этот НКА находится всегда, независимо от входных символов. Если следующий символ — 1, то N может "догадаться", что эта 1 есть n-й символ с конца. Поэтому одновременно с переходом в q_0 НКА N переходит в состояние q_1. Из состояния q_1 по любому символу N переходит в состояние q_2. Следующий символ переводит N в состояние q_3 и так далее, пока $n - 1$ последующий символ не переведет N в допускающее состояние q_n. Формальные утверждения о работе состояний N выглядят следующим образом.

1. N находится в состоянии q_0 по прочтении любой входной последовательности w.
2. N находится в состоянии q_i ($i = 1, 2, \ldots, n$) по прочтении входной последовательности w тогда и только тогда, когда i-й символ с конца w есть 1, т.е. w имеет вид $x1a_1a_2\ldots a_{i-1}$, где a_j — входные символы.

"Принцип голубятни"

В примере 2.13 мы использовали важный технический прием, применяемый в различных обоснованиях. Он называется *принципом голубятни*.[4] Простыми словами, если у вас больше голубей, чем клеток для них, и каждый голубь залетает в одну из клеток, то хотя бы в одной клетке окажется больше одного голубя. В нашем примере "голубями" являются последовательности из n элементов, а "клетками" — состояния. Поскольку состояний меньше, чем последовательностей, две различные последовательности должны вести в одно и то же состояние.

Принцип голубятни может показаться очевидным, но в действительности он зависит от конечности числа клеток. Поэтому он применим к автоматам с конечным числом состояний и неприменим к автоматам, число состояний которых бесконечно.

Чтобы убедиться в том, что конечность числа клеток существенна, рассмотрим случай, когда есть бесконечное число клеток, соответствующих целым числам 1, 2, …. Пронумеруем голубей числами 0, 1, 2, …, т.е. так, чтобы их было на одного больше, чем клеток. Тогда можно поместить голубя с номером i в $(i + 1)$-ю клетку для всех $i \geq 0$. Тогда каждый из бесконечного числа голубей попадет в клетку, и никакие два голубя не окажутся в одной клетке.

Мы не доказываем эти утверждения формально. Скажем лишь, что доказательство представляет собой несложную индукцию по $|w|$, как в примере 2.9. Завершая доказа-

тельство того, что данный автомат допускает именно те цепочки, у которых на n-й позиции с конца стоит 1, мы рассмотрим утверждение (2) при $i = n$. В нем говорится, что N находится в состоянии q_n тогда и только тогда, когда n-й символ с конца есть 1. Но q_n является единственным допускающим состоянием, поэтому это условие также точно характеризует множество цепочек, допускаемых автоматом N. □

2.3.7. Упражнения к разделу 2.3

2.3.1. Преобразуйте следующий НКА в эквивалентный НКА.

	0	1
$\rightarrow p$	$\{p, q\}$	$\{p\}$
q	$\{r\}$	$\{r\}$
r	$\{s\}$	$\varnothing$
$*s$	$\{s\}$	$\{s\}$

2.3.2. Преобразуйте следующий НКА в эквивалентный ДКА

	0	1
$\rightarrow p$	$\{q, s\}$	$\{q\}$
$*q$	$\{r\}$	$\{q, r\}$
r	$\{s\}$	$\{p\}$
$*s$	$\varnothing$	$\{p\}$

2.3.3. Преобразуйте следующий НКА в эквивалентный ДКА и опишите неформально язык, который он допускает.

	0	1
$\rightarrow p$	$\{p, q\}$	$\{p\}$
q	$\{r, s\}$	$\{t\}$
r	$\{p, r\}$	$\{t\}$
$*s$	$\varnothing$	$\varnothing$
$*t$	$\varnothing$	$\varnothing$

2.3.4. (!) Найдите недетерминированные конечные автоматы, которые допускают следующие языки. Постарайтесь максимально использовать возможности недетерминизма:

---

[4] В русскоязычной литературе часто употребляется термин “принцип Дирихле”. — *Прим. ред.*

а) (*) множество цепочек над алфавитом {0, 1, …, 9}, последняя цифра которых встречается еще где-то в них;

Дьявольские состояния и ДКА с неопределенными переходами

Формально определяя ДКА, мы требовали, чтобы по каждому входному символу он имел переход в одно и только одно состояние. Однако иногда бывает более удобно устроить ДКА таким образом, чтобы он "умирал" в ситуации, когда входная последовательность уже не может быть допустимой, что бы к ней ни добавлялось. Рассмотрим, например, автомат на рис. 1.2, единственной задачей которого является распознавание одиночного ключевого слова **then**. Чисто технически, данный автомат не является ДКА, так как для каждого состояния в нем отсутствуют переходы по большинству входных символов.

Но этот автомат является НКА. И если посредством конструкции подмножеств превратить его в ДКА, то вид автомата практически не изменится, хотя при этом в нем появится *дьявольское состояние*, которое не является допускающим и переходит само в себя по любому символу. Это состояние соответствует $\varnothing$ — пустому множеству состояний автомата на рис. 1.2.

Вообще говоря, можно добавить дьявольское состояние в любой автомат, если он имеет *не более* одного перехода для всякого состояния и входного символа. Тогда добавляется переход в дьявольское состояние из остальных состояний q по всем символам, для которых переход из q не определен. В результате получается ДКА в точном смысле слова. Поэтому иногда будем говорить об автомате как о ДКА, если он имеет *не более* одного перехода из любого состояния по любому входному символу, а не только в случае, когда он имеет *только один* переход.

б) множество цепочек над алфавитом {0, 1, …, 9}, последняя цифра цепочки которых больше нигде в них не встречается;

в) множество цепочек из 0 и 1, в которых содержится два 0, разделенных позициями в количестве, кратном 4. Отметим, что нуль позиций можно также считать кратным 4.

2.3.5. В части "необходимость" теоремы 2.12 было пропущено индуктивное доказательство того, что если $\hat{\delta}_D(q, w) = p$, то $\hat{\delta}_N(q, w) = \{p\}$, где индукция велась бы по $|w|$. Приведите это доказательство.

2.3.6. (!) В замечании "Дьявольские состояния и ДКА с неопределенными переходами" утверждалось, что если НКА N по каждому входному символу содержит переход не более чем в одно состояние (т.е. $\delta(q, a)$ есть не более чем одноэлементное множество), то ДКА D, построенный по N с помощью конструкции подмножеств, содержит точно те же состояния и переходы, что и N,

плюс переходы в новое дьявольское состояние из тех состояний и по тем входным символам, для которых переходы N не определены. Докажите это утверждение.

2.3.7. В примере 2.13 утверждалось, что НКА находится в состоянии q_i (i = 1, 2, …, n) по прочтении входной последовательности w тогда и только тогда, когда i-й символ с конца есть 1. Докажите это утверждение.

2.4. Приложение: поиск в тексте

В предыдущем разделе была рассмотрена абстрактная “проблема”, состоявшая в том, что нужно было выяснить, оканчивается ли данная последовательность двоичных чисел на 01. В этом разделе мы увидим, что подобного рода абстракции прекрасно подходят для описания таких реальных задач, возникающих в приложениях, как поиск в сети Internet и извлечение информации из текста.

2.4.1. Поиск цепочек в тексте

В век Internet и электронных библиотек с непрерывным доступом обычной является следующая проблема. Задано некоторое множество слов, и требуется найти все документы, в которых содержится одно (или все) из них. Популярным примером такого процесса служит работа поисковой машины, которая использует специальную технологию поиска, называемую *обращенными индексами* (inverted indexes). Для каждого слова, встречающегося в Internet (а их около 100,000,000), хранится список адресов всех мест, где оно встречается. Машины с очень большим объемом оперативной памяти обеспечивают постоянный доступ к наиболее востребованным из этих списков, позволяя многим людям одновременно осуществлять поиск документов.

В методе обращенных индексов конечные автоматы не используются, но этот метод требует значительных затрат времени для копирования содержимого сети и переписывания индексов. Существует множество смежных приложений, в которых применить технику обращенных индексов нельзя, зато можно с успехом использовать методы на основе автоматов. Те приложения, для которых подходит технология поиска на основе автоматов, имеют следующие отличительные особенности.

1. Содержимое хранилища текста, в котором производится поиск, быстро меняется. Вот два примера:
    а) каждый день аналитики ищут статьи со свежими новостями по соответствующим темам. К примеру, финансовый аналитик может искать статьи с определенными аббревиатурами ценных бумаг или названиями компаний;
    б) “робот-закупщик” по требованию клиента отслеживает текущие цены по определенным наименованиям товаров. Он извлекает из сети страницы, содер-

жащие каталоги, а затем просматривает эти страницы в поисках информации о ценах по конкретному наименованию.

2. Документы, поиск которых осуществляется, не могут быть каталогизированы. Например, очень непросто отыскать в сети все страницы, содержащие информацию обо всех книгах, которые продает компания Amazon.com, поскольку эти страницы генерируются как бы "на ходу" в ответ на запрос. Однако мы можем отправить запрос на книги по определенной теме, скажем, "конечные автоматы", а затем искать в той части текста, которая содержится на появившихся страницах, определенное слово, например слово "прекрасно".

2.4.2. Недетерминированные конечные автоматы для поиска в тексте

Пусть нам дано множество слов, которые мы в дальнейшем будем называть *ключевыми словами*, и нужно отыскать в тексте места, где встречается любое из этих слов. В подобных приложениях бывает полезно построить недетерминированный конечный автомат, который, попадая в одно из допускающих состояний, дает знать, что встретил одно из ключевых слов. Текст документа, символ за символом, подается на вход НКА, который затем распознает в нем ключевые слова. Существует простая форма НКА, распознающего множество ключевых слов.

1. Есть начальное состояние с переходом в себя по каждому входному символу, например, печатному символу ASCII при просмотре текста. Начальное состояние можно представлять себе, как "угадывание" того, что ни одно из ключевых слов еще не началось, даже если несколько букв одного из ключевых слов уже прочитано.
2. Для каждого ключевого слова $a_1a_2\ldots a_k$ имеется k состояний, скажем $q_1, q_2, \ldots, q_k$. Для входного символа a_1 есть переход из начального состояния в q_1, для входного символа a_2 — переход из q_1 в q_2 и т.д. Состояние q_k является допускающим и сигнализирует о том, что ключевое слово $a_1a_2\ldots a_k$ обнаружено.

Пример 2.14. Предположим, что мы хотим построить НКА, распознающий слова `web` и `ebay`. Диаграмма переходов данного НКА, изображенная на рис. 2.16, построена с помощью изложенных выше правил. Начальное состояние — это состояние 1, а Σ обозначает множество печатаемых символов ASCII. Состояния 2–4 отвечают за распознавание слова `web`, а состояния 5–8 — за распознавание слова `ebay`. □

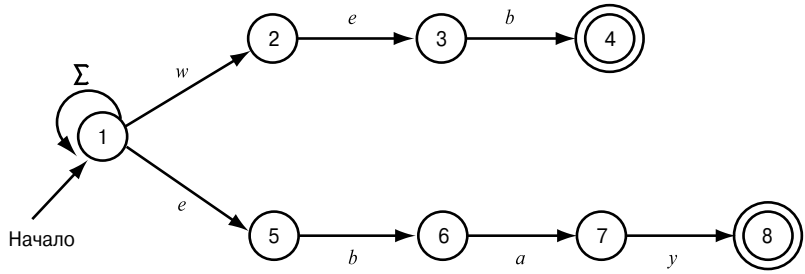


Рис. 2.16. НКА, осуществляющий поиск слов `web` *и* `ebay`

Безусловно, НКА — не программа. Для реализации данного НКА у нас есть две основные возможности.

1. Написать программу, имитирующую работу данного НКА путем вычисления множества состояний, в которых он находится по прочтении каждого из входных символов. Такая реализация была рассмотрена на рис. 2.10.
2. Преобразовать данный НКА в эквивалентный ему ДКА, используя конструкцию подмножеств. Затем непосредственно реализовать ДКА.

В некоторых программах, обрабатывающих текст, таких, например, как наиболее продвинутые версии команды `grep` (`egrep` и `fgrep`) операционной системы UNIX, используется комбинация этих двух подходов. Однако в нашем случае более удобно преобразование к ДКА, так как это, во-первых, просто, а во-вторых, гарантирует, что число состояний при этом не возрастет.

2.4.3. ДКА, распознающий множество ключевых слов

Конструкция подмножеств применима к любому НКА. Но, применяя ее к НКА, построенному по множеству ключевых слов, как в разделе 2.4.2, мы обнаружим, что число состояний соответствующего ДКА никогда не превосходит числа состояний этого НКА. А поскольку нам известно, что при переходе к ДКА в худшем случае может произойти экспоненциальный рост числа состояний, то последнее замечание ободряет и объясняет, почему для распознавания ключевых слов так часто используется метод построения соответствующего НКА, а по нему — ДКА. Правила, в соответствии с которыми строятся состояния ДКА, состоят в следующем:

а) если q_0 — начальное состояние НКА, то $\{q_0\}$ — одно из состояний ДКА;

б) допустим, p — одно из состояний НКА, и НКА попадает в него из начального состояния по пути, отмеченному символами $a_1a_2\ldots a_m$. Тогда одним из состояний ДКА является множество, состоящее из q_0, p и всех остальных состояний НКА, в которые можно попасть из q_0 по пути, отмеченному суффиксом (окончанием) цепочки $a_1a_2\ldots a_m$, т.е. последовательностью символов вида $a_j\,a_{j+1}\ldots a_m$.

Отметим, что, вообще, всякому состоянию p НКА соответствует одно состояние ДКА. Но на шаге (б) может получиться так, что два состояния на самом деле дают одно и то же множество состояний НКА и поэтому сливаются в одно состояние ДКА. Например, если два ключевых слова начинаются с одной и той же буквы, скажем, a, то два состояния НКА, в которые он попадает из q_0 по дуге с меткой a, дают одно и то же множество состояний НКА и, следовательно, сливаются в одно состояние ДКА.

Пример 2.15. На рис. 2.17 показано, как по НКА (см. рис. 2.16) построен ДКА. Каждое из состояний ДКА расположено на том же самом месте, что и состояние p, по которому оно построено, в соответствии с приведенным выше правилом (б). Рассмотрим, например, состояние {1, 3, 5}, обозначенное для краткости как 135. Это состояние было построено по состоянию 3. Как и всякое множество состояний нашего ДКА, оно включает состояние 1. Кроме того, оно включает состояние 5, так как в него автомат попадает из 1 по окончанию `e` цепочки `we`, приводящей в состояние 3 на рис. 2.16.

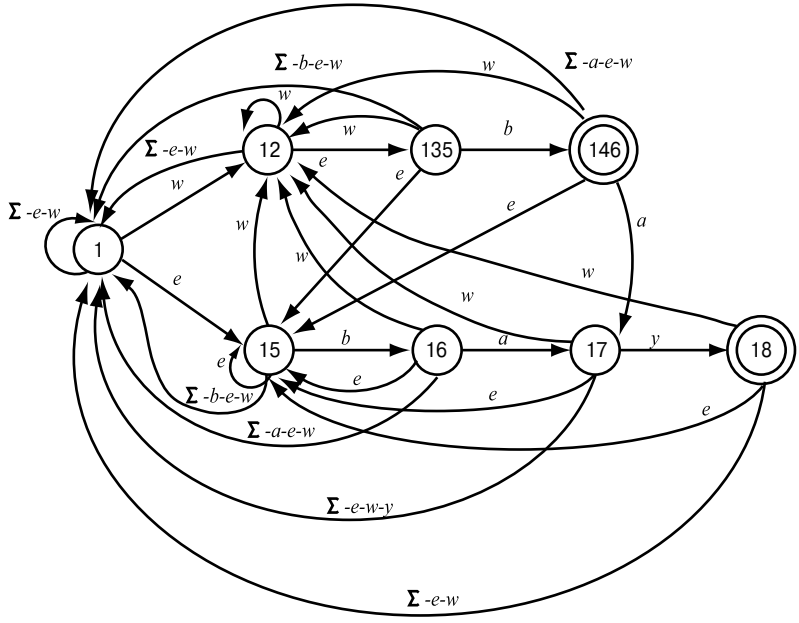


Рис. 2.17. Преобразовование НКА, изображенного на рис. 2.16, в ДКА

Для каждого из состояний ДКА переходы могут быть найдены с помощью конструкции подмножеств. Но тут можно поступить проще. Для всякого множества состояний, включающего начальное состояние q_0 и некоторые другие состояния $p_1, p_2, \ldots, p_n$, и для каждого входного символа x вычислим те состояния НКА, в которые по x переходят p_i. Тогда данное состояние ДКА, т.е. $\{q_0, p_1, p_2, \ldots, p_n\}$, будет иметь переход по символу x в состояние ДКА, содержащее q_0 и все те состояния, в которые переходят p_i. По всем тем символам x, по которым переходов ни из одного p_i нет, данный ДКА будет иметь пере-

ход по символу x в состояние, содержащее q_0 и все те состояния исходного НКА, которые достигаются переходом из q_0 по дуге с меткой x.

Рассмотрим состояние 135 на рис. 2.17. НКА на рис. 2.16 имеет переходы по символу b из состояний 3 и 5 в состояния 4 и 6, соответственно. Следовательно, 135 по b переходит в 146. По входному символу e переходов из состояний НКА 3 и 5 нет, но есть переход из 1 в 5. Таким образом, по e ДКА переходит из 135 в 15. Точно так же, по w 135 переходит в 12.

По любому другому символу x переходов из состояний 3 и 5 нет, а состояние 1 имеет переход только в себя. Таким образом, по любому символу из Σ, за исключением b, e и w, 135 переходит в 1. Обозначим это множество как $\Sigma - b - e - w$, а также используем подобные обозначения для других множеств, получающихся из Σ удалением нескольких символов. □

2.4.4. Упражнения к разделу 2.4

2.4.1. Постройте НКА, распознающие следующие множества цепочек:

а) (*) `abc`, `abd` и `aacd`. Входным алфавитом считать $\{a, b, c, d\}$;

б) `0101`, `101` и `011`;

в) `ab`, `bc` и `ca`. Входным алфавитом считать $\{a, b, c\}$.

2.4.2. Преобразуйте ваши НКА из упражнения 2.4.1 в ДКА.

2.5. Конечные автоматы с эпсилон-переходами

Рассмотрим еще одно обобщение понятия конечного автомата. Придадим автомату новое "свойство" — возможность совершать переходы по ε, пустой цепочке, т.е. спонтанно, не получая на вход никакого символа. Эта новая возможность, как и недетерминизм, введенный в разделе 2.3, не расширяет класса языков, допустимых конечными автоматами, но дает некоторое дополнительное "удобство программирования". Кроме того, рассмотрев в разделе 3.1 регулярные выражения, мы увидим, что последние тесно связаны с НКА, имеющими ε-переходы. Такие автоматы будем называть *ε-НКА*. Они оказываются полезными при доказательстве эквивалентности между классами языков, задаваемых конечными автоматами и регулярными выражениями.

2.5.1. Использование ε-переходов

Вначале будем оперировать с ε-НКА неформально, используя диаграммы переходов с ε в качестве возможной метки. В следующих примерах автомат можно рассматривать как допускающий последовательности меток, среди которых могут быть ε, вдоль путей

из начального состояния в допускающее. Но при этом каждое ε вдоль пути "невидимо", т.е. в последовательность ничего не добавляет.

Пример 2.16. На рис. 2.18 изображен ε-НКА, допускающий десятичные числа, которые состоят из следующих элементов.

1. Необязательный знак + или –.
2. Цепочка цифр.
3. Разделяющая десятичная точка.
4. Еще одна цепочка цифр. Эта цепочка, как и цепочка (2), может быть пустой, но хотя бы одна из них непуста.

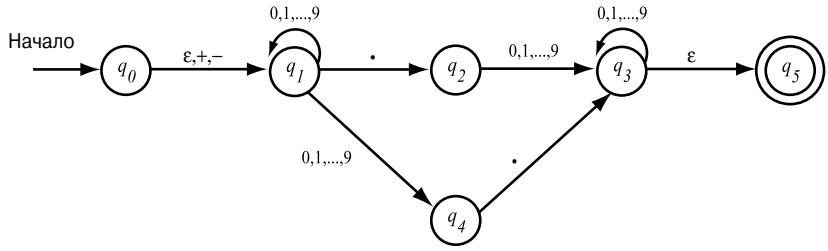
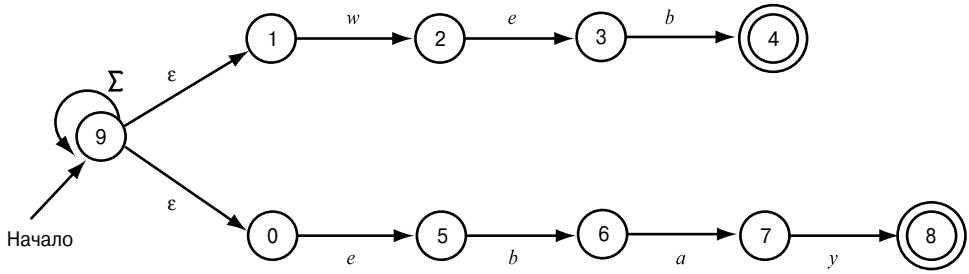


Рис. 2.18. ε-*НКА, допускающий десятичные числа*

Особого интереса заслуживает переход из состояния q_0 в q_1 по любому из символов +, – или ε. Состояние q_1, таким образом, представляет ситуацию, когда прочитан знак числа, если он есть, но не прочитана ни одна из цифр, ни десятичная точка. Состояние q_2 соответствует ситуации, когда только что прочитана десятичная точка, а цифры целой части числа либо уже были прочитаны, либо нет. В состоянии q_4 уже наверняка прочитана хотя бы одна цифра, но еще не прочитана десятичная точка. Таким образом, q_3 интерпретируется как ситуация, когда мы прочитали десятичную точку и хотя бы одну цифру слева или справа от нее. Мы можем оставаться в состоянии q_3, продолжая читать цифры, но можем и "догадаться", что цепочка цифр закончена, и спонтанно перейти в допускающее состояние q_5. □

Пример 2.17. Метод распознавания множества ключевых слов, предложенный в примере 2.14, можно упростить, разрешив ε-переходы. Например, НКА на рис. 2.16, распознающий ключевые слова `web` и `ebay`, можно реализовать и с помощью ε-переходов, как показано на рис. 2.19. Суть в том, что для каждого ключевого слова строится полная последовательность состояний, как если бы это было единственное слово, которое автомат должен распознавать. Затем добавляется новое начальное состояние (состояние 9 на рис. 2.19) с ε-переходами в начальные состояния автоматов для каждого из ключевых слов. □



2.19. Использование ε-переходов для распознавания ключевых слов

2.5.2. Формальная запись ε-НКА

ε-НКА можно представлять точно так же, как и НКА, с той лишь разницей, что функция переходов должна содержать информацию о переходах по ε. Формально, ε-НКА A можно представить в виде $A = (Q, \Sigma, \delta, q_0, F)$, где все компоненты имеют тот же смысл, что и для НКА, за исключением δ, аргументами которой теперь являются состояние из Q и элемент множества $\Sigma \cup \{\varepsilon\}$, т.е. либо некоторый входной символ, либо ε. Никаких недоразумений при этом не возникает, поскольку мы оговариваем, что ε, символ пустой цепочки, не является элементом алфавита Σ.

Пример 2.18. ε-НКА на рис. 2.18 можно формально представить как

$$E = (\{q_0, q_1, \ldots, q_5\}, \{., +, -, 0, 1, \ldots, 9\}, \delta, q_0, \{q_5\}),$$

где функция переходов δ определена таблицей переходов на рис. 2.20. □

	ε	+, –	.	0, 1, …, 9
q_0	$\{q_1\}$	$\{q_1\}$	$\varnothing$	$\varnothing$
q_1	$\varnothing$	$\varnothing$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_3\}$
q_3	$\{q_5\}$	$\varnothing$	$\varnothing$	$\{q_3\}$
q_4	$\varnothing$	$\varnothing$	$\{q_3\}$	$\varnothing$
q_5	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

Рис. 2.20. Таблица переходов к рис. 2.18

2.5.3. Что такое ε-замыкание

Дадим формальное определение расширенной функции переходов для ε-НКА, которое приведет к определению допустимости цепочек и языков для данного типа автоматов и в конце концов поможет понять, почему ДКА могут имитировать работу ε-НКА. Одна-

ко прежде нужно определить одно из центральных понятий, так называемое *ε-замыкание* состояния. Говоря нестрого, мы получаем ε-замыкание состояния q, совершая все возможные переходы из этого состояния, отмеченные ε. Но после совершения этих переходов и получения новых состояний снова выполняются ε-переходы, уже из новых состояний, и т.д. В конце концов, мы находим все состояния, в которые можно попасть из q по любому пути, каждый переход в котором отмечен символом ε. Формально мы определяем ε-замыкание, ECLOSE, рекурсивно следующим образом.

Базис. ECLOSE(q) содержит состояние q.

Индукция. Если ECLOSE(q) содержит состояние p, и существует переход, отмеченный ε, из состояния p в состояние r, то ECLOSE(q) содержит r. Точнее, если δ есть функция переходов рассматриваемого ε-НКА и ECLOSE(q) содержит p, то ECLOSE(q) содержит также все состояния из $\delta(p, \varepsilon)$.

Пример 2.19. У автомата, изображенного на рис. 2.18, каждое состояние является собственным ε-замыканием, за исключением того, что ECLOSE(q_0) = $\{q_0, q_1\}$ и ECLOSE(q_3) = $\{q_3, q_5\}$. Это объясняется тем, что имеется лишь два ε-перехода, один из которых добавляет q_1 в ECLOSE(q_0), а другой — q_5 в ECLOSE(q_3).

На рис. 2.21 приведен более сложный пример. Для данного в нем набора состояний, который может быть частью некоторого ε-НКА, мы можем заключить, что

ECLOSE(1) = {1, 2, 3, 4, 6}.

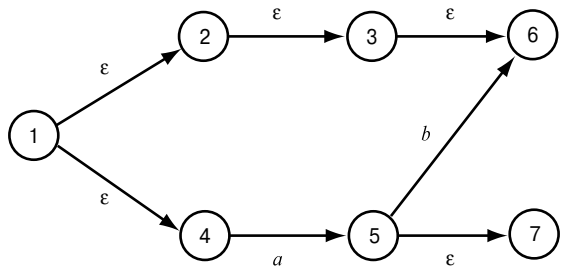


Рис. 2.21. Несколько состояний и переходов

В каждое из этих состояний можно попасть из состояния 1, следуя по пути, отмеченному исключительно ε. К примеру, в состояние 6 можно попасть по пути 1→2→3→6. Состояние 7 не принадлежит ECLOSE(1), поскольку, хотя в него и можно попасть из состояния 1, в соответствующем пути содержится переход 4→5, отмеченный не ε. И не важно, что в состояние 6 можно попасть из состояния 1, следуя также по пути 1→4→5→6, в котором присутствует не ε-переход. Существования одного пути, отмеченного только ε, уже достаточно для того, чтобы состояние 6 содержалось в ECLOSE(1). □

2.5.4. Расширенные переходы и языки ε-НКА

С помощью ε-замыкания легко объяснить, как будут выглядеть переходы ε-НКА для заданной последовательности входных (не-ε) символов. Исходя из этого, можно определить, что означает для ε-НКА допустимость входных данных.

Пусть $E = (Q, \Sigma, \delta, q_0, F)$ — некоторый ε-НКА. Для отображения того, что происходит при чтении некоторой последовательности символов, сначала определим $\hat{\delta}$ — расширенную функцию переходов. Замысел таков: определить $\hat{\delta}(q, w)$ как множество состояний, в которые можно попасть по путям, конкантенации меток вдоль которых дают цепочку w. При этом, как и всегда, символы ε, встречающиеся вдоль пути, ничего не добавляют к w. Соответствующее рекурсивное определение $\hat{\delta}$ имеет следующий вид.

Базис. $\hat{\delta}(q, \varepsilon) = \text{ECLOSE}(q)$. Таким образом, если ε — метка пути, то мы можем совершать переходы лишь по дугам с меткой ε, начиная с состояния q; это дает нам в точности то же, что и ECLOSE(q).

Индукция. Предположим, что w имеет вид xa, где a — последний символ w. Отметим, что a есть элемент Σ и, следовательно, не может быть ε, так как ε не принадлежит Σ. Мы вычисляем $\hat{\delta}(q, w)$ следующим образом.

1. Пусть $\{p_1, p_2, \ldots, p_k\}$ есть $\hat{\delta}(q, x)$, т.е. p_i — это все те и только те состояния, в которые можно попасть из q по пути, отмеченному x. Этот путь может оканчиваться одним или несколькими ε-переходами, а также содержать и другие ε-переходы.

2. Пусть $\bigcup_{i=1}^{k} \delta(p_i, a)$ есть множество $\{r_1, r_2, \ldots, r_m\}$, т.е. нужно совершить все переходы, отмеченные символом a, из тех состояний, в которые мы можем попасть из q по пути, отмеченному x. Состояния r_i — лишь *некоторые* из тех, в которые мы можем попасть из q по пути, отмеченному w. В остальные такие состояния можно попасть из состояний r_i посредством переходов с меткой ε, как описано ниже в (3).

3. $\hat{\delta}(q, w) = \bigcup_{j=1}^{m} \text{ECLOSE}(r_j)$. На этом дополнительном шаге, где мы берем замыкание и добавляем все выходящие из q пути, отмеченные w, учитывается возможность существования дополнительных дуг, отмеченных ε, переход по которым может быть совершен после перехода по последнему "непустому" символу a.

Пример 2.20. Вычислим $\hat{\delta}(q_0, 5.6)$ для ε-НКА на рис. 2.18. Для этого выполним следующие шаги.

- $\hat{\delta}(q_0, \varepsilon) = \text{ECLOSE}(q_0) = \{q_0, q_1\}$.
- Вычисляем $\hat{\delta}(q_0, 5)$ следующим образом.

1. Находим переходы по символу 5 из состояний q_0 и q_1, полученных при вычислении $\hat{\delta}(q_0, \varepsilon)$: $\delta(q_0, 5) \cup \delta(q_1, 5) = \{q_1, q_4\}$.
2. Находим ε-замыкание элементов, вычисленных на шаге (1). Получаем: $\text{ECLOSE}(q_1) \cup \text{ECLOSE}(q_4) = \{q_1\} \cup \{q_4\} = \{q_1, q_4\}$, т.е. множество $\hat{\delta}(q_0, 5)$.

Эта двушаговая схема применяется к следующим двум символам.

- Вычисляем $\hat{\delta}(q_0, 5.)$.
  1. Сначала $\delta(q_1, .) \cup \delta(q_4, .) = \{q_2\} \cup \{q_3\} = \{q_2, q_3\}$.
  2. Затем $\hat{\delta}(q_0, 5.) = \text{ECLOSE}(q_2) \cup \text{ECLOSE}(q_3) = \{q_2\} \cup \{q_3, q_5\} = \{q_2, q_3, q_5\}$.
- Наконец, вычисляем $\hat{\delta}(q_0, 5.6)$.
  1. Сначала $\delta(q_2, 6) \cup \delta(q_3, 6) \cup \delta(q_5, 6) = \{q_3\} \cup \{q_3\} \cup \varnothing = \{q_3\}$.
  2. Затем $\hat{\delta}(q_0, 5.6) = \text{ECLOSE}(q_3) = \{q_3, q_5\}$.

□

Теперь можно определить язык ε-НКА $E = (Q, \Sigma, \delta, q_0, F)$ так, как и было задумано ранее: $L(E) = \{w \mid \hat{\delta}(q, w) \cap F \neq \varnothing\}$. Таким образом, язык E — это множество цепочек w, которые переводят автомат из начального состояния хотя бы в одно из допускающих. Так, в примере 2.20 мы видели, что $\hat{\delta}(q_0, 5.6)$ содержит допускающее состояние q_5, поэтому цепочка 5.6 принадлежит языку ε-НКА.

2.5.5. Устранение ε-переходов

Для всякого ε-НКА E можно найти ДКА D, допускающий тот же язык, что и E. Поскольку состояния D являются подмножествами из состояний E, то используемая конструкция очень напоминает конструкцию подмножеств. Единственное отличие состоит в том, что нужно присоединить еще и ε-переходы E, применив механизм ε-замыкания.

Пусть $E = (Q_E, \Sigma, \delta_E, q_E, F_E)$. Тогда эквивалентный ДКА

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

определяется следующим образом.

1. Q_D есть множество подмножеств Q_E. Точнее, как мы выясним, для D допустимыми состояниями являются только *ε-замкнутые подмножества* Q_E, т.е. такие множества $S \subseteq Q_E$, для которых $S = \text{ECLOSE}(S)$. Иначе говоря, ε-замкнутые множества состояний S — это такие множества, у которых всякий ε-переход из состояния, принадлежащего S, приводит снова в состояние из S. Заметим, что $\varnothing$ есть ε-замкнутое множество.
2. $q_D = \text{ECLOSE}(q_0)$, т.е., замыкая множество, содержащее только начальное состояние E, мы получаем начальное состояние D. Заметим, что это правило отличается от использованного ранее в конструкции подмножеств — там за начальное состояние по-

строенного автомата принималось множество, содержащее только начальное состояние данного НКА.

3. F_D — это такие множества состояний, которые содержат хотя бы одно допускающее состояние автомата E. Таким образом, $F_D = \{S \mid S$ принадлежит Q_D и $S \cap F_E \neq \varnothing\}$.

4. $\delta_D(S, a)$ для всех a из Σ и множеств S из Q_D вычисляется следующим образом:

   а) пусть $S = \{p_1, p_2, \ldots, p_k\}$;

   б) вычислим $\bigcup_{i=1}^{k} \delta(p_i, a)$; пусть это будет множество $\{r_1, r_2, \ldots, r_m\}$;

   в) тогда $\delta_D(S, a) = \bigcup_{j=1}^{m}$ ECLOSE(r_j).

Пример 2.21. Удалим ε-переходы из ε-НКА (см. рис. 2.18), который далее называется E. По E мы строим ДКА D, изображенный на рис. 2.22. Для того чтобы избежать излишнего нагромождения, мы удалили на рис. 2.22 дьявольское состояние $\varnothing$ и все переходы в него. Поэтому, глядя на рис. 2.22, следует иметь в виду, что у каждого состояния есть еще дополнительные переходы в состояние $\varnothing$ по тем входным символам, для которых переход на рисунке отсутствует. Кроме того, у состояния $\varnothing$ есть переход в себя по любому входному символу.

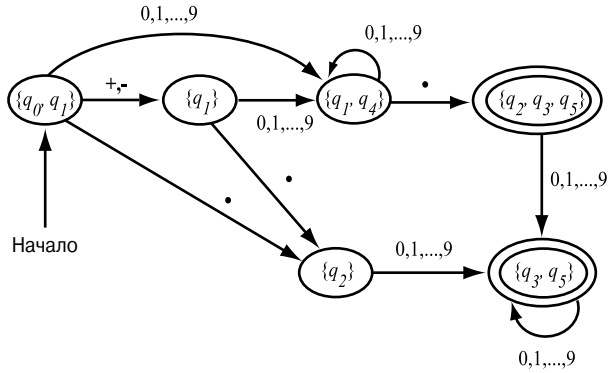


Рис. 2.22. ДКА D*, полученный устранением* ε*-переходов на рис. 2.18*

Поскольку начальное состояние E — это q_0, начальным состоянием D является ECLOSE(q_0), т.е. множество $\{q_0, q_1\}$. В первую очередь нужно найти состояния, в которые переходят q_0 и q_1 по различным символам из Σ; напомним, что это знаки плюс и минус, точка и цифры от 0 до 9. Как видно на рис. 2.18, по символам + и - q_1 никуда не переходит, в то время как q_0 переходит в q_1. Таким образом, чтобы вычислить $\delta_D(\{q_0, q_1\}, +)$, нужно взять ε-замыкание $\{q_1\}$. Поскольку ε-переходов, выходящих из q_1, нет, получаем, что $\delta_D(\{q_0, q_1\}, +) = \{q_1\}$. Точно так же находится $\delta_D(\{q_0, q_1\}, -) = \{q_1\}$. Эти два перехода изображены одной дугой на рис. 2.22.

Теперь найдем $\delta_D(\{q_0, q_1\}, .)$. Как видно на рис. 2.18, по точке q_0 никуда не переходит, а q_1 переходит в q_2. Поэтому нужно взять замыкание $\{q_2\}$. Но состояние q_2 является собственным замыканием, так как ε-переходов из него нет.

И, наконец, в качестве примера перехода из $\{q_0, q_1\}$ по произвольной цифре найдем $\delta_D(\{q_0, q_1\}, 0)$. По цифре q_0 никуда не переходит, а q_1 переходит сразу в q_1 и q_4. Так как ни одно из этих состояний не имеет выходящих ε-переходов, делаем вывод, что $\delta_D(\{q_0, q_1\}, 0) = \{q_1, q_4\}$. Последнее равенство справедливо для всех цифр.

Итак, мы объяснили, как строятся дуги на рис. 2.22. Остальные переходы находятся аналогично; проверка предоставляется читателю. Поскольку q_5 есть единственное допускающее состояние E, допускающими состояниями D являются те его достижимые состояния, которые содержат q_5. На рис. 2.22 эти два состояния, $\{q_3, q_5\}$ и $\{q_2, q_3, q_5\}$, отмечены двойными кружками. □

Теорема 2.22. *Язык* L допускается некоторым ε-НКА тогда и только тогда, когда L допускается некоторым ДКА.

Доказательство. (*Достаточность*) Доказательство в эту сторону просто. Допустим, $L = L(D)$ для некоторого ДКА D. Преобразуем D в ε-НКА E, добавив переходы $\delta(q, \varepsilon) = \varnothing$ для всех состояний q автомата D. Чисто технически нужно также преобразовать переходы D по входным символам к виду НКА-переходов. Например, $\delta_D(q, a) = p$ нужно превратить в множество, содержащее только состояние p, т.е. $\delta_E(q, a) = \{p\}$. Таким образом, E и D имеют одни и те же переходы, но при этом, совершенно очевидно, E не содержит переходов по ε, выходящих из какого-либо состояния.

(*Необходимость*) Пусть $E = (Q_E, \Sigma, \delta_E, q_E, F_E)$ — некоторый ε-НКА. Применим описанную выше модифицированную конструкцию подмножеств для построения ДКА

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D).$$

Нужно доказать, что $L(D) = L(E)$. Для этого покажем, что расширенные функции переходов E и D совпадают. Формально покажем индукцией по длине w, что $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_0, w)$.

Базис. Если $|w| = 0$, то $w = \varepsilon$. По определению замыкания $\hat{\delta}_E(q_0, \varepsilon) = \text{ECLOSE}(q_0)$. Кроме того, по определению начального состояния D $q_D = \text{ECLOSE}(q_0)$. Наконец, нам известно, что для любого ДКА $\hat{\delta}(p, \varepsilon) = p$, каково бы ни было состояние p. Поэтому, в частности, $\hat{\delta}_D(q_D, \varepsilon) = \text{ECLOSE}(q_0)$. Таким образом, доказано, что $\hat{\delta}_E(q_0, \varepsilon) = \hat{\delta}_D(q_D, \varepsilon)$.

Индукция. Предположим, что $w = xa$, где a — последний символ цепочки w, и что для x утверждение справедливо. Таким образом, $\hat{\delta}_E(q_0, x) = \hat{\delta}_D(q_D, x)$. Пусть оба эти множества состояний представляют собой $\{p_1, p_2, \ldots, p_k\}$.

По определению $\hat{\delta}$ для ε-НКА находим $\hat{\delta}_E(q_0, w)$ следующим образом.

1. Пусть $\bigcup_{i=1}^{k} \delta_E(p_i, a)$ есть $\{r_1, r_2, \ldots, r_m\}$.

2. Тогда $\hat{\delta}_E(q_0, w) = \bigcup_{j=1}^{m} \text{ECLOSE}(r_j)$.

Внимательно рассмотрев, как ДКА D строится посредством описанной выше модифицированной конструкции подмножеств, мы видим, что $\hat{\delta}_D(\{p_1, p_2, \ldots, p_k\}, a)$ построено с помощью описанных только что шагов (1) и (2). Таким образом, значение $\hat{\delta}_D(q_D, w)$, т.е. $\hat{\delta}_D(\{p_1, p_2, \ldots, p_k\}, a)$, совпадает с $\hat{\delta}_E(q_0, w)$. Тем самым доказаны равенство $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_D, w)$ и индуктивная часть теоремы. □

2.5.6. Упражнения к разделу 2.5

2.5.1. (*) Рассмотрите следующий ε-НКА и:

	ε	a	b	c
$\to p$	$\varnothing$	$\{p\}$	$\{q\}$	$\{r\}$
q	$\{p\}$	$\{q\}$	$\{r\}$	$\varnothing$
$*r$	$\{q\}$	$\{r\}$	$\varnothing$	$\{p\}$

а) найдите ε-замыкание каждого из состояний;

б) выпишите все цепочки, длина которых не более 3, допустимые данным автоматом;

в) преобразуйте данный автомат в ДКА.

2.5.2. Выполните задание упражнения 2.5.1 со следующим ε-НКА.

	ε	a	b	c
$\to p$	$\{q, r\}$	$\varnothing$	$\{q\}$	$\{r\}$
q	$\varnothing$	$\{p\}$	$\{r\}$	$\{p, q\}$
$*r$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

2.5.3. Постройте ε-НКА, которые допускают следующие языки. Для упрощения построений используйте, по возможности, ε-переходы:

а) множество всех цепочек, состоящих из нуля или нескольких символов a, после которых стоит нуль или несколько символов b, и вслед за ними нуль или несколько символов c;

б) (!) множество цепочек, состоящих либо из повторяющихся один или несколько раз фрагментов 01, либо из повторяющихся один или несколько раз фрагментов 010;

в) (!) множество цепочек из 0 и 1, в которых хотя бы на одной из последних десяти позиций стоит 1.

Резюме

- *Детерминированные конечные автоматы*. ДКА имеет конечное число состояний и конечное множество входных символов. Одно из состояний выделено как начальное, и нуль или несколько состояний являются допускающими. Функция переходов определяет, как изменяются состояния при обработке входных символов.
- *Диаграммы переходов*. Автомат удобно представлять в виде графа, в котором вершины соответствуют состояниям, а дуги, отмеченные входными символами, соответствуют переходам из состояния в состояние. Начальное состояние отмечается стрелкой, а допускающие состояния выделяются двойными кружками.
- *Язык автомата*. Автомат допускает цепочки. Цепочка является допустимой, если, стартуя в начальном состоянии и обрабатывая символы этой цепочки по одному, автомат переходит в некоторое допускающее состояние. В терминах диаграмм **переходов** цепочка является допустимой, если она состоит из символов, отмечающих путь из начального состояния в одно из допускающих.
- *Недетерминированный конечный автомат*. НКА отличается от ДКА тем, что НКА может иметь любое число переходов (в том числе, ни одного) из данного состояния в по данному входному символу.
- *Конструкция подмножеств*. Множества состояний НКА можно рассматривать как состояния некоторого ДКА. Таким образом, мы можем преобразовать НКА в ДКА, допускающий тот же самый язык.
- *ϵ-переходы*. Можно расширить понятие НКА, разрешив переходы по пустой цепочке, т.е. вообще без входных символов. Расширенные таким образом НКА могут быть преобразованы в ДКА, допускающие те же самые языки.
- *Приложения типа "поиск в тексте"*. Недетерминированные конечные автоматы представляют собой удобный способ моделирования программы поиска одного или нескольких ключевых слов в больших массивах текста. Эти автоматы либо непосредственно имитируются с помощью программы, либо вначале преобразуются в ДКА, а затем уже реализуются в виде программы.

Литература

Принято считать, что начало формальному изучению систем с конечным числом состояний было положено в работе [2]. Однако эта работа была посвящена не теперешним ДКА, а так называемой вычислительной модели "нейронных сетей". ДКА, в общеприня-

том смысле слова, были введены независимо в нескольких различных вариантах в работах [1], [3] и [4]. Материал по недетерминированным автоматам и конструкции подмножеств взят из работы [5].

1. D. A. Huffman, “The synthesis of sequential switching circuits”, *J.Franklin Inst.* **257**:3–4 (1954), pp. 161–190 and 275–303.
2. W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervuous activity”, *Bull. Math. Biophysics* **5** (1943), pp. 115–133. (Маккалок У.С., Питтс Э. Логическое исчисление идей, относящихся к нервной активности. / сб. “*Автоматы*”. — М.: ИЛ, 1956. — С. 362–384.)
3. G. H. Mealy, “A method for synthesizing sequential circuits”, *Bell System Technical Journal* 34:5 (1955), pp. 1045–1079.
4. E. F. Moore, “Gedanken experiments on sequential machines”, in [6], pp. 129–153. (Мур Э.Ф. Умозрительные эксперименты с последовательностными машинами. / сб. “*Автоматы*”. — М.: ИЛ, 1956. — С. 179–210.)
5. M. O. Rabin and D. Scott, “Finite automata and their decision problems”, *IBM J. Research and Development* **3**:2 (1959), pp. 115–125. (Рабин М.О., Скотт Д. Конечные автоматы и задачи их разрешения. — *Кибернетический сборник, вып*. 4. — М.: ИЛ, 1962. — С. 56–71.)
6. C. E. Shannon and J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956. (Шеннон К.Э., Мак-Карти Дж. Теория автоматов. / сб. “*Автоматы*”. — М.: ИЛ, 1956.)

ГЛАВА 3

Регулярные выражения и языки

В этой главе вводится система записи “регулярных выражений”. Такие выражения представляют собой еще один способ определения языков, рассмотренный вкратце в разделе 1.1.2. Регулярные выражения можно рассматривать также как “язык программирования” для описания некоторых важных приложений, например, программ текстового поиска или компонентов компилятора. Регулярные выражения тесно связаны с НКА и могут служить удобной альтернативой для описания программных компонентов.

Определив регулярные выражения, покажем, что они могут задавать регулярные, и только регулярные, языки. Обсудим также, каким образом регулярные выражения используются в различных системах программного обеспечения. Затем исследуем алгебраические законы для регулярных выражений. Эти законы во многом подобны алгебраическим законам арифметики, однако между алгебрами регулярных и арифметических выражений есть и существенные различия.

3.1. Регулярные выражения

Перейдем от “машинного” задания языков с помощью ДКА и НКА к алгебраическому описанию языков с помощью регулярных выражений. Установим, что регулярные выражения определяют точно те же языки, что и различные типы автоматов, а именно, регулярные языки. В то же время, в отличие от автоматов, регулярные выражения позволяют определять допустимые цепочки декларативным способом. Поэтому регулярные выражения используются в качестве входного языка во многих системах, обрабатывающих цепочки. Рассмотрим два примера.

1. Команды поиска, например, команда `grep` операционной системы UNIX или аналогичные команды для поиска цепочек, которые можно встретить в Web-броузерах или системах форматирования текста. В таких системах регулярные выражения используются для описания шаблонов, которые пользователь ищет в файле. Различные поисковые системы преобразуют регулярное выражение либо в ДКА, либо в НКА и применяют этот автомат к файлу, в котором производится поиск.
2. Генераторы лексических анализаторов, такие как Lex или Flex. Напомним, что лексический анализатор — это компонент компилятора, разбивающий исходную про-

грамму на логические единицы (*лексемы*), которые состоят из одного или нескольких символов и имеют определенный смысл. Примерами лексем являются ключевые слова (например `while`), идентификаторы (любая буква, за которой следует нуль или несколько букв и/или цифр) и такие знаки, как + или <=. Генератор лексических анализаторов получает формальные описания лексем, являющиеся по существу регулярными выражениями, и создает ДКА, который распознает, какая из лексем появляется на его входе.

3.1.1. Операторы регулярных выражений

Регулярные выражения обозначают (задают, или представляют) языки. В качестве простого примера рассмотрим регулярное выражение $\mathbf{01}^*+\mathbf{10}^*$. Оно определяет язык всех цепочек, состоящих либо из одного нуля, за которым следует любое количество единиц, либо из одной единицы, за которой следует произвольное количество нулей. В данный момент мы не рассчитываем, что читатель знает, как интерпретируются регулярные выражения, поэтому наше утверждение о языке, задаваемом этим выражением, пока должно быть принято на веру. Чтобы понять, почему наша интерпретация заданного регулярного выражения правильна, необходимо определить все использованные в этом выражении символы, поэтому вначале познакомимся со следующими тремя операциями над языками, соответствующими операторам регулярных выражений[1].

1. *Объединение* двух языков L и M, обозначаемое $L \cup M$, — это множество цепочек, которые содержатся либо в L, либо в M, либо в обоих языках. Например, если $L = \{001, 10, 111\}$ и $M = \{\varepsilon, 001\}$, то $L \cup M = \{\varepsilon, 10, 001, 111\}$.
2. *Конкатенация* языков L и M — это множество цепочек, которые можно образовать путем дописывания к любой цепочке из L любой цепочки из M. Выше в разделе 1.5.2 было дано определение конкатенации двух цепочек: результатом ее является запись одной цепочки вслед за другой. Конкатенация языков обозначается либо точкой, либо вообще никак не обозначается, хотя оператор конкатенации часто называют "точкой". Например, если $L = \{001, 10, 111\}$ и $M = \{\varepsilon, 001\}$, то $L.M$, или просто LM, — это $\{001, 10, 111, 001001, 10001, 111001\}$. Первые три цепочки в LM — это цепочки из L, соединенные с ε. Поскольку ε является единицей (нейтральным элементом) для операции конкатенации, результирующие цепочки будут такими же, как и цепочки из L. Последние же три цепочки в LM образованы путем соединения каждой цепочки из L со второй цепочкой из M, т.е. с 001. Например, 10 из L, соединенная с 001 из M, дает 10001 для LM.

[1] Будем употреблять также термины "*регулярные операции*" и "*регулярные операторы*", принятые в русскоязычной литературе. — *Прим. ред.*

3. *Итерация* (“*звездочка*”, или *замыкание Клини*[2]) языка L обозначается L^* и представляет собой множество всех тех цепочек, которые можно образовать путем конкатенации любого количества цепочек из L. При этом допускаются повторения, т.е. одна и та же цепочка из L может быть выбрана для конкатенации более одного раза. Например, если $L = \{0, 1\}$, то L^* — это все цепочки, состоящие из нулей и единиц. Если $L = \{0, 11\}$, то в L^* входят цепочки из нулей и единиц, содержащие четное количество единиц, например, цепочки 011, 11110 или ε, и не входят цепочки 01011 или 101. Более формально язык L^* можно представить как бесконечное объединение $\bigcup_{i\geq 0} L^i$, где $L^0 = \{\varepsilon\}$, $L^1 = L$ и L^i для $i > 1$ равен $LL\ldots L$ (конкатенация i копий L).

Пример 3.1. Поскольку идея итерации языка может показаться довольно сложной, рассмотрим несколько примеров. Для начала возьмем $L = \{0, 11\}$. $L^0 = \{\varepsilon\}$ независимо от языка L; нулевая степень означает выбор нулевого количества цепочек из языка L. $L^1 = L$, что означает выбор одной цепочки из L. Таким образом, первые два члена в разложении L^* дают $\{\varepsilon, 0, 11\}$.

Далее рассмотрим L^2. Выберем две цепочки из L и, поскольку их можно выбирать с повторениями, получим четыре варианта, которые дают $L^2 = \{00, 011, 110, 1111\}$. Аналогично, L^3 представляет собой множество цепочек, образованных троекратным выбором из двух цепочек языка L. Следовательно, L^3 имеет вид

$$\{000, 0011, 0110, 1100, 01111, 11011, 11110, 111111\}$$

Для вычисления L^* необходимо вычислить L^i для каждого i и объединить все эти языки. Язык L^i содержит 2^i элементов. Хотя каждое множество L^i конечно, объединение бесконечного числа множеств L^i образует, как правило, бесконечный язык, что справедливо, в частности, и для нашего примера.

Пусть теперь L — множество всех цепочек, состоящих из нулей. Заметим, что такой язык бесконечен, в отличие от предыдущего примера, где был рассмотрен конечный язык. Однако нетрудно увидеть, что представляет собой L^*. Как всегда, $L^0 = \{\varepsilon\}$, $L^1 = L$. L^2 — это множество цепочек, которые можно образовать, если взять одну цепочку из нулей и соединить ее с другой цепочкой из нулей. В результате получим цепочку, также состоящую из нулей. Фактически, любую цепочку из нулей можно записать как конкатенацию двух цепочек из нулей (не забывайте, что ε — тоже “цепочка из нулей”; она всегда может быть одной из двух соединяемых цепочек). Следовательно, $L^2 = L$. Аналогично, $L^3 = L$ и так далее. Таким образом, бесконечное объединение $L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$ совпадает с L в том особом случае, когда язык L является множеством всех нулевых цепочек.

---

[2] Термин “замыкание Клини” связан с именем С. К. Клини, который ввел понятие регулярного выражения и, в частности, эту операцию.

В качестве последнего примера рассмотрим $\varnothing^* = \{\varepsilon\}$. Заметим, что $\varnothing^0 = \{\varepsilon\}$, тогда как $\varnothing^i$ для любого $i \geq 1$ будет пустым множеством, поскольку мы не можем выбрать ни одной цепочки из пустого множества. Фактически, $\varnothing$ является одним из всего двух языков, итерация которых *не является* бесконечным множеством. □

3.1.2. Построение регулярных выражений

Все алгебры начинаются с некоторых элементарных выражений. Обычно это константы и/или переменные. Применяя определенный набор операторов к этим элементарным выражениям и уже построенным выражениям, можно конструировать более сложные выражения. Обычно необходимо также иметь некоторые методы группирования операторов и операндов, например, с помощью скобок. К примеру, обычная арифметическая алгебра начинается с констант (целые и действительные числа) и переменных и позволяет нам строить более сложные выражения с помощью таких арифметических операторов, как + или ×.

Использование оператора "звездочка"

Впервые оператор "звездочка" появился в разделе 1.5.2, где применялся к алфавиту, например Σ^*. С помощью этого оператора образуются все цепочки, символы которых принадлежат алфавиту Σ. Оператор итерации в значительной мере похож на "звездочку", хотя существует несколько тонких отличий в типах.

Предположим, что L — это язык, содержащий цепочки длины 1, и для каждого символа a из Σ существует цепочка a в L. Тогда, хотя L и Σ "выглядят" одинаково, они принадлежат к различным типам: L представляет собой множество цепочек, а Σ — множество символов. С другой стороны, L^* обозначает тот же язык, что и Σ^*.

Алгебра регулярных выражений строится по такой же схеме: используются константы и переменные для обозначения языков и операторы для обозначения трех операций из раздела 3.1.1 — объединение, точка и звездочка. Регулярные выражения можно определить рекурсивно. В этом определении не только характеризуются правильные регулярные выражения, но и для каждого регулярного выражения E описывается представленный им язык, который обозначается через $L(E)$.

Базис. Базис состоит из трех частей.

1. Константы ε и $\varnothing$ являются регулярными выражениями, определяющими языки $\{\varepsilon\}$ и $\varnothing$, соответственно, т.е. $L(\varepsilon) = \{\varepsilon\}$ и $L(\varnothing) = \varnothing$.
2. Если a — произвольный символ, то **a** — регулярное выражение, определяющее язык $\{a\}$, т.е. $L(\mathbf{a}) = \{a\}$. Заметим, что для записи выражения, соответствующего символу, используется жирный шрифт. Это соответствие, т.е. что **a** относится к a, должно быть очевидным.

3. Переменная, обозначенная прописной курсивной буквой, например, L, представляет произвольный язык.

Индукция. Индуктивный шаг состоит из четырех частей, по одной для трех операторов и для введения скобок.

1. Если E и F — регулярные выражения, то $E + F$ — регулярное выражение, определяющее объединение языков $L(E)$ и $L(F)$, т.е. $L(E + F) = L(E) \cup L(F)$.
2. Если E и F — регулярные выражения, то EF — регулярное выражение, определяющее конкатенацию языков $L(E)$ и $L(F)$. Таким образом, $L(EF) = L(E)L(F)$. Заметим, что для обозначения оператора конкатенации — как операции над языками, так и оператора в регулярном выражении — можно использовать точку. Например, регулярное выражение **0.1** означает то же, что и **01**, и представляет язык {01}. Однако мы избегаем использовать точку в качестве оператора конкатенации в регулярных выражениях[3].
3. Если E — регулярное выражение, то E^* — регулярное выражение, определяющее итерацию языка $L(E)$. Таким образом, $L(E^*) = (L(E))^*$.
4. Если E — регулярное выражение, то (E) — регулярное выражение, определяющее тот же язык $L(E)$, что и выражение E. Формально, $L((E)) = L(E)$.

Выражения и соответствующие языки

Строго говоря, регулярное выражение E — это просто выражение, а не язык. Мы используем $L(E)$ для обозначения языка, который соответствует E. Однако довольно часто говорят "E", на самом деле подразумевая "$L(E)$". Это соглашение используется в случаях, когда ясно, что речь идет о языке, а не о регулярном выражении.

Пример 3.2. Напишем регулярное выражение для множества цепочек из чередующихся нулей и единиц. Сначала построим регулярное выражение для языка, состоящего из одной-единственной цепочки 01. Затем используем оператор "звездочка" для того, чтобы построить выражение для всех цепочек вида 0101...01.

Базисное правило для регулярных выражений говорит, что **0** и **1** — это выражения, обозначающие языки {0} и {1}, соответственно. Если соединить эти два выражения, то получится регулярное выражение **01** для языка {01}. Как правило, если мы хотим написать выражение для языка, состоящего из одной цепочки w, то используем саму w как регулярное выражение. Заметим, что в таком регулярном выражении символы цепочки w обычно выделяют жирным шрифтом, но изменение шрифта предназначено лишь для того, чтобы отличить выражение от цепочки, и не должно восприниматься как что-то существенное.

---

[3] В UNIX точка в регулярных выражениях используется для совершенно другой цели — представления любого знака кода ASCII.

Далее, для получения всех цепочек, состоящих из нуля или нескольких вхождений 01, используем регулярное выражение $(\mathbf{01})^*$. Заметим, что выражение **01** заключается в скобки, чтобы не путать его с выражением $\mathbf{01}^*$. Цепочки языка $\mathbf{01}^*$ начинаются с 0, за которым следует любое количество 1. Причина такой интерпретации объясняется в разделе 3.1.3 и состоит в том, что операция "звездочка" имеет высший приоритет по сравнению с операцией "точка", и поэтому аргумент оператора итерации выбирается до выполнения любых конкатенаций.

Однако $L((\mathbf{01})^*)$ — не совсем тот язык, который нам нужен. Он включает только те цепочки из чередующихся нулей и единиц, которые начинаются с 0 и заканчиваются 1. Мы должны также учесть возможность того, что вначале стоит 1 и/или в конце 0. Одним из решений является построение еще трех регулярных выражений, описывающих три другие возможности. Итак, $(\mathbf{10})^*$ представляет те чередующиеся цепочки, которые начинаются символом 1 и заканчиваются символом 0, $\mathbf{0}(\mathbf{10})^*$ можно использовать для цепочек, которые начинаются и заканчиваются символом 0, а $\mathbf{1}(\mathbf{01})^*$ — для цепочек, которые и начинаются, и заканчиваются символом 1. Полностью это регулярное выражение имеет следующий вид.

$$(\mathbf{01})^* + (\mathbf{10})^* + \mathbf{0}(\mathbf{10})^* + \mathbf{1}(\mathbf{01})^*$$

Заметим, что оператор + используется для объединения тех четырех языков, которые вместе дают все цепочки, состоящие из чередующихся символов 0 и 1.

Однако существует еще одно решение, приводящее к регулярному выражению, которое имеет значительно отличающийся и к тому же более краткий вид. Снова начнем с выражения $(\mathbf{01})^*$. Можем добавить необязательную единицу в начале, если слева к этому выражению допишем выражение $\varepsilon + \mathbf{1}$. Аналогично, добавим необязательный 0 в конце с помощью конкатенации с выражением $\varepsilon + \mathbf{0}$. Например, используя свойства оператора +, получим, что

$$L(\varepsilon + \mathbf{1}) = L(\varepsilon) \bigcup L(\mathbf{1}) = \{\varepsilon\}\{1\} = \{\varepsilon, 1\}.$$

Если мы допишем к этому языку любой другой язык L, то выбор цепочки ε даст нам все цепочки из L, а выбрав 1, получим $1w$ для каждой цепочки w из L. Таким образом, совокупность цепочек из чередующихся нулей и единиц может быть представлена следующим выражением.

$$(\varepsilon + \mathbf{1})(\mathbf{01})^*(\varepsilon + \mathbf{0})$$

Обратите внимание на то, что суммируемые выражения необходимо заключать в скобки, чтобы обеспечить правильную группировку операторов. □

3.1.3. Приоритеты регулярных операторов

Как и в других алгебрах, операторы регулярных выражений имеют определенные "приоритеты", т.е. операторы связываются со своими операндами в определенном порядке. Мы знакомы с понятием приоритетности для обычных арифметических выраже-

ний. Например, мы знаем, что в выражении $xy + z$ умножение xy выполняется перед сложением, так что это выражение эквивалентно выражению со скобками $(xy) + z$, а не $x(y + z)$. Аналогично, в арифметике мы группируем одинаковые операторы слева направо, поэтому $x – y – z$ эквивалентно выражению $(x – y) – z$, а не $x – (y – z)$. Для операторов регулярных выражений определен следующий порядок приоритетов.

1. Оператор "звездочка" имеет самый высокий приоритет, т.е. этот оператор применяется только к наименьшей последовательности символов, находящейся слева от него и являющейся правильно построенным регулярным выражением.
2. Далее по порядку приоритетности следует оператор конкатенации, или "точка". Связав все "звездочки" с их операндами, связываем операторы конкатенации с соответствующими им операндами, т.е. все *смежные* (соседние, без промежуточных операторов) выражения группируются вместе. Поскольку оператор конкатенации является ассоциативным, то не имеет значения, в каком порядке мы группируем последовательные конкатенации. Если же необходимо сделать выбор, то следует группировать их, начиная слева. Например, **012** группируется как (**01**)**2**.
3. В заключение, со своими операндами связываются операторы объединения (операторы +). Поскольку объединение тоже является ассоциативным оператором, то и здесь не имеет большого значения, в каком порядке сгруппированы последовательные объединения, однако мы будем придерживаться группировки, начиная с левого края выражения.

Конечно, иногда нежелательно, чтобы группирование в регулярном выражении определялось только приоритетом операторов. В таких случаях можно расставить скобки и сгруппировать операнды по своему усмотрению. Кроме того, не запрещается заключать в скобки операнды, которые вы хотите сгруппировать, даже если такое группирование подразумевается правилами приоритетности операторов.

Пример 3.3. Выражение $\mathbf{01}^* + \mathbf{1}$ группируется как $(\mathbf{0}(\mathbf{1}^*)) + \mathbf{1}$. Сначала выполняется оператор "звездочка". Поскольку символ **1**, находящийся непосредственно слева от оператора, является допустимым регулярным выражением, то он один будет операндом "звездочки". Далее группируем конкатенацию **0** и $(\mathbf{1})^*$ и получаем выражение $(\mathbf{0}(\mathbf{1}^*))$. Наконец, оператор объединения связывает последнее выражение с выражением, которое находится справа, т.е. с **1**.

Заметим, что язык данного выражения, сгруппированного согласно правилам приоритетности, содержит цепочку 1 плюс все цепочки, начинающиеся с 0, за которым следует любое количество единиц (в том числе и ни одной). Если бы мы захотели сначала сгруппировать точку, а потом звездочку, то следовало бы использовать скобки: $(\mathbf{01})^* + \mathbf{1}$. Язык этого выражения состоит из цепочки 1 и всех цепочек, в которых 01 повторяется нуль или несколько раз. Для того чтобы сначала выполнить объединение, его нужно заключить в скобки: $\mathbf{0}(\mathbf{1}^* + \mathbf{1})$. Язык этого выражения состоит из цепочек, которые начинаются с 0 и продолжаются любым количеством единиц. □

3.1.4. Упражнения к разделу 3.1

3.1.1. Напишите регулярные выражения для следующих языков:

а) (*) множество цепочек с алфавитом $\{a, b, c\}$, содержащих хотя бы один символ a и хотя бы один символ b;

б) множество цепочек из нулей и единиц, в которых десятый от правого края символ равен 1;

в) множество цепочек из нулей и единиц, содержащих не более одной пары последовательных единиц.

3.1.2. (!) Напишите регулярные выражения для следующих языков:

а) (*) множество всех цепочек из нулей и единиц, в которых каждая пара смежных нулей находится перед парой смежных единиц;

б) множество цепочек, состоящих из нулей и единиц, в которых число нулей кратно пяти.

3.1.3. (!!) Напишите регулярные выражения для следующих языков:

а) множество всех цепочек из нулей и единиц, в которых нет подцепочки 101;

б) множество всех цепочек, в которых поровну нулей и единиц и ни один их префикс не содержит нулей на два больше, чем единиц, или единиц на две больше, чем нулей;

в) множество всех цепочек из нулей и единиц, в которых число нулей делится на пять, а количество единиц четно.

3.1.4. (!) Опишите обычными словами языки следующих регулярных выражений:

а) (*) $(\mathbf{1} + \varepsilon)(\mathbf{00}^*\mathbf{1})^*\mathbf{0}^*$;

б) $(\mathbf{0}^*\mathbf{1}^*)^*\mathbf{000}(\mathbf{0} + \mathbf{1})^*$;

в) $(\mathbf{0} + \mathbf{10})^*\mathbf{1}^*$.

3.1.5. (*!) В примере 3.1 отмечено, что $\varnothing$ — это один из двух языков, итерация которых является конечным множеством. Укажите второй язык.

3.2. Конечные автоматы и регулярные выражения

Хотя описание языков с помощью регулярных выражений принципиально отличается от конечноавтоматного, оказывается, что обе эти нотации представляют одно и то же множество языков, называемых “регулярными”. Выше мы показали, что детерминированные конечные автоматы, а также два вида недетерминированных конечных автоматов — с ε-переходами и без ε-переходов — допускают один и тот же класс языков. Для того чтобы показать, что регулярные выражения задают тот же класс языков, необходимо доказать следующее.

1. Любой язык, задаваемый одним из этих автоматов, может быть также определен регулярным выражением. Для доказательства можно предположить, что язык допускается некоторым ДКА.
2. Любой язык, определяемый регулярным выражением, может быть также задан с помощью одного из вышеуказанных автоматов. Для этой части доказательства проще всего показать, что существует НКА с ε-переходами, допускающий тот же самый язык.

На рис. 3.1 показаны все эквивалентности, которые уже доказаны или будут доказаны. Дуга, ведущая от класса *X* к классу *Y*, означает, что каждый язык, определяемый классом *X*, определяется также классом *Y*. Поскольку данный граф является сильно связным (в нем можно перейти от каждой из четырех вершин к любой другой вершине), понятно, что все четыре класса на самом деле эквивалентны.

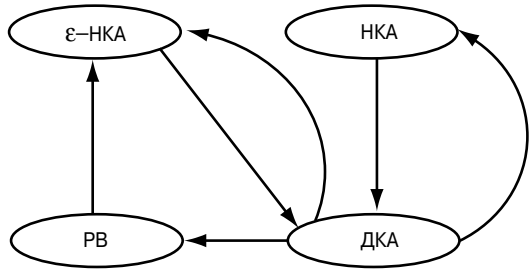


Рис. 3.1. План доказательства эквивалентности четырех различных нотаций для регулярных языков

3.2.1. От ДКА к регулярным выражениям

Построение регулярного выражения для языка, допускаемого некоторым ДКА, оказывается на удивление сложным. Приблизительно это выглядит так: мы строим выражения, описывающие множества цепочек, которыми помечены определенные пути на диаграмме ДКА. Однако эти пути могут проходить только через ограниченное подмножество состояний. При индуктивном определении таких выражений мы начинаем с самых простых выражений, описывающих пути, которые не проходят ни через одно состояние (т.е. являются отдельными вершинами или дугами). Затем индуктивно строим выражения, которые позволяют этим путям проходить через постепенно расширяющиеся множества состояний. В конце этой процедуры получим пути, которые могут проходить через любые состояния, т.е. сгенерируем выражения, представляющие все возможные пути. Эти идеи используются в доказательстве следующей теоремы.

Теорема 3.4. Если $L = L(A)$ для некоторого ДКА A, то существует регулярное выражение R, причем $L = L(R)$.

Доказательство. Предположим, что $\{1, 2, \ldots, n\}$ — множество состояний автомата A для некоторого натурального n. Независимо от того, какими эти состояния являются на

самом деле, их конечное число n, поэтому их можно переименовать, используя первые n положительных целых чисел. Нашей первой и наиболее сложной задачей является построение набора регулярных выражений, которые описывают постепенно расширяющиеся множества путей в диаграмме переходов автомата A.

Обозначим через $R_{ij}^{(k)}$ регулярное выражение, язык которого состоит из множества меток w путей, ведущих от состояния i к состоянию j автомата A и не имеющих промежуточных состояний с номерами больше k. Заметим, что начальная и конечная точки пути не являются "промежуточными", поэтому мы не требуем, чтобы i и/или j были меньше или равны k.

Условия, налагаемые на пути выражениями $R_{ij}^{(k)}$, представлены на рис. 3.2. Здесь на вертикальной оси расположены состояния, начиная с 1 внизу до n вверху, а горизонтальная ось представляет движение вдоль пути. Заметим, что на этой диаграмме показан случай, когда i и j больше, чем k, но любое из этих чисел, или оба, могут быть меньше или равны k. Также обратите внимание на то, что путь дважды проходит через вершину k, но только в крайних точках поднимается выше, чем k.

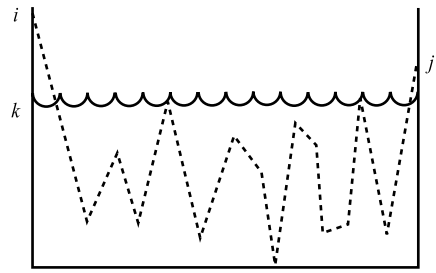


Рис. 3.2. Путь, отметка которого принадлежит языку регулярного выражения $R_{ij}^{(k)}$

Для построения выражения $R_{ij}^{(k)}$ используют следующее индуктивное определение, которое начинается с $k = 0$ и достигает $k = n$. Заметим, что при $k = n$ пути ничем не ограничиваются, поскольку нет состояний с номерами, которые больше, чем n.

Базис. В качестве базиса примем $k = 0$. Поскольку все состояния пронумерованы от 1 и далее, то это условие означает, что у пути вообще нет промежуточных состояний. Существует только два вида путей, удовлетворяющих такому условию.

1. Дуга, ведущая от вершины (состояния) i к вершине j.
2. Путь длины 0, состоящий лишь из некоторой вершины i.

Если $i \neq j$, то возможен только первый случай. Необходимо проанализировать данный ДКА A и найти такие входные символы a, по которым есть переход из состояния i в состояние j:

а) если таких символов нет, то $R_{ij}^{(0)} = \varnothing$;

б) если существует только один такой символ a, то $R_{ij}^{(0)} = \mathbf{a}$;

в) если существуют такие символы a_1, a_2, …, a_k, которыми помечены дуги из состояния i в состояние j, то $R_{ij}^{(0)} = \mathbf{a}_1 + \mathbf{a}_2 + \ldots + \mathbf{a}_k$.

В то же время, если $i = j$, то допустимыми путями являются путь длины 0 и все петли, которые начинаются и заканчиваются в состоянии i. Путь длины 0 может быть представлен регулярным выражением ε, потому что вдоль этого пути нет символов. Следовательно, добавляем ε к выражениям, полученным выше в пунктах (а)–(в). Таким образом, в случае (а) [нет ни одного символа a] получим выражение ε, в случае (б) [один символ a] выражение примет вид $\varepsilon + \mathbf{a}$, и в случае (в) [несколько символов] получим выражение $\varepsilon + \mathbf{a}_1 + \mathbf{a}_2 + \ldots + \mathbf{a}_k$.

Индукция. Предположим, что существует путь из состояния i в состояние j, не проходящий через состояния с номерами, которые больше, чем k. Необходимо рассмотреть две ситуации.

1. Путь вообще не проходит через состояние k. В этом случае метка пути принадлежит языку $R_{ij}^{(k-1)}$.

2. Путь проходит через состояние k по меньшей мере один раз. Тогда мы можем разделить путь на несколько частей, как показано на рис. 3.3. Первая часть ведет от состояния i к состоянию k, но не проходит через k, последняя ведет из k в j, также не проходя через k, а все части, расположенные внутри пути, ведут из k в k, не проходя через k. Заметим, что если путь проходит через состояние k только один раз, то "внутренних" частей нет, а есть только путь из i в k и путь из k в j. Множество меток для всех путей такого вида может быть представлено регулярным выражением $R_{ik}^{(k-1)} \left(R_{kk}^{(k-1)} \right)^* R_{kj}^{(k-1)}$. Таким образом, первое выражение представляет часть пути, ведущую в состояние k в первый раз, второе — часть, ведущую из k в k нуль или несколько раз, и третье выражение — часть пути, которая выходит из состояния k в последний раз и ведет в состояние j.

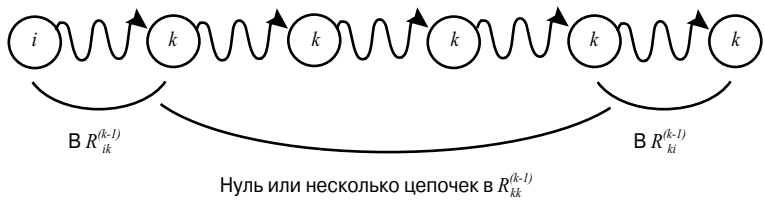


Рис. 3.3. Путь из i в j может быть разбит на несколько сегментов в каждой точке, в которой он проходит через состояние k

После объединения выражений для путей двух рассмотренных выше типов получим следующее выражение для меток всех путей, ведущих из состояния i в состояние j, которые не проходят через состояния с номерами, которые больше, чем k.

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)} \left(R_{kk}^{(k-1)} \right)^* R_{kj}^{(k-1)}$$

Поскольку данные выражения строятся в порядке возрастания верхнего индекса, можно построить любое выражение $R_{ij}^{(k)}$, так как оно зависит только от выражений с меньшими значениями верхнего индекса.

В итоге получим $R_{ij}^{(n)}$ для всех i и j. Можно предположить, что состояние 1 является начальным, а множество допускающих (заключительных) состояний может быть любым. Тогда регулярным выражением для языка, допускаемого данным автоматом, будет сумма (объединение) всех тех выражений $R_{1j}^{(n)}$, в которых состояние j является допускающим. □

Пример 3.5. Преобразуем ДКА, представленный на рис. 3.4, в соответствующее регулярное выражение. Этот ДКА допускает все цепочки, содержащие хотя бы один 0. Чтобы понять, почему это так, заметим, что автомат переходит из начального состояния 1 в заключительное состояние 2, как только на входе появляется 0. Далее автомат остается в заключительном состоянии 2 при любой входной последовательности.

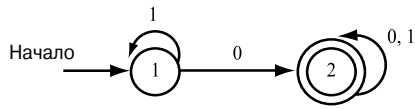


Рис. 3.4. ДКА, допускающий все цепочки, которые содержат хотя бы один 0

Ниже приведены базисные выражения для построений согласно теореме 3.4.

$R_{11}^{(0)}$	$\varepsilon + \mathbf{1}$
$R_{12}^{(0)}$	$\mathbf{0}$
$R_{21}^{(0)}$	$\varnothing$
$R_{22}^{(0)}$	$(\varepsilon + \mathbf{0} + \mathbf{1})$

Например, в выражении $R_{11}^{(0)}$ присутствует член ε, потому что и начальным, и конечным является состояние 1. Это выражение включает также **1**, поскольку существует путь из состояния 1 в состояние 1 по входу 1. Выражение $R_{12}^{(0)}$ равно **0**, потому что есть дуга с меткой 0, ведущая из состояния 1 в состояние 2. Здесь нет члена ε, поскольку начальное и конечное состояния различаются. И, наконец, $R_{21}^{(0)} = \varnothing$, так как нет путей, ведущих из состояния 2 в состояние 1.

Теперь применим индукцию для построения более сложных выражений. Вначале они соответствуют путям, проходящим через состояние 1, а затем путям, которые могут про-

ходить через состояния 1 и 2, т.е. любым путям. Правило для вычисления выражения $R_{ij}^{(1)}$ представляет собой пример общего правила из индуктивной части теоремы 3.4.

$$R_{ij}^{(1)} = R_{ij}^{(0)} + R_{i1}^{(0)} (R_{11}^{(0)})^* R_{1j}^{(0)} \tag{3.1}$$

В таблице на рис. 3.5 сначала представлены выражения, полученные с помощью прямой подстановки в приведенную выше формулу, а затем упрощенные выражения, которые определяют те же языки, что и более сложные выражения.

	Прямая подстановка	Упрощенное выражение
$R_{11}^{(1)}$	$\varepsilon + \mathbf{1} + (\varepsilon + \mathbf{1})(\varepsilon + \mathbf{1})^*(\varepsilon + \mathbf{1})$	$\mathbf{1}^*$
$R_{12}^{(1)}$	$\mathbf{0} + (\varepsilon + \mathbf{1})(\varepsilon + \mathbf{1})^*\mathbf{0}$	$\mathbf{1}^*\mathbf{0}$
$R_{21}^{(1)}$	$\varnothing + \varnothing(\varepsilon + \mathbf{1})^*(\varepsilon + \mathbf{1})$	$\varnothing$
$R_{22}^{(1)}$	$\varepsilon + \mathbf{0} + \mathbf{1} + \varnothing(\varepsilon + \mathbf{1})^*\mathbf{0}$	$\varepsilon + \mathbf{0} + \mathbf{1}$

Рис. 3.5. Регулярные выражения для путей, которые могут проходить только через состояние 1

Например, рассмотрим выражение $R_{12}^{(1)}$. Подставив $i = 1$ и $j = 2$ в (3.1), получим $R_{12}^{(0)} + R_{11}^{(0)} (R_{11}^{(0)})^* R_{12}^{(0)}$.

Общим принципом упрощения является следующий: если R — произвольное регулярное выражение, то $(\varepsilon + R)^* = R^*$. Он основан на том, что обе части этого равенства описывают язык, образованный конкатенациями нуля или нескольких цепочек из $L(R)$. В нашем случае $(\varepsilon + \mathbf{1})^* = \mathbf{1}^*$; отметим, что оба выражения описывают цепочки, состоящие из любого количества единиц. Далее, $(\varepsilon + \mathbf{1})\mathbf{1}^* = \mathbf{1}^*$. Опять-таки, легко заметить, что оба выражения означают “любое количество единиц”. Следовательно, исходное выражение $R_{12}^{(1)}$ эквивалентно выражению $\mathbf{0} + \mathbf{1}^*\mathbf{0}$. Последнее описывает язык, содержащий цепочку 0 и все цепочки, заканчивающиеся символом 0, перед которым стоит произвольное количество единиц. Такой язык также можно определить более простым выражением $\mathbf{1}^*\mathbf{0}$.

Выражение $R_{11}^{(1)}$ упрощается аналогично рассмотренному выше выражению $R_{12}^{(1)}$. Упрощение $R_{11}^{(1)}$ и $R_{12}^{(1)}$ зависит от двух следующих правил, описывающих операции с $\varnothing$ и выполнимых для любого регулярного выражения R.

1. $\varnothing R = R\varnothing = \varnothing$, т.е. $\varnothing$ является нулем (*аннулятором*) для конкатенации. В результате конкатенации $\varnothing$, слева или справа, с любым другим выражением получается $\varnothing$. Это правило очевидно, поскольку для того, чтобы в результате конкатенации получить некоторую цепочку, мы должны взять цепочки из обоих аргументов конкатенации. Если один из аргументов равен $\varnothing$, выбор цепочки из него становится невозможным.
2. $\varnothing + R = R + \varnothing = R$, т.е. $\varnothing$ является единицей для операции объединения. В результате объединения любого выражения с $\varnothing$ получим то же самое выражение.

Итак, выражение $\varnothing(\varepsilon + \mathbf{1})^*(\varepsilon + \mathbf{1})$ можно заменить $\varnothing$. После сказанного должны быть понятны и два последних упрощения.

Теперь вычислим выражения $R_{ij}^{(2)}$. Индуктивное правило для $k = 2$ имеет следующий вид.

$$R_{ij}^{(2)} = R_{ij}^{(1)} + R_{i1}^{(1)} (R_{11}^{(1)})^* R_{1j}^{(1)} \tag{3.2}$$

Если подставим упрощенные выражения из таблицы на рис. 3.5 в уравнение (3.2), то получим выражения, представленные на рис. 3.6. На этом рисунке также приведены упрощенные выражения, полученные согласно правилам, описанным для рис. 3.5.

	Прямая подстановка	Упрощенное выражение
$R_{11}^{(2)}$	$\mathbf{1}^* + \mathbf{1}^*\mathbf{0}(\varepsilon + \mathbf{0} + \mathbf{1})^*\varnothing$	$\mathbf{1}^*$
$R_{12}^{(2)}$	$\mathbf{1}^*\mathbf{0} + \mathbf{1}^*\mathbf{0}(\varepsilon + \mathbf{0} + \mathbf{1})^*(\varepsilon + \mathbf{0} + \mathbf{1})$	$\mathbf{1}^*\mathbf{0}(\mathbf{0} + \mathbf{1})^*$
$R_{21}^{(2)}$	$\varnothing + (\varepsilon + \mathbf{0} + \mathbf{1})(\varepsilon + \mathbf{0} + \mathbf{1})^*\varnothing$	$\varnothing$
$R_{22}^{(2)}$	$\varepsilon + \mathbf{0} + \mathbf{1} + (\varepsilon + \mathbf{0} + \mathbf{1})(\varepsilon + \mathbf{0} + \mathbf{1})^*(\varepsilon + \mathbf{0} + \mathbf{1})$	$(\mathbf{0} + \mathbf{1})^*$

Рис. 3.6. Регулярные выражения для путей, которые могут проходить через любое состояние

Окончательное регулярное выражение, эквивалентное автомату, представленному на рис. 3.4, строится путем объединения всех тех выражений, для которых первое состояние является начальным, а второе — заключительным. В нашем примере 1 — начальное состояние, а 2 — заключительное, поэтому нам нужно лишь выражение $R_{12}^{(2)}$, равное $\mathbf{1}^*\mathbf{0}(\mathbf{0} + \mathbf{1})^*$. Оно очень просто интерпретируется. Его язык состоит из всех цепочек, начинающихся с нулевого или некоторого количества единиц, за которыми следует 0, а за ним — любая цепочка из нулей и единиц. Иначе говоря, это все цепочки из 0 и 1, содержащие хотя бы один 0. □

3.2.2. Преобразование ДКА в регулярное выражение методом исключения состояний

Метод преобразования ДКА в регулярное выражение, представленный в разделе 3.2.1, работает всегда. Как вы, возможно, заметили, он на самом деле не зависит от того, детерминирован ли этот автомат, и точно так же применим и к НКА, и даже к ε-НКА. Однако такой метод построения регулярного выражения очень трудоемок. Не только потому, что для автомата с n состояниями необходимо построить порядка n^3 выражений, но и потому, что с каждым из n шагов индукции длина выражения может возрастать в среднем в четыре раза, если эти выражения не упрощать. Таким образом, размеры результирующих выражений могут достигать порядка 4^n символов.

Существует аналогичный метод, избавляющий от некоторых повторных действий. Например, во всех выражениях с верхним индексом (k) в доказательстве теоремы 3.4 ис-

пользуется одно и то же подвыражение $(R_{kk}^{(k-1)})^*$, которое приходится выписывать в общей сложности n^2 раз.

Метод построения регулярных выражений, который мы изучим в этом разделе, предполагает исключение состояний. Если исключить некоторое состояние s, то все пути автомата, проходящие через это состояние, исчезают. Для того чтобы язык автомата при этом не изменился, необходимо написать на дуге, ведущей непосредственно из некоторого состояния q в состояние p, метки всех тех путей, которые вели из состояния q в состояние p, проходя через состояние s. Поскольку теперь метка такой дуги будет содержать цепочки, а не отдельные символы, и таких цепочек может быть даже бесконечно много, то мы не можем просто записать список этих цепочек в качестве метки. К счастью, существует простой и конечный способ для представления всех подобных цепочек, а именно, использование регулярных выражений.

Таким образом, мы пришли к рассмотрению автоматов, у которых метками являются регулярные выражения. Язык такого автомата представляет собой объединение по всем путям, ведущим от начального к заключительному состоянию, языков, образуемых с помощью конкатенации языков регулярных выражений, расположенных вдоль этих путей. Обратите внимание на то, что это правило согласуется с определением языка для любого рассмотренного выше типа автоматов. Каждый символ a или ε, если он разрешен, можно рассматривать как регулярное выражение, языком которого является единственная цепочка, т.е. $\{a\}$ или $\{\varepsilon\}$. Можно принять это замечание за основу описываемой ниже процедуры исключения состояний.

На рис. 3.7 показано состояние s, которое мы собираемся исключить. Предположим, что автомат, содержащий s, содержит состояния q_1, q_2, …, q_k, предшествующие s, и p_1, p_2, …, p_m, следующие за s. Возможно, некоторые из состояний q совпадают с состояниями p, но мы предполагаем, что среди q и p нет s, даже если существует петля, которая начинается и заканчивается в s, как показано на рис. 3.7. Над каждой дугой, ведущей к состоянию s, указано регулярное выражение; дуга, ведущая из состояния q_i, помечена выражением Q_i. Аналогично, регулярным выражением P_i помечена дуга, ведущая из s в состояние p_i, для каждого i. Петля на s имеет метку S. Наконец, регулярным выражением R_{ij} помечена каждая дуга, ведущая из q_i в p_j для всех i и j. Заметим, что некоторых из этих дуг в автомате может не быть. В этом случае в качестве выражения над дугой используется символ $\varnothing$.

На рис. 3.8 показано, что получится, если исключить состояние s. Все дуги, проходящие через s, удалены. Чтобы это компенсировать, для каждого состояния q_i, предшествующего s, и для каждого p_j, следующего за s, вводится регулярное выражение, представляющее все пути, которые начинаются в q_i, идут к s, возможно, проходят петлю на s нуль или несколько раз, и наконец ведут в p_j. Выражение для таких путей равно $Q_i S^* P_j$. Это выражение добавляется (с помощью оператора объединения) к выражению над дугой из q_i в p_j. Если дуга $q_i \to p_j$ отсутствует, то вначале ей соответствует регулярное выражение $\varnothing$.

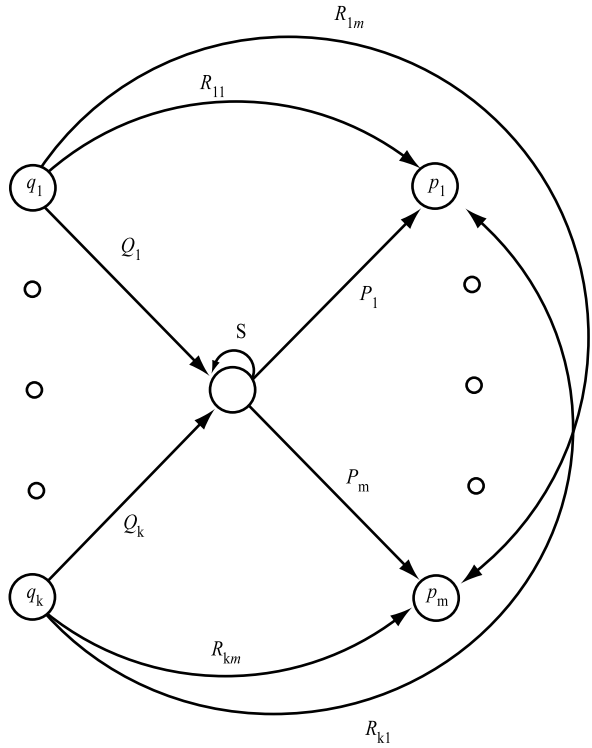


Рис. 3.7. Состояние s, подлежащее исключению

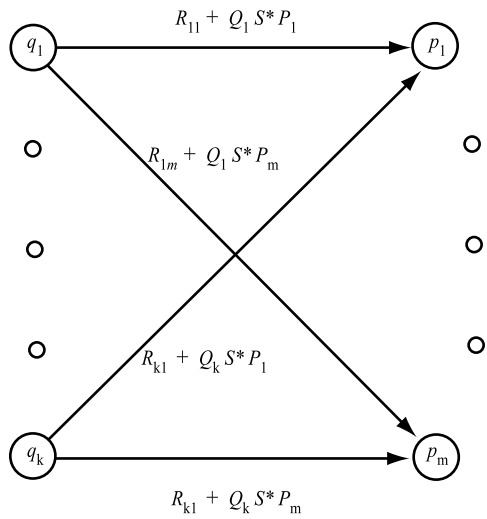


Рис. 3.8. Результат исключения состояния s из автомата, изображенного на рис. 3.7

Стратегия построения регулярного выражения по конечному автомату такова.

1. Для каждого допускающего состояния q применить описанный выше процесс сокращения, чтобы построить эквивалентный автомат с дугами, помеченными регулярными выражениями. Исключить все состояния, кроме q и начального состояния q_0.

2. Если $q \neq q_0$, то должен остаться автомат с двумя состояниями, подобный автомату на рис. 3.9. Регулярное выражение для допустимых цепочек может быть записано поразному. Один из видов — $(R + SU^*T)^*SU^*$. Поясним его. Можно переходить из начального состояния в него же любое количество раз, проходя путями, метки которых принадлежат либо $L(R)$, либо $L(SU^*T)$. Выражение SU^*T представляет пути, которые ведут в допускающее состояние по пути с меткой из языка $L(S)$, затем, возможно, несколько раз проходят через допускающее состояние, используя пути с метками из $L(U)$, и наконец возвращаются в начальное состояние, следуя по пути с меткой из $L(T)$. Отсюда нужно перейти в допускающее состояние, уже никогда не возвращаясь в начальное, вдоль пути с меткой из $L(S)$. Находясь в допускающем состоянии, можно сколько угодно раз вернуться в него по пути с меткой из $L(U)$.

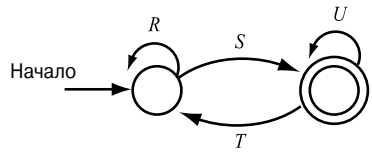


Рис. 3.9. Обобщенный автомат с двумя состояниями

3. Если же начальное состояние является допускающим, то необходимо точно так же исключить состояния исходного автомата, удалив все состояния, кроме начального. В результате получим автомат с одним состоянием, подобный представленному на рис. 3.10. Регулярное выражение R^* задает цепочки, допускаемые этим автоматом.

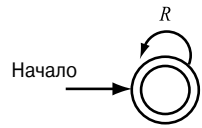


Рис. 3.10. Обобщенный автомат с одним состоянием

4. Искомое выражение представляет собой сумму (объединение) всех выражений, полученных с помощью сокращенного автомата для каждого допускающего состояния согласно правилам 2 и 3.

Пример 3.6. Рассмотрим НКА, представленный на рис. 3.11, допускающий цепочки из нулей и единиц, у которых либо на второй, либо на третьей позиции с конца стоит 1. Вначале преобразуем этот автомат в автомат с регулярными выражениями в качестве меток.

Пока исключение состояний не производилось, то все, что нам нужно сделать, это заменить метки “0, 1” эквивалентным регулярным выражением **0** + **1**. Результат показан на рис. 3.12.

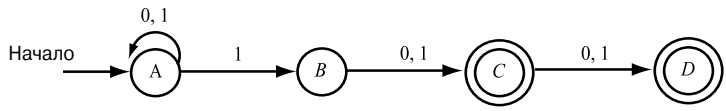


Рис. 3.11. НКА, допускающий цепочки, у которых 1 находится либо на второй, либо на третьей позиции с конца цепочки

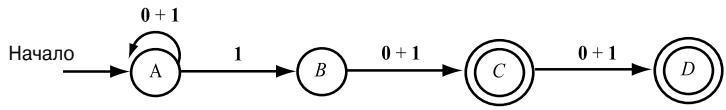


Рис. 3.12. Автомат, изображенный на рис. 3.11, с регулярными выражениями в качестве меток

Исключим сначала состояние *B*. Поскольку это состояние не является ни начальным, ни допускающим, то его не будет ни в одном из сокращенных автоматов. Мы избавимся от лишней работы, если исключим это состояние до того, как начнем строить два сокращенных автомата, соответствующих двум его допускающим состояниям.

Существует одно состояние *A*, предшествующее *B*, и одно последующее состояние *C*. Используя обозначения регулярных выражений диаграммы, представленной на рис. 3.7, получим: $Q_1 = \mathbf{1}$, $P_1 = \mathbf{0} + \mathbf{1}$, $R_{11} = \varnothing$ (потому что из *A* в *C* дуги нет) и $S = \varnothing$ (поскольку нет петли в состоянии *B*). В результате, выражение над новой дугой из *A* в *C* равно $\varnothing + \mathbf{1}\varnothing^*(\mathbf{0} + \mathbf{1})$.

Для сокращения полученного выражения сначала исключаем первый элемент $\varnothing$, который можно игнорировать при объединении. Выражение приобретает вид $\mathbf{1}\varnothing^*(\mathbf{0} + \mathbf{1})$. Напомним, что регулярное выражение $\varnothing^*$ эквивалентно регулярному выражению ε, поскольку

$$L(\varnothing^*) = \{\varepsilon\} \cup L(\varnothing) \cup L(\varnothing)L(\varnothing) \cup \ldots$$

Все члены этого объединения, кроме первого, пусты, поэтому $L(\varnothing^*) = \{\varepsilon\}$, что совпадает с $L(\varepsilon)$. Следовательно, $\mathbf{1}\varnothing^*(\mathbf{0} + \mathbf{1})$ эквивалентно выражению $\mathbf{1}(\mathbf{0} + \mathbf{1})$, которое использовано для дуги $A \rightarrow C$ на рис. 3.13.

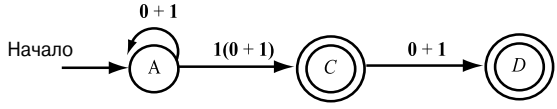


Рис. 3.13. Исключение состояния B

Далее нужно по отдельности исключить состояния *C* и *D*. Процедура исключения состояния *C* аналогична описанному выше исключению состояния *B*, в результате чего получится автомат, представленный на рис. 3.14.

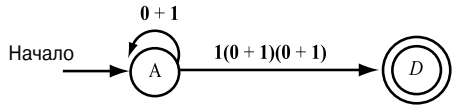


Рис. 3.14. Автомат с двумя состояниями A и D

В обозначениях обобщенного автомата с двумя состояниями, изображенного на рис. 3.9, соответствующие регулярные выражения для рис. 3.14 равны: $R = \mathbf{0} + \mathbf{1}$, $S = \mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$, $T = \varnothing$ и $U = \varnothing$. Выражение U^* можно заменить на ε, т.е. исключить его из конкатенации, поскольку, как показано выше, $\varnothing^* = \varepsilon$. Кроме того, выражение SU^*T эквивалентно $\varnothing$, потому что T — один из операндов конкатенации — равен $\varnothing$. Таким образом, общее выражение $(R + SU^*T)^*SU^*$ в данном случае упрощается до R^*S, или $(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$. Говоря неформально, язык этого выражения состоит из цепочек, заканчивающихся единицей с двумя последующими символами, каждый из которых может быть либо нулем, либо единицей. Этот язык представляет одну часть цепочек, которые допускаются автоматом, изображенным на рис. 3.11: у них на третьей позиции с конца стоит 1.

Теперь снова нужно вернуться к рис. 3.13 и исключить состояние *D*. Поскольку в этом автомате нет состояний, следующих за *D*, то согласно рис. 3.7 необходимо лишь удалить дугу, ведущую из *C* в *D*, вместе с состоянием *D*. В результате получится автомат, показанный на рис. 3.15.

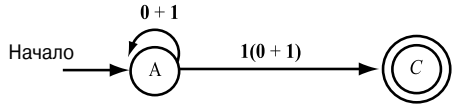


Рис. 3.15. Автомат с двумя состояниями, полученный в результате исключения состояния D

Порядок исключения состояний

Как было отмечено в примере 3.6, если состояние не является ни начальным, ни допускающим, то оно исключается во всех сокращенных автоматах. Таким образом, одно из преимуществ процесса исключения состояний по сравнению с механической генерацией регулярных выражений, описанной в разделе 3.2.1, состоит в том, что сначала мы раз и навсегда исключаем все состояния, которые не являются ни начальными, ни допускающими. Мы вынуждены повторять процедуру сокращения, только когда необходимо исключить несколько допускающих состояний.

Но даже и в этом случае можно совместить некоторые действия процедуры сокращения. Например, если автомат содержит три допускающих состояния *p*, *q* и *r*, то можно вначале исключить *p*, а затем отдельно исключить либо *q*, либо *r*, получив автоматы для допускающих состояний *r* и *q*, соответственно. Затем можно снова начать с трех допускающих состояний и, исключив состояния *q* и *r*, получить автомат для *p*.

Этот автомат очень похож на автомат, изображенный на рис. 3.14; различаются только метки над дугами, ведущими из начального состояния в допускающее. Следовательно, можно применить правило для автомата с двумя состояниями и упростить данное выражение, получив в результате $(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})$. Это выражение представляет другой тип цепочек, допустимых этим автоматом, — цепочки, у которых 1 стоит на второй позиции с конца.

Осталось объединить оба построенные выражения, чтобы получить следующее выражение для всего автомата (см. рис. 3.11).

$$(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1}) + (\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$$

□

3.2.3. Преобразование регулярного выражения в автомат

Теперь завершим план, представленный на рис. 3.1, показав, что любой язык L, являющийся языком $L(R)$ для некоторого регулярного выражения R, будет также языком $L(E)$ для некоторого ε-НКА E. Это доказательство проведем методом структурной индукции по выражению R. Сначала покажем, как строить автоматы для базовых выражений: отдельных символов, ε и $\emptyset$. Затем опишем, каким образом объединять эти автоматы в бо́льшие автоматы, которые допускают объединение, конкатенацию или итерацию языков, допускаемых меньшими автоматами.

Все конструируемые автоматы представляют собой ε-НКА с одним допускающим состоянием.

Теорема 3.7. Любой язык, определяемый регулярным выражением, можно задать некоторым конечным автоматом.

Доказательство. Предположим, что $L = L(R)$ для регулярного выражения R. Покажем, что $L = L(E)$ для некоторого ε-НКА E, обладающего следующими свойствами.

1. Он имеет ровно одно допускающее состояние.
2. У него нет дуг, ведущих в начальное состояние.
3. У него нет дуг, выходящих из допускающего состояния.

Доказательство проводится структурной индукцией по выражению R, следуя рекурсивному определению регулярных выражений из раздела 3.1.2.

Базис. Базис состоит из трех частей, представленных на рис. 3.16. В части (*а*) рассматривается выражение ε. Языком такого автомата является $\{\varepsilon\}$, поскольку единственный путь из начального в допускающее состояние помечен выражением ε. В части (*б*) показана конструкция для $\emptyset$. Понятно, что, поскольку отсутствуют пути из начального состояния в допускающее, языком этого автомата будет $\emptyset$. Наконец, в части (*в*) представлен автомат для регулярного выражения **a**. Очевидно, что язык этого автомата состоит из одной цепочки a и равен $L(\mathbf{a})$. Кроме того, все эти автоматы удовлетворяют условиям (1), (2) и (3) индуктивной гипотезы.

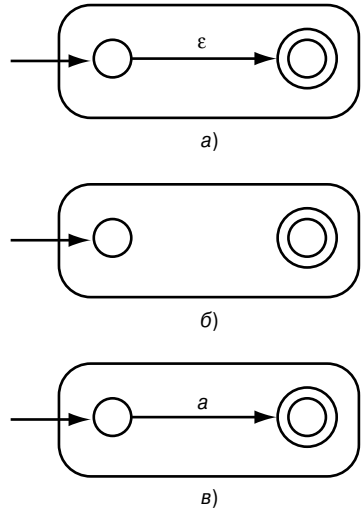


Рис. 3.16. Базис построения автомата по регулярному выражению

Индукция. Три части индукции представлены на рис. 3.17. Предположим, что утверждение теоремы истинно для непосредственных подвыражений данного регулярного выражения, т.е. языки этих подвыражений являются также языками ε-НКА с единственным допускающим состоянием. Возможны четыре случая.

1. Данное выражение имеет вид $R + S$ для некоторых подвыражений R и S. Тогда ему соответствует автомат, представленный на рис. 3.17, *а*. В этом автомате из нового начального состояния можно перейти в начальное состояние автомата для выражения либо R, либо S. Затем мы попадаем в допускающее состояние одного из этих автоматов, следуя по пути, помеченному некоторой цепочкой из языка $L(R)$ или $L(S)$, соответственно. Попав в допускающее состояние автомата для R или S, можно по одному из ε-путей перейти в допускающее состояние нового автомата. Следовательно, язык автомата, представленного на рис. 3.17, *а*, равен $L(R) \bigcup L(S)$.
2. Выражение имеет вид RS для некоторых подвыражений R и S. Автомат для этой конкатенации представлен на рис. 3.17, *б*. Отметим, что начальное состояние первого автомата становится начальным состоянием для всего автомата, представляющего конкатенацию, а допускающим для него будет допускающее состояние второго автомата. Идея состоит в том, что путь, ведущий из начального в допускающее состояние, сначала проходит через автомат для R по некоторому пути, помеченному цепочкой из $L(R)$, а потом — через автомат для S по пути, помеченному цепочкой из $L(S)$. Следовательно, путями автомата, представленного на рис. 3.17, *б*, будут те и только те пути, которые помечены цепочками из языка $L(R)L(S)$.
3. Выражение имеет вид R^* для некоторого подвыражения R. Используем автомат, представленный на рис. 3.17, *в*. Этот автомат позволяет пройти по следующим путям:

а) из начального состояния непосредственно в допускающее по пути, помеченному ε. Этот путь позволяет допустить цепочку ε, которая принадлежит $L(R^*)$ независимо от выражения R;

б) перейти в начальное состояние автомата для R, пройти через этот автомат один или несколько раз, и затем попасть в допускающее состояние. Это множество путей дает возможность допускать цепочки, которые принадлежат языкам $L(R)$, $L(R)L(R)$, $L(R)L(R)L(R)$ и так далее, порождая таким образом все цепочки из $L(R^*)$, за исключением, возможно, цепочки ε. Но она получена в п. 3, *а* как отметка дуги непосредственно из начального в допускающее состояние.

4. Выражение имеет вид (R) для некоторого подвыражения R. Автомат для R может быть автоматом и для (R), поскольку скобки не влияют на язык, задаваемый выражением.

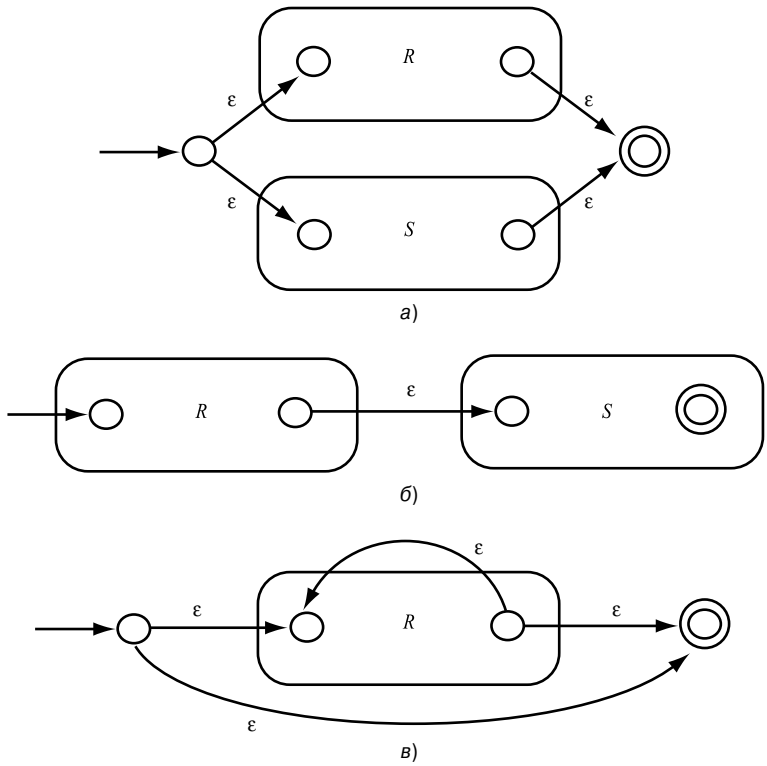


Рис. 3.17. Индуктивный шаг преобразования регулярного выражения в ε-НКА

Легко заметить, что построенные автоматы удовлетворяют всем трем условиям индуктивной гипотезы: одно допускающее состояние, отсутствие дуг, ведущих в начальное состояние, и дуг, выходящих из допускающего состояния. □

Пример 3.8. Преобразуем регулярное выражение $(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})$ в ε-НКА. Построим сначала автомат для $\mathbf{0} + \mathbf{1}$. Для этого используем два автомата, построенные согласно рис. 3.16, *в*: один с меткой **0** на дуге, другой — с меткой **1**. Эти автоматы соединены с помощью конструкции объединения (см. рис. 3.17, *а*). Результат изображен на рис. 3.18, *а*.

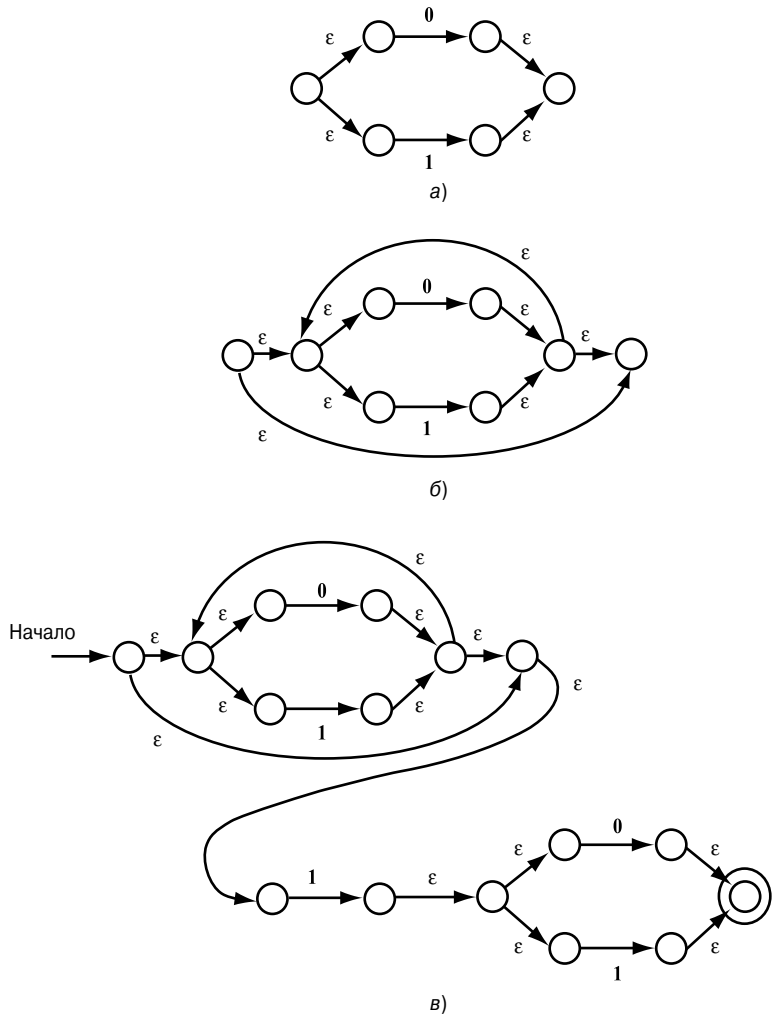


Рис. 3.18. Автомат, построенный для примера 3.8

Далее, применим к автомату (см. рис. 3.18, *а*) конструкцию итерации (см. рис. 3.17, *в*). Полученный автомат изображен на рис. 3.18, *б*. На последних двух шагах применяется конструкция конкатенации (см. рис. 3.17, *б*). Сначала автомат, представленный на рис. 3.18, *б*, соединяется с автоматом, допускающим только цепочку 1. Последний получается путем еще одного применения базисной конструк-

ции (см. рис. 3.16, *в*) с меткой **1** на дуге. Отметим, что для распознавания цепочки 1 необходимо создать *новый* автомат; здесь нельзя использовать автомат для 1, являющийся частью автомата, изображенного на рис. 3.18, *а*. Третьим автоматом в конкатенации будет еще один автомат для выражения **0** + **1**. Опять-таки, необходимо создать копию автомата (см. рис. 3.18, *а*), поскольку нельзя использовать автомат для **0** + **1**, представляющий собой часть автомата (см. рис. 3.18, *б*).

Полный автомат показан на рис. 3.18, *в*. Заметим, что если удалить ε-переходы, то этот ε-НКА будет весьма похож на более простой автомат (см. рис. 3.15), также допускающий цепочки с 1 на предпоследней позиции. □

3.2.4. Упражнения к разделу 3.2

3.2.1. ДКА представлен следующей таблицей переходов:

	0	**1**
$\rightarrow q_1$	q_2	q_1
q_2	q_3	q_1
$*q_3$	q_3	q_2

а) (*****) выпишите все регулярные выражения $R_{ij}^{(0)}$. *Замечание*. Состояние q_i можно рассматривать как состояние с целым номером i;

б) (*****) выпишите все регулярные выражения $R_{ij}^{(1)}$. Постарайтесь максимально упростить эти выражения;

в) выпишите все регулярные выражения $R_{ij}^{(2)}$. Постарайтесь максимально упростить эти выражения;

г) напишите регулярное выражение для языка заданного автомата;

д) (*****) постройте диаграмму переходов для этого ДКА и напишите регулярное выражение для его языка, исключив состояние q_2.

3.2.2. Повторите упражнение 3.2.1 для следующего ДКА.

	0	**1**
$\rightarrow q_1$	q_2	q_3
q_2	q_1	q_3
$*q_3$	q_2	q_1

Отметим, что решения для пунктов *а*, *б* и *д* непригодны в данном упражнении.

3.2.3. Преобразуйте следующий ДКА в регулярное выражение, используя технику исключения состояний из раздела 3.2.2.

	0	**1**
$\rightarrow *p$	s	p
q	p	s
r	r	q
s	q	r

3.2.4. Преобразуйте следующие регулярные выражения в НКА с ε-переходами;

а) (*) $\mathbf{01}^*$;

б) $(\mathbf{0} + \mathbf{1})\mathbf{01}$;

в) $\mathbf{00}(\mathbf{0} + \mathbf{1})^*$.

3.2.5. Исключите ε-переходы из НКА, полученных вами в упражнении 3.2.4. Решение для пункта *а* можно найти на Web-страницах этой книги.

3.2.6. (!) Пусть $A = (Q, \Sigma, \delta, q_0, \{q_f\})$ — это такой ε-НКА, в котором нет переходов в состояние q_0 и из состояния q_f. Опишите язык, допускаемый каждой из следующих модификаций автомата A (в терминах языка $L = L(A)$):

а) (*) автомат, образованный по A путем добавления ε-перехода из q_f в q_0;

б) (*) автомат, построенный по A с помощью добавления ε-перехода из состояния q_0 в каждое состояние, достижимое из q_0 (по путям, метки которых могут содержать как символы из Σ, так и ε);

в) автомат, полученный по A посредством добавления ε-перехода в q_f из каждого состояния, из которого по какому-либо пути достижимо q_f;

г) автомат, построенный по A путем одновременного выполнения пунктов *б* и *в*.

3.2.7. (!!) Существует несколько упрощений конструкции теоремы 3.7, в которой регулярное выражение преобразовывалось в ε-НКА. Вот три из них.

1. Для оператора объединения вместо создания новых начального и допускающего состояний можно сгруппировать оба начальных состояния в одно, сохранив все их переходы. Аналогично, можно сгруппировать оба допускающих состояния в одно; к нему ведут все переходы, которые вели к каждому из исходных состояний.
2. Для оператора конкатенации можно объединять допускающее состояние первого автомата с начальным состоянием второго.
3. Для оператора итерации можно просто добавить ε-переходы из допускающего состояния в начальное, и наоборот.

В результате каждого из этих упрощений мы по-прежнему получаем правильную конструкцию, т.е. искомый ε-НКА, который для любого регулярного выра-

жения допускает язык этого выражения. Сочетание каких усовершенствований (1, 2 или 3) можно применить к этой конструкции, чтобы в результате получался правильный автомат для любого регулярного выражения?

3.2.8. (***!!**) Постройте алгоритм, который по данному ДКА A вычисляет количество цепочек длины n, допускаемых ДКА A (n не связано с количеством состояний автомата A). Ваш алгоритм должен быть полиномиальным как относительно n, так и относительно количества состояний A. *Указание*. Используйте технику, предложенную в доказательстве теоремы 3.4.

3.3. Применение регулярных выражений

Основным средством приложений для поиска образцов (образов, шаблонов) в тексте являются регулярные выражения, задающие "схему" образца, который нужно распознать. Регулярные выражения компилируются в детерминированные или недетерминированные автоматы, которые затем моделируются для получения программы распознавания образов в тексте. В этом разделе мы рассмотрим два важных класса приложений, основанных на регулярных выражениях: лексические анализаторы и поиск в тексте.

3.3.1. Регулярные выражения в UNIX

Прежде чем рассмотреть данные приложения, ознакомимся с системой обозначений, используемой в UNIX для расширенных регулярных выражений. Эти обозначения предоставляют много дополнительных возможностей. На самом деле, расширения UNIX включают некоторые особенности, в частности, возможность именовать и ссылаться на предыдущие цепочки, соответствующие шаблону, что, фактически, позволяет распознавать нерегулярные языки. Здесь эти особенности не рассматриваются, но вводятся сокращения, позволяющие записывать сложные регулярные выражения в сжатом виде.

Первое усовершенствование в системе обозначений регулярных выражений связано с тем, что большинство приложений работает с символами в коде ASCII. В наших примерах обычно использовался алфавит {0, 1}. Наличие только двух символов позволяет использовать сокращенные выражения вроде **0** + **1** для обозначения "любого символа". Однако если алфавит состоит, скажем, из 128 символов, то аналогичное выражение включало бы список всех этих символов и стало бы весьма неудобным для написания. Таким образом, регулярные выражения в UNIX позволяют задавать *классы символов* для представления множеств символов в наиболее кратком виде. Существуют следующие правила для классов символов.

- Символ . (точка) обозначает "любой символ".
- Последовательность $[a_1a_2\ldots a_k]$ обозначает регулярное выражение

  $\boldsymbol{a}_1 + \boldsymbol{a}_2 + \ldots + \boldsymbol{a}_k$

Такое обозначение позволяет записывать примерно вдвое меньше символов, поскольку нет необходимости писать знак "+". Например, четыре символа, используемые в операторах сравнения языка *C*, можно выразить в виде `[<>=!]`.

- В квадратных скобках записывается диапазон вида *x*–*y* для обозначения всех символов от *x* до *y* из последовательности символов в коде ASCII. Поскольку коды цифр, а также символов верхнего и нижнего регистров упорядочены, то многие важные классы символов можно записывать с помощью нескольких ключевых цепочек. Например, все цифры могут быть представлены в виде `[0-9]`, символы верхнего регистра могут быть выражены как `[A-Z]`, а множество всех букв и цифр можно записать как `[A-Za-z0-9]`. Если необходимо включить в такой список символов знак минуса, то его помещают в самом начале или в самом конце списка, чтобы не было путаницы с использованием его для обозначения диапазона символов. Например, набор цифр вместе с точкой и знаками плюс и минус, используемый для образования десятичных чисел со знаком, можно записать в виде выражения `[-+.0-9]`. Квадратные скобки и другие символы, имеющие специальные значения в регулярных выражениях UNIX, задаются в качестве обычных символов с помощью обратной косой черты (\)перед ними.
- Для некоторых наиболее часто используемых классов символов введены специальные обозначения. Рассмотрим несколько примеров:
  а) `[:digit:]` обозначает множество из десяти цифр, как и `[0-9]`[4];
  б) `[:alpha:]` обозначает любой символ (латинского) алфавита, как и `[A-Za-z]`;
  в) `[:alnum:]` обозначает все цифры и буквы (буквенные и цифровые символы), как и `[A-Za-z0-9]`.

Кроме того, в регулярных выражениях UNIX используется несколько операторов, с которыми мы раньше не сталкивались. Ни один из этих операторов не расширяет множество выражаемых языков, но иногда облегчает выражение того, что нам нужно.

1. Оператор | используется вместо + для обозначения объединения.
2. Оператор ? значит "ни одного или один из". Таким образом, $R?$ в UNIX означает то же, что и $\varepsilon + R$ в системе записи регулярных выражений, принятой в этой книге.
3. Оператор + значит "один или несколько из". Следовательно, $R+$ в UNIX является сокращением для RR^* в наших обозначениях.

---

[4] Преимущество обозначения *`[:digit:]`* состоит в том, что если вместо кода ASCII используется другой, в котором коды цифр расположены не по порядку, то *`[:digit:]`* все так же будет обозначать *`[0123456789]`*, тогда как *`[0-9]`* будет представлять все символы, коды которых находятся в промежутке между кодами для *`0`* и для *`9`* включительно.

4. Оператор $\{n\}$ обозначает "n копий". Таким образом, $R\{5\}$ в UNIX является сокращенной записью для *RRRRR* в наших обозначениях.

Отметим, что в регулярных выражениях UNIX для группирования подвыражений используются скобки, как и в регулярных выражениях из раздела 3.1.2, и тот же порядок приоритетов операторов (в этом смысле `?`, + и $\{n\}$ трактуются как $*$). Оператор $*$ используется в UNIX (конечно же, не как верхний индекс) в рассмотренном выше значении.

3.3.2. Лексический анализ

Одним из наиболее ранних применений регулярных выражений было использование их для спецификации компонента компилятора, называемого "лексическим анализатором". Этот компонент сканирует исходную программу и распознает все *лексемы*, т.е. подцепочки последовательных символов, логически составляющие единое целое. Типичными примерами лексем являются ключевые слова и идентификаторы, но существует и множество других примеров.

> **Полное описание регулярных выражений в UNIX**
>
> Читатель, желающий ознакомиться с полным списком операторов и сокращений, используемых в системе записи регулярных выражений в UNIX, может найти их на учебных страницах для различных команд. Существуют некоторые различия между версиями UNIX, но команда типа `man grep` выдаст вам обозначения, используемые для команды `grep`, которая является основной. Кстати, "grep" означает "Global (search for) Regular Expression and Print" (Глобальный поиск регулярного выражения и печать).

UNIX-команда `lex` и ее GNU-версия `flex` получают на вход список регулярных выражений в стиле UNIX, за каждым из которых в фигурных скобках следует код, указывающий, что должен делать лексический анализатор, если найдет экземпляр этой лексемы. Такая система называется *генератором лексического анализатора*, поскольку на ее вход поступает высокоуровневое описание лексического анализатора и по этому описанию она создает функцию, которая представляет собой работающий лексический анализатор.

Такие команды, как `lex` и `flex`, оказались очень удобными, поскольку мощность нотации регулярных выражений необходима и достаточна для описания лексем. Эти команды способны использовать процедуру преобразования регулярного выражения в ДКА для того, чтобы генерировать эффективную функцию, разбивающую исходную программу на лексемы. Они превращают задачу построения лексического анализатора в "послеобеденную работу", тогда как до создания этих основанных на регулярных выражениях средств построение лексического анализатора вручную могло занимать несколько месяцев. Кроме того, если по какой-либо причине возникает необходимость модифи-

цировать лексический анализатор, то намного проще изменить одно или два регулярных выражения, чем забираться внутрь загадочного кода, чтобы исправить дефект.

Пример 3.9. На рис. 3.19 приведен пример фрагмента входных данных команды `lex`, описывающих некоторые лексемы языка C. В первой строке обрабатывается ключевое слово `else`, и ее действие заключается в возвращении символьной константы (в данном примере это `ELSE`) в программу синтаксического анализа для дальнейшей обработки. Вторая строка содержит регулярное выражение, описывающее идентификаторы: буква, за которой следует нуль или несколько букв и/или цифр. Ее действие заключается в следующем. Сначала идентификатор заносится в таблицу символов (если его там еще нет). Затем `lex` выделяет найденную лексему в буфере, так что в этой части кода известно, какой именно идентификатор был обнаружен. Наконец, лексический анализатор возвращает символьную константу `ID`, с помощью которой в этом примере обозначаются идентификаторы.

Третий вход на рис. 3.19 предназначен для знака `>=`, который является двухсимвольным оператором. В последнем показанном примере обрабатывается односимвольный оператор `=`. На практике используют выражения, описывающие ключевые слова, знаки и такие символы пунктуации, как запятые или скобки, а также семейства констант, например, числа или цепочки. Многие из этих выражений чрезвычайно просты — это последовательности, состоящие из одного или нескольких определенных символов. Однако для некоторых выражений требуется вся мощь нотации регулярных выражений. Другими примерами успешного применения возможностей регулярных выражений в командах типа `lex` служат целые числа, числа с плавающей точкой, символьные цепочки и комментарии. □

```
else                        {возвращает (ELSE);}
[A-Za-z][A-Za-z0-9]*   {код для ввода найденного идентификатора
                        в таблицу символов;
                        возвращает (ID);
                        }
>=                          {возвращает (GE);}
=                           {возвращает (EQ);}
...
```

Рис. 3.19. Пример входных данных команды `lex`

Преобразование набора выражений, подобных представленным на рис. 3.19, в автомат происходит почти так же, как это было формально описано в предыдущих разделах. Сначала строится автомат для объединения всех этих выражений. Вообще говоря, этот автомат свидетельствует лишь о том, что распознана *какая-то* лексема. Однако если учесть конструкцию теоремы 3.7 для объединения выражений, состояние ε-НКА точно указывает, к какому типу принадлежит распознанная лексема.

Единственная проблема заключается в том, что лексема может одновременно иметь сразу несколько типов; например, цепочка `else` соответствует не только регулярному выражению **else**, но и выражению для идентификаторов. В генераторе лексического анализатора применяется следующее стандартное решение: приоритет отдается выражению, находящемуся в списке первым. Таким образом, если необходимо, чтобы ключевые слова типа `else` были *зарезервированными* (т.е. не использовались в качестве идентификаторов), то они просто перечисляются в списке перед выражением для идентификаторов.

3.3.3. Поиск образцов в тексте

В разделе 2.4.1 мы отметили, что автоматы могут применяться для эффективного поиска наборов определенных слов в таких больших хранилищах данных, как Web. Хотя инструментальные средства и технология для этого пока не настолько хорошо развиты, как для лексических анализаторов, регулярные выражения все же очень полезны для описания процедур поиска желаемых образцов. Как и для лексических анализаторов, возможность перехода от естественного, описательного представления в виде регулярных выражений к эффективной (основанной на автоматах) реализации является важным интеллектуальным средством решения поставленной задачи.

Общая проблема, для решения которой технология регулярных выражений оказалась весьма полезной, заключается в описании нечетко определенного класса образцов в тексте. Нечеткость описания, в сущности, является гарантией того, что с самого начала нет нужды корректно и полно описывать образцы. Не исключено, что мы никогда не сможем получить точное и полное описание. С помощью регулярных выражений можно, не прилагая больших усилий, описывать такие образцы и быстро изменять эти описания, если результат нас не устраивает. Кроме того, “компилятор” для регулярных выражений пригоден и для преобразования записываемых выражений в выполнимый код.

В качестве примера обсудим одну типичную проблему, возникающую во многих Web-приложениях. Предположим, необходимо просмотреть огромное количество Web-страниц и отметить адреса. Возможно, мы хотим составить список электронных адресов. Или пытаемся классифицировать фирмы по их месторасположению, чтобы отвечать на запросы типа “найдите мне ресторан в пределах 10-ти минут езды от того места, где я сейчас нахожусь”.

В частности, мы сосредоточим внимание на распознавании названий улиц. Что такое название улицы? Необходимо это выяснить, и, если во время тестирования программы будет установлено, что пропущены какие-то варианты, то нужно будет изменять выражения таким образом, чтобы включить все, что не было учтено. Начнем с того, что название улицы может заканчиваться словом “Street” (улица) или его сокращением “St.” Однако некоторые люди живут на “Avenues” (проспектах) или “Roads” (шоссе), что тоже может быть записано в сокращенном виде. Таким образом, в качестве окончания нашего выражения можно использовать следующие варианты.

```
Street | St\. | Avenue | Ave\. |Road | Rd\.
```

В этом выражении использованы обозначения в стиле UNIX с вертикальной чертой вместо + для оператора объединения. Обратите также внимание, что перед точками стоит обратная косая черта, поскольку в выражениях UNIX точка имеет специальное значение — "любой символ", а в этом случае нам необходимо, чтобы в конце сокращений стоял символ "точка".

Перед таким обозначением, как `Street`, должно стоять название улицы. Обычно оно состоит из прописной буквы, за которой следует несколько строчных букв. В UNIX этот образец можно описать с помощью выражения `[A-Z][a-z]*`. Однако названия некоторых улиц состоят из нескольких слов, например, "Rhode Island Avenue in Washington DC". Поэтому, обнаружив, что названия такого вида пропущены, необходимо исправить наше описание таким образом, чтобы получилось следующее выражение.

```
'[A-Z][a-z]*( [A-Z][a-z]*)*'
```

Это выражение начинается с группы, состоящей из прописной буквы и, возможно, нескольких строчных букв. Далее может следовать несколько групп, состоящих из пробела, еще одной прописной буквы и, возможно, нескольких строчных. В выражениях UNIX пробел является обычным символом, и, чтобы представленное выше выражение не выглядело в командной строке UNIX как два выражения, разделенных пробелом, нужно все выражение заключить в апострофы. Сами апострофы не являются частью выражения.

Теперь в адрес нужно включить номер дома. Большинство номеров домов представляют собой цепочки из цифр. Однако в некоторых номерах после цифр стоит буква, например, "123A Main St." Поэтому выражение для номеров должно включать необязательную прописную букву после цифр: `[0-9]+[A-Z]?`. Заметьте, что мы здесь использовали UNIX-оператор + для "одной или нескольких" цифр и оператор `?` для "ни одной или одной" прописной буквы. Полное выражение для адресов улиц будет иметь следующий вид.

```
'[0-9]+[A-Z]? [A-Z][a-z]*( [A-Z][a-z]*)*
(Street|St\.|Avenue|Ave\.|Road|Rd\.)'
```

Используя это выражение, получим вполне приемлемый результат. Однако в какой-то момент мы обнаружим, что пропустили следующие случаи.

1. Улицы, которые называются иначе, чем "street", "avenue" или "road". Например, мы упустили "Boulevard" (бульвар), "Place" (площадь), "Way" (дорога) и их сокращения.
2. Названия улиц, которые являются числами или частично содержат числа, например, "42nd Street" (42-я улица).
3. Почтовые ящики и маршруты сельской доставки.
4. Названия улиц, которые не оканчиваются словом типа "Street". Например, "El Camino Real in Silicon Valley". С испанского это переводится как "Королевское шоссе в Силиконовой Долине", но если сказать "El Camino Real Road" ("Королевское

шоссе шоссе"), то это будет повторением. Так что приходится иметь дело с адресами типа "2000 El Camino Real".

5. Все другие странные ситуации, которые мы даже не можем вообразить. А вы можете?

Таким образом, с помощью компилятора регулярных выражений процесс постепенного приближения к полному распознавателю адресов существенно упрощается по сравнению с использованием традиционного языка программирования.

3.3.4. Упражнения к разделу 3.3

3.3.1. (!) Напишите регулярное выражение для описания телефонных номеров всех видов, которые только можно себе представить. Учтите международные номера, а также тот факт, что в разных странах используется разное количество цифр в кодах областей и в локальных номерах телефонов.

3.3.2. (!!) Напишите регулярное выражение для представления заработной платы в том виде, в котором она указывается в объявлениях о работе. Учтите, что может быть указан размер зарплаты в час, в неделю, месяц или год. Она может включать или не включать знак доллара или какой-либо другой единицы, например "К". Рядом может находиться слово или слова, обозначающие, что речь идет о зарплате. Предложение: просмотрите рекламные объявления в газетах или списки вакансий в режиме on-line, чтобы получить представление о том, какие образцы могут вам пригодиться.

3.3.3. (!) В конце раздела 3.3.3 мы привели несколько примеров изменений, которые могут быть внесены в регулярное выражение, описывающее адреса. Модифицируйте построенное там выражение таким образом, чтобы включить все упомянутые варианты.

3.4. Алгебраические законы для регулярных выражений

В примере 3.5 мы столкнулись с необходимостью упрощения регулярных выражений для того, чтобы их размер не превышал разумные пределы. Мы объяснили, почему одно выражение можно было заменить другим. Во всех рассмотренных ситуациях основной вывод заключался в том, что эти выражения оказывались *эквивалентными*, т.е. задавали один и тот же язык. В этом разделе будет предложен ряд алгебраических законов, позволяющих рассматривать вопрос эквивалентности двух регулярных выражений на более высоком уровне. Вместо исследования определенных регулярных выражений, мы рассмотрим пары регулярных выражений с переменными в качестве аргументов. Два выражения с переменными являются *эквивалентными*, если при подстановке любых языков вместо переменных оба выражения представляют один и тот же язык.

Рассмотрим подобный пример в алгебре обычной арифметики. Одно дело сказать, что 1 + 2 = 2 + 1. Данный частный случай закона коммутативности операции сложения легко проверить: выполнив операцию сложения в обеих частях этого равенства, получим 3 = 3. Однако *коммутативный закон сложения* утверждает большее, а именно, что $x + y = y + x$, где x и y — переменные, которые можно заменять любыми двумя числами. Следовательно, он гласит, что, какие бы числа мы ни складывали, получим один и тот же результат независимо от порядка суммирования.

Регулярные выражения, подобно обычным арифметическим, подчиняются ряду законов. Многие из них подобны законам арифметики, если рассматривать объединение как сложение, а конкатенацию — как умножение. Однако в некоторых случаях эта аналогия нарушается. Кроме того, существует ряд законов для регулярных выражений, которым в арифметике аналогов нет, особенно если речь идет об операторе итерации. Следующие разделы содержат перечень главных законов для регулярных выражений. В завершение обсуждается вопрос, как вообще можно проверить, является ли некоторый формулируемый для регулярных выражений закон на самом деле законом, т.е. выполняется ли он для любых языков, подставляемых вместо его переменных.

3.4.1. Ассоциативность и коммутативность

Коммутативность — это свойство операции, заключающееся в том, что при перестановке ее операндов результат не меняется. Арифметический пример мы уже приводили выше: $x + y = y + x$. *Ассоциативность* — это свойство операции, позволяющее перегруппировывать операнды, если оператор применяется дважды. Например, ассоциативный закон умножения имеет вид: $(x \times y) \times z = x \times (y \times z)$. Для регулярных выражений выполняются три закона такого типа.

- $L + M = M + L$, т.е. *коммутативный закон для объединения* утверждает, что два языка можно объединять в любом порядке.
- $(L + M) + N = L + (M + N)$. Этот закон — *ассоциативный закон объединения* — говорит, что для объединения трех языков можно сначала объединить как два первых, так и два последних из них. Обратите внимание на то, что вместе с коммутативным законом объединения этот закон позволяет объединять любое количество языков в произвольном порядке, разбивая их на любые группы, и результат будет одним и тем же. Очевидно, что некоторая цепочка принадлежит объединению $L_1 \cup L_2 \cup \ldots \cup L_k$ тогда и только тогда, когда она принадлежит одному или нескольким языкам L_i.
- $(LM)N = L(MN)$. Этот *ассоциативный закон конкатенации* гласит, что для конкатенации трех языков можно сначала соединить как два первых, так и два последних из них.

В предложенном списке нет "закона" $LM = ML$, гласящего, что операция конкатенации является коммутативной, поскольку это утверждение неверно.

Пример 3.10. Рассмотрим регулярные выражения **01** и **10**. Эти выражения задают языки {01} и {10}, соответственно. Поскольку эти языки различаются, то общий закон $LM = ML$ для них не выполняется. Если бы он выполнялся, то можно было бы подставить регулярное выражение **0** вместо L и **1** вместо M и получить неверное заключение **01** = **10**. □

3.4.2. Единичные и нулевые элементы

Единичным (нейтральным) элементом (*единицей*) операции называется элемент, для которого верно следующее утверждение: если данная операция применяется к единичному элементу и некоторому другому элементу, то результат равен другому элементу. Например, 0 является единичным элементом операции сложения, поскольку $0 + x = x + 0 = x$, а 1 — единичным элементом для умножения, потому что $1 \times x = x \times 1 = x$. *Нулевым элементом* (*нулем*, *аннулятором*) операции называется элемент, для которого истинно следующее: результатом применения данной операции к нулевому и любому другому элементу является нулевой элемент. Например, 0 является нулевым элементом умножения, поскольку $0 \times x = x \times 0 = 0$. Операция сложения нулевого элемента не имеет.

Для регулярных выражений существует три закона, связанных с этими понятиями.

- $\varnothing + L = L + \varnothing = L$. Этот закон утверждает, что $\varnothing$ является единицей объединения.
- $\varepsilon L = L\varepsilon = L$. Этот закон гласит, что ε является единицей конкатенации.
- $\varnothing L = L\varnothing = \varnothing$. Этот закон утверждает, что $\varnothing$ является нулевым элементом конкатенации.

Эти законы являются мощным средством упрощения выражений. Например, если необходимо объединить несколько выражений, часть которых упрощена до $\varnothing$, то $\varnothing$ можно исключить из объединения. Аналогично, если при конкатенации нескольких выражений некоторые из них можно упростить до ε, то ε можно исключить из конкатенации. Наконец, если при конкатенации любого количества выражений хотя бы одно из них равно $\varnothing$, то результат такой конкатенации — $\varnothing$.

3.4.3. Дистрибутивные законы

Дистрибутивный закон связывает две операции и утверждает, что одну из операций можно применить отдельно к каждому аргументу другой операции. Арифметическим примером этого закона является дистрибутивный закон умножения относительно сложения, т.е. $x \times (y + z) = x \times y + x \times z$. Поскольку умножение коммутативно, то не имеет значения, с какой стороны, слева или справа от суммы, применяется умножение. Для регулярных выражений существует аналогичный закон, но, поскольку операция конкатенации некоммутативна, то мы сформулируем его в виде следующих двух законов.

- $L(M + N) = LM + LN$. Этот закон называется *левосторонним дистрибутивным законом конкатенации относительно объединения*.
- $(M + N)L = ML + NL$. Этот закон называется *правосторонним дистрибутивным законом конкатенации относительно объединения*.

Докажем левосторонний дистрибутивный закон. Правосторонний доказывается аналогично. В доказательстве используются только языки, и оно не зависит от того, существуют ли для этих языков регулярные выражения.

Теорема 3.11. Если L, M и N — произвольные языки, то

$$L(M \cup N) = LM \cup LN$$

Доказательство. Это доказательство аналогично доказательству дистрибутивного закона, представленного в теореме 1.10. Требуется показать, что цепочка w принадлежит $L(M \cup N)$ тогда и только тогда, когда она принадлежит $LM \cup LN$.

(Необходимость) Если w принадлежит $L(M \cup N)$, то $w = xy$, где x принадлежит L, а y принадлежит либо M, либо N. Если y принадлежит M, то xy принадлежит LM и, следовательно, принадлежит $LM \cup LN$. Аналогично, если y принадлежит N, то xy принадлежит LN и, следовательно, принадлежит $LM \cup LN$.

(Достаточность) Предположим, что w принадлежит $LM \cup LN$. Тогда w принадлежит либо LM, либо LN. Пусть w принадлежит LM. Тогда $w = xy$, где x принадлежит L, а y принадлежит M. Если y принадлежит M, то y также содержится в $M \cup N$. Следовательно, xy принадлежит $L(M \cup N)$. Если w не принадлежит LM, то она точно содержится в LN, и аналогичные рассуждения показывают, что w принадлежит $L(M \cup N)$. □

Пример 3.12. Рассмотрим регулярное выражение $\mathbf{0} + \mathbf{01}^*$. В объединении можно "вынести за скобки $\mathbf{0}$", но сначала необходимо представить выражение $\mathbf{0}$ в виде конкатенации $\mathbf{0}$ с чем-то еще, а именно с ε. Используем свойства единичного элемента для конкатенации, меняя $\mathbf{0}$ на $\mathbf{0}\varepsilon$, в результате чего получим выражение $\mathbf{0}\varepsilon + \mathbf{01}^*$. Теперь можно применить левосторонний дистрибутивный закон, чтобы заменить это выражение на $\mathbf{0}(\varepsilon + \mathbf{1}^*)$. Далее, учитывая, что ε принадлежит $L(\mathbf{1}^*)$, заметим, что $\varepsilon + \mathbf{1}^* = \mathbf{1}^*$, и окончательно упростим выражение до $\mathbf{01}^*$. □

3.4.4. Закон идемпотентности

Операция называется *идемпотентной*, если результат применения этой операции к двум одинаковым значениям как операндам равен этому значению. Обычные арифметические операции не являются идемпотентными. В общем случае $x + x \neq x$ и $x \times x \neq x$ (хотя существуют *некоторые* значения x, для которых это равенство выполняется, например $0 + 0 = 0$). Однако объединение и пересечение являются идемпотентными операциями, поэтому для регулярных выражений справедлив следующий закон.

- $L + L = L$. Этот закон — *закон идемпотентности операции объединения* — утверждает, что объединение двух одинаковых выражений можно заменить одним таким выражением.

3.4.5. Законы, связанные с оператором итерации

Существует ряд законов, связанных с операцией итерации и ее разновидностями + и ? в стиле UNIX. Перечислим эти законы и вкратце поясним, почему они справедливы.

- $(L^*)^* = L^*$. Этот закон утверждает, что при повторной итерации язык уже итерированного выражения не меняется. Язык выражения $(L^*)^*$ содержит все цепочки, образованные конкатенацией цепочек языка L^*. Последние же цепочки построены из цепочек языка L. Таким образом, цепочки языка $(L^*)^*$ также являются конкатенациями цепочек из L и, следовательно, принадлежат языку L^*.
- $\varnothing^* = \varepsilon$. Итерация языка $\varnothing$ состоит из одной-единственной цепочки ε, что уже обсуждалось в примере 3.6.
- $\varepsilon^* = \varepsilon$. Легко проверить, что единственной цепочкой, которую можно образовать конкатенацией любого количества пустых цепочек, будет все та же пустая цепочка.
- $L^+ = LL^* = L^*L$. Напомним, что L^+ по определению равно $L + LL + LLL + \ldots$. Поскольку $L^* = \varepsilon + L + LL + LLL + \ldots$, то

  $LL^* = L\varepsilon + LL + LLL + LLLL + \ldots$

  Если учесть, что $L\varepsilon = L$, то очевидно, что бесконечные разложения для LL^* и для L^+ совпадают. Это доказывает, что $L^+ = LL^*$. Доказательство того, что $L^+ = L^*L$, аналогично[5].
- $L^* = L^+ + \varepsilon$. Это легко доказать, поскольку в разложении $L+$ присутствуют те же члены, что и в разложении L^*, за исключением цепочки ε. Заметим, что если язык L содержит цепочку ε, то слагаемое "$+\varepsilon$" лишнее, т.е. в этом случае $L^+ = L^*$.
- $L? = \varepsilon + L$. В действительности это правило является определением оператора "?".

3.4.6. Установление законов для регулярных выражений

Каждый из вышеперечисленных законов был доказан, формально или неформально. Однако есть еще бесконечно много законов для регулярных выражений, которые можно было бы сформулировать. Существует ли некая общая методика, с помощью которой можно было бы легко доказывать истинность таких законов? Оказывается, что вопрос о справедливости некоторого закона сводится к вопросу о равенстве двух определенных

[5] Как следствие, отметим, что любой язык коммутирует со своей итерацией: $LL^* = L^*L$. Это свойство не противоречит тому, что в общем случае конкатенация не является коммутативной.

языков. Интересно, что эта методика тесно связана с регулярными операциями и не может быть распространена на выражения, использующие некоторые другие операции, например пересечение.

Чтобы проиллюстрировать суть этой методики, рассмотрим следующий закон.

$$(L + M)^* = (L^*M^*)^*$$

Этот закон утверждает, что в результате итерации объединения двух произвольных языков получим тот же язык, что и в результате итерации языка L^*M^*, состоящего из всех цепочек, образованных из нуля или нескольких цепочек из L, за которыми следует нуль или несколько цепочек из M.

Чтобы доказать этот закон, сначала предположим, что цепочка w принадлежит языку выражения $(L + M)^*$[6]. Тогда $w = w_1w_2...w_k$ для некоторого k, где каждая цепочка w_i принадлежит либо L, либо M. Из этого следует, что каждая такая цепочка w_i принадлежит языку L^*M^*. Действительно, если цепочка w_i принадлежит языку L, то она также принадлежит языку L^*. Тогда из M не берем ни одной цепочки, т.е. из M^* выбираем ε. Если w_i принадлежит M, доказательство аналогично. Если каждая w_i принадлежит L^*M^*, то w принадлежит итерации этого языка.

Чтобы завершить доказательство, необходимо доказать обратное утверждение, т.е. что все цепочки из $(L^*M^*)^*$ принадлежат также $(L + M)^*$. Опустим эту часть доказательства, поскольку нашей целью является не доказательство данного закона, а установление следующего важнейшего свойства регулярных выражений.

Любое регулярное выражение с переменными можно рассматривать как *конкретное* регулярное выражение, не содержащее переменных, если считать, что каждая переменная — это просто отдельный символ. Например, в выражении $(L + M)^*$ переменные L и M можно заменить символами a и b, соответственно, и получить регулярное выражение $(\mathbf{a} + \mathbf{b})^*$.

Язык конкретного выражения дает представление о виде цепочек любого языка, образованного из исходного выражения в результате подстановки произвольных языков вместо переменных. Например, при анализе выражения $(L + M)^*$ мы отметили, что любая цепочка w, составленная из последовательности цепочек, выбираемых либо из L, либо из M, принадлежит языку $(L + M)^*$. Можно прийти к тому же заключению, рассмотрев язык конкретного выражения $L((\mathbf{a} + \mathbf{b})^*)$, который, очевидно, представляет собой множество всех цепочек, состоящих из символов a и b. В одну из цепочек этого множества можно подставить любую цепочку из L вместо символа a и любую цепочку из M вместо символа b. При этом различные вхождения символов a и b можно заменять различными цепочками. Если такую подстановку осуществить для всех цепочек из $(\mathbf{a} + \mathbf{b})^*$, то в результате получим множество всех цепочек, образованных конкатенацией цепочек из L и/или M в любом порядке.

---

[6] Для простоты отождествим регулярные выражения с их языками, чтобы не повторять фразу “язык выражения” перед каждым регулярным выражением.

Сформулированное выше утверждение может показаться очевидным, но, как указано во врезке “Расширение данной проверки за пределы регулярных выражений может оказаться ошибочным”, оно будет неверным, если к трем операциям регулярных выражений добавить некоторые другие операции. В следующей теореме доказывается общий закон для регулярных выражений.

Теорема 3.13. Пусть E — регулярное выражение с переменными L_1, L_2, …, L_m. Построим конкретное регулярное выражение C, заменив каждое вхождение L_i символом a_i, $i = 1, 2, \ldots, m$. Тогда для произвольных языков L_1, L_2, …, L_m любую цепочку w из $L(E)$ можно представить в виде $w = w_1w_2\ldots w_k$, где каждая w_i принадлежит одному из этих языков, например L_{j_i}, а цепочка $a_{j_1}a_{j_2}\ldots a_{j_k}$ принадлежит языку $L(C)$. Говоря менее формально, мы можем построить $L(E)$, исходя из каждой цепочки языка $L(C)$, скажем, $a_{j_1}a_{j_2}\ldots a_{j_k}$, и заменяя в ней каждый из символов a_{j_i} любой цепочкой из соответствующего языка L_{j_i}.

Доказательство. Доказательство проведем структурной индукцией по выражению E.

Базис. Базисными являются случаи, когда E представляет собой ε, $\varnothing$ или переменную L. В первых двух случаях нечего доказывать, потому что конкретное выражение C совпадает с E. Если же E есть переменная L, то $L(E) = L$. Конкретное выражение C равно просто **a**, где a — символ, соответствующий переменной L. Следовательно, $L(C) = \{a\}$. Если в эту единственную цепочку вместо символа a подставить любую цепочку из L, то получим язык L, который есть также $L(E)$.

Индукция. Рассмотрим три случая в зависимости от заключительной операции выражения E. Сначала предположим, что $E = F + G$, т.е. заключительной является операция объединения. Пусть C и D — конкретные выражения, построенные соответственно по F и G с помощью подстановки в эти выражения определенных символов вместо языковых переменных. Заметим, что в оба выражения F и G вместо всех одинаковых переменных должны быть подставлены одинаковые символы. Тогда конкретное выражение, полученное из выражения E, равно $C + D$, и $L(C + D) = L(C) + L(D)$.

Предположим, что w — цепочка из языка $L(E)$, полученная в результате замены языковых переменных выражения E некоторыми определенными языками. Тогда w принадлежит либо $L(F)$, либо $L(G)$. Согласно индуктивной гипотезе цепочка w получена, исходя из некоторой конкретной цепочки w_1, принадлежащей $L(C)$ или $L(D)$, соответственно, с помощью подстановки цепочек из соответствующих языков вместо символов цепочки w_1. Таким образом, в обоих случаях цепочка w может быть построена, начиная с некоторой конкретной цепочки w_1 из $L(C + D)$, путем одних и тех же подстановок цепочек вместо символов.

Необходимо также рассмотреть случаи, когда E представляет собой FG или F^*. Однако доказательства для конкатенации и итерации аналогичны приведенному выше доказательству для объединения, поэтому оставляем их читателю. □

3.4.7. Проверка истинности алгебраических законов для регулярных выражений

Теперь можно сформулировать и обосновать проверку истинности законов для регулярных выражений. Проверка истинности закона $E = F$, где E и F — два регулярных выражения с одним и тем же набором переменных, состоит в следующем.

1. Преобразуем E и F в конкретные регулярные выражения C и D соответственно, заменяя каждую переменную конкретным символом.
2. Проверим равенство $L(C) = L(D)$. Если оно выполняется, то закон $E = F$ истинен, а если нет — ложен. Заметим, что проверять, определяют ли два регулярных выражения один и тот же язык, мы научимся в разделе 4.4. Однако можно использовать некоторые специальные (ad-hoc) средства для проверки равенства пар интересующих нас языков. Напомним, что если языки *не* совпадают, то достаточно построить контрпример, т.е. найти хотя бы одну цепочку, принадлежащую только одному из них.

Теорема 3.14. Предложенная выше проверка правильно определяет истинность законов для регулярных выражений.

Доказательство. Докажем, что $L(E) = L(F)$ для любых языков, подставленных вместо переменных E и F, тогда и только тогда, когда $L(C) = L(D)$.

(Необходимость) Предположим, что $L(E) = L(F)$ для любых языков, подставляемых вместо переменных. В частности, выберем для каждой переменной L конкретный символ a, заменяющий L в выражениях C и D. Тогда $L(C) = L(E)$ и $L(D) = L(F)$. Поскольку мы предположили, что $L(E) = L(F)$, то $L(C) = L(D)$.

(Достаточность) Теперь предположим, что $L(C) = L(D)$. Согласно теореме 3.13 $L(E)$ и $L(F)$ построены с помощью замены конкретных символов в цепочках из $L(C)$ и $L(D)$ цепочками из языков, соответствующих этим символам. Если $L(C)$ и $L(D)$ состоят из одних и тех же цепочек, то оба языка, построенные таким способом, тоже будут совпадать; т.е. $L(E) = L(F)$. □

Пример 3.15. Проанализируем предполагаемый закон $(L + M)^* = (L^*M^*)^*$. Если заменить переменные L и M, соответственно, конкретными символами a и b, получим регулярные выражения $(\mathbf{a} + \mathbf{b})^*$ и $(\mathbf{a}^*\mathbf{b}^*)^*$. Легко убедиться в том, что оба эти выражения задают язык всех возможных цепочек, составленных из a и b. Следовательно, оба конкретных выражения представляют один и тот же язык, и данный закон выполняется.

В качестве еще одного примера рассмотрим закон $L^* = L^*L^*$. Конкретными языками будут $\mathbf{a}^*$ и $\mathbf{a}^*\mathbf{a}^*$, соответственно, и каждый из них представляет собой множество всех цепочек, состоящих из a. Снова видим, что данный закон выполняется, т.е. конкатенация итераций одного и того же языка дает ту же самую итерацию.

Наконец, рассмотрим предполагаемый закон $L + ML = (L + M)L$. Если заменить символами a и b переменные L и M, соответственно, то получим два конкретных выражения $\mathbf{a} + \mathbf{ba}$ и $(\mathbf{a} + \mathbf{b})\mathbf{a}$. Однако языки этих выражений не совпадают. Например, цепочка aa

принадлежит второму языку, но не принадлежит первому. Следовательно, этот предполагаемый закон ложен. □

Расширение данной проверки за пределы регулярных выражений может оказаться ошибочным

Рассмотрим расширенную алгебру регулярных выражений, включающую операцию пересечения. Интересно, что добавление операции $\cap$ к трем представленным ранее операциям регулярных выражений не увеличивает множество задаваемых языков, что будет доказано ниже в теореме 4.8. В то же время сформулированная выше проверка алгебраических законов перестает работать.

Рассмотрим "закон" $L \cap M \cap N = L \cap M$, утверждающий, что пересечение некоторых трех языков равно пересечению только двух первых из них. Очевидно, что этот закон ложен. Например, если $L = M = \{a\}$, а $N = \emptyset$. Но проверка, основанная на конкретизации переменных, может не определить ложность этого закона. Если мы заменим L, M и N символами a, b и c, соответственно, то должны будем проверить равенство $\{a\} \cap \{b\} \cap \{c\} = \{a\} \cap \{b\}$. Поскольку обе части этого соотношения являются пустым множеством, равенство языков выполняется, и согласно нашей проверке этот "закон" будет истинным, хотя в действительности это не так.

3.4.8. Упражнения к разделу 3.4

3.4.1. Проверьте следующие тождества для регулярных выражений:

а) (∗) $R + S = S + R$;

б) $(R + S) + T = R + (S + T)$;

в) $(RS)T = R(ST)$;

г) $R(S + T) = RS + RT$;

д) $(R + S)T = RT + ST$;

е) (∗) $(R^*)^* = R^*$;

ж) $(\varepsilon + R)^* = R^*$;

з) $(R^*S^*)^* = (R + S)^*$.

3.4.2. (!) Докажите или опровергните каждое из следующих утверждений для регулярных выражений:

а) (∗) $(R + S)^* = R^* + S^*$;

б) $(RS + R)^*R = R(SR + R)^*$;

в) (∗) $(RS + R)^*RS = (RR^*S)^*$;

г) $(R + S)^*S = (R^*S)^*$;

д) $S(RS + S)^*R = RR^*S(RR^*S)^*$.

3.4.3. В примере 3.6 было построено регулярное выражение

$(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1}) + (\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$.

С помощью дистрибутивных законов преобразуйте его в два различных, более простых, эквивалентных выражения.

3.4.4. В начале раздела 3.4.6 приведена часть доказательства того, что $(L^*M^*)^* = (L + M)^*$. Завершите это доказательство, показав, что все цепочки из $(L^*M^*)^*$ принадлежат также $(L + M)^*$.

3.4.5. (!) Завершите доказательство теоремы 3.13 для случаев, когда регулярное выражение E представляет собой FG или F^*.

Резюме

- *Регулярные выражения*. Этот алгебраический способ описания задает те же языки, что и конечные автоматы, а именно, регулярные языки. Регулярными операторами являются объединение, конкатенация (“точка”) и итерация (“звездочка”).
- *Регулярные выражения на практике*. Системы, подобные UNIX, и различные их команды используют язык расширенных регулярных выражений, существенно упрощающий записи многих обычных выражений. Классы символов позволяют легко записывать выражения для наборов символов, а такие операторы, как “один или несколько из” и “не более, чем один из”, расширяют круг обычных регулярных операторов.
- *Эквивалентность регулярных выражений и конечных автоматов*. Произвольный ДКА можно преобразовать в регулярное выражение с помощью индуктивной процедуры, в которой последовательно строятся выражения для меток путей, проходящих через постепенно увеличивающиеся множества состояний. В качестве альтернативы преобразованию ДКА в регулярное выражение можно также использовать метод исключения состояний. С другой стороны, мы можем рекурсивно построить ε-НКА по регулярному выражению, а потом в случае необходимости преобразовать полученный ε-НКА в ДКА.
- *Алгебра регулярных выражений*. Регулярные выражения подчиняются многим алгебраическим законам арифметики, хотя есть и различия. Объединение и конкатенация ассоциативны, но только объединение коммутативно. Конкатенация дистрибутивна относительно объединения. Объединение идемпотентно.
- *Проверка истинности алгебраических тождеств*. Чтобы проверить эквивалентность регулярных выражений с переменными в качестве аргументов, необходимо подставить вместо этих переменных различные константы и проверить, будут ли совпадать языки, полученные в результате.

Литература

Идея регулярных выражений и доказательство их эквивалентности конечным автоматам представлены в работе Клини [3]. Преобразование регулярного выражения в ε-НКА в том виде, как оно выглядит в этой книге, известно как “Метод Мак-Нотона-Ямады” из работы [4]. Проверку тождеств регулярных выражений, в которой переменные рассматриваются как константы, предложил Дж. Гишер [2]. В его отчете, который считается устным, продемонстрировано, как добавление нескольких других операций, например, пересечения или перемешивания (см. упражнение 7.3.4), приводит к ошибочности данной проверки, хотя класс задаваемых языком при этом не изменяется.

Еще до появления системы UNIX К. Томпсон исследовал возможность применения регулярных выражений в таких командах, как `grep`, и придуманный им алгоритм выполнения таких команд можно найти в [5]. На ранней стадии развития UNIX появились другие команды, в которых активно использовались расширенные регулярные выражения, например, команда `lex`, предложенная М. Леском. Описание этой команды и других технологий, связанных с регулярными выражениями, можно найти в [1].

1. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading MA, 1986. (Ахо А. В., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструменты. — М.: Издательский дом “Вильямс”, 2001.)
2. J. L. Gisher, STAN-CS-TR-84-1033 (1984).
3. S. C. Kleene, “Representation of events in nerve nets and finite automata”, In C. E. Shannon and J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956, pp. 3–42. (Клини С.К. Представление событий в нервных сетях. / сб. “*Автоматы*”. — М.: ИЛ, 1956. — С. 15–67.)
4. R. McNaughton and H. Yamada, “Regular expressions and state graphs for automata”, *IEEE Trans. Electronic Computers* **9**:1 (Jan., 1960), pp. 39–47.
5. K. Thompson, “Regular expression search algorithm”, *Comm. ACM* **11**:6 (June, 1968), pp. 419–422.

ГЛАВА 4

Свойства регулярных языков

В этой главе рассматриваются свойства регулярных языков. В разделе 4.1 предлагается инструмент для доказательства нерегулярности некоторых языков — теорема, которая называется “леммой о накачке” (“pumping lemma”)[1].

Одними из важнейших свойств регулярных языков являются “свойства замкнутости”. Эти свойства позволяют создавать распознаватели для одних языков, построенных из других с помощью определенных операций. Например, пересечение двух регулярных языков также является регулярным. Таким образом, при наличии автоматов для двух различных регулярных языков можно (механически) построить автомат, который распознает их пересечение. Поскольку автомат для пересечения языков может содержать намного больше состояний, чем любой из двух данных автоматов, то “свойство замкнутости” может оказаться полезным инструментом для построения сложных автоматов. Конструкция для пересечения уже использовалась в разделе 2.1.

Еще одну важную группу свойств регулярных языков образуют “свойства разрешимости”. Изучение этих свойств позволяет ответить на важнейшие вопросы, связанные с автоматами. Так, можно выяснить, определяют ли два различных автомата один и тот же язык. Разрешимость этой задачи позволяет “минимизировать” автоматы, т.е. по данному автомату найти эквивалентный ему с наименьшим возможным количеством состояний. Задача минимизации уже в течение десятилетий имеет большое значение при проектировании переключательных схем, поскольку стоимость схемы (площади чипа, занимаемого схемой) снижается при уменьшении количества состояний автомата, реализованного схемой.

4.1. Доказательство нерегулярности языков

В предыдущих разделах было установлено, что класс языков, известных как регулярные, имеет не менее четырех различных способов описания. Это языки, допускаемые ДКА, НКА и ε-НКА; их можно также определять с помощью регулярных выражений.

Не каждый язык является регулярным. В этом разделе предлагается мощная техника доказательства нерегулярности некоторых языков, известная как “лемма о накачке”. Ни-

---

[1] В русскоязычной литературе в свое время был принят термин “лемма о разрастании”. Однако, на наш взгляд, “накачка” точнее отражает суть происходящего. — *Прим. ред.*

же приводится несколько примеров нерегулярных языков. В разделе 4.2 лемма о накачке используется вместе со свойствами замкнутости регулярных языков для доказательства нерегулярности других языков.

4.1.1. Лемма о накачке для регулярных языков

Рассмотрим язык $L_{01} = \{0^n1^n \mid n \geq 1\}$. Этот язык состоит из всех цепочек вида 01, 0011, 000111 и так далее, содержащих один или несколько нулей, за которыми следует такое же количество единиц. Утверждается, что язык L_{01} нерегулярен. Неформально, если бы L_{01} был регулярным языком, то допускался бы некоторым ДКА A, имеющим какое-то число состояний k. Предположим, что на вход автомата поступает k нулей. Он находится в некотором состоянии после чтения каждого из $k + 1$ префиксов входной цепочки, т.е. ε, 0, 00, …, 0^k. Поскольку есть только k различных состояний, то согласно "принципу голубятни", прочитав два различных префикса, например, 0^i и 0^j, автомат должен находится в одном и том же состоянии, скажем, q.

Допустим, что, прочитав i или j нулей, автомат A получает на вход 1. По прочтении i единиц он должен допустить вход, если ранее получил i нулей, и отвергнуть его, получив j нулей. Но в момент поступления 1 автомат A находится в состоянии q и не способен "вспомнить", какое число нулей, i или j, было принято. Следовательно, его можно "обманывать" и заставлять работать неправильно, т.е. допускать, когда он не должен этого делать, или наоборот.

Приведенное неформальное доказательство можно сделать точным. Однако к заключению о нерегулярности языка L_{01} приводит следующий общий результат.

Теорема 4.1 (*лемма о накачке для регулярных языков*). Пусть L — регулярный язык. Существует константа n (зависящая от L), для которой каждую цепочку w из языка L, удовлетворяющую неравенству $|w| \geq n$, можно разбить на три цепочки $w = xyz$ так, что выполняются следующие условия.

1. $y \neq \varepsilon$.
2. $|xy| \leq n$.
3. Для любого $k \geq 0$ цепочка xy^kz также принадлежит L.

Это значит, что всегда можно найти такую цепочку y недалеко от начала цепочки w, которую можно "накачать". Таким образом, если цепочку y повторить любое число раз или удалить (при $k = 0$), то результирующая цепочка все равно будет принадлежать языку L.

Доказательство. Пусть L — регулярный язык. Тогда $L = L(A)$ для некоторого ДКА A. Пусть A имеет n состояний. Рассмотрим произвольную цепочку w длиной не менее n, скажем, $w = a_1a_2\ldots a_m$, где $m \geq n$ и каждый a_i есть входной символ. Для $i = 0, 1, 2, \ldots, n$ определим состояние p_i как $\hat{\delta}(q_0, a_1a_2\ldots a_i)$, где δ — функция переходов автомата A, а q_0 — его начальное состояние. Заметим, что $p_0 = q_0$.

Рассмотрим $n + 1$ состояний p_i при $i = 0, 1, 2, \ldots, n$. Поскольку автомат A имеет n различных состояний, то по "принципу голубятни" найдутся два разных целых числа i и j ($0 \le i < j \le n$), при которых $p_i = p_j$. Теперь разобьем цепочку w на xyz.

1. $x = a_1a_2\ldots a_i$.
2. $y = a_{i+1}a_{i+2}\ldots a_j$.
3. $z = a_{j+1}a_{j+2}\ldots a_m$.

Таким образом, x приводит в состояние p_i, y — из p_i обратно в p_i (так как $p_i = p_j$), а z — это остаток цепочки w. Взаимосвязи между цепочками и состояниями показаны на рис. 4.1. Заметим, что цепочка x может быть пустой при $i = 0$, а z — при $j = n = m$. Однако цепочка y не может быть пустой, поскольку i строго меньше j.

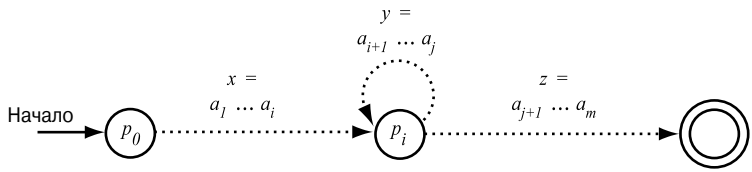


Рис. 4.1. Каждая цепочка, длина которой больше числа состояний автомата, приводит к повторению некоторого состояния

Теперь посмотрим, что происходит, когда на вход автомата A поступает цепочка xy^kz для любого $k \ge 0$. При $k = 0$ автомат переходит из начального состояния q_0 (которое есть также p_0) в p_i, прочитав x. Поскольку $p_i = p_j$, то z переводит A из p_i в допускающее состояние (см. рис. 4.1).

Если $k > 0$, то по x автомат A переходит из q_0 в p_i, затем, читая y^k, k раз циклически проходит через p_i, и, наконец, по z переходит в допускающее состояние. Таким образом, для любого $k \ge 0$ цепочка xy^kz также допускается автоматом A, т.е. принадлежит языку L. □

4.1.2. Применение леммы о накачке

Рассмотрим несколько примеров использования леммы о накачке. В каждом примере эта лемма применяется для доказательства нерегулярности некоторого предлагаемого языка.

Лемма о накачке как игра двух противников

В разделе 1.2.3 говорилось о том, что любую теорему, утверждение которой содержит несколько чередующихся кванторов "для всех" ("для любого") и "существует", можно представить в виде игры двух противников. Лемма о накачке служит важным примером теорем такого типа, так как содержит четыре разных квантора: "**для любого** регулярного языка L **существует** n, при котором **для всех** w из L, удовлетворяющих неравенству $|w| \ge n$, **существует** цепочка xyz, равная w, удовлетворяющая …". Применение леммы о накачке можно представить в виде игры со следующими правилами.

1. Игрок 1 выбирает язык L, нерегулярность которого нужно доказать.
2. Игрок 2 выбирает n, но не открывает его игроку 1; первый игрок должен построить игру для всех возможных значений n.
3. Игрок 1 выбирает цепочку w, которая может зависеть от n, причем ее длина должна быть не меньше n.
4. Игрок 2 разбивает цепочку w на x, y и z, соблюдая условия леммы о накачке, т.е. $y \neq \varepsilon$ и $|xy| \leq n$. Опять-таки, он не обязан говорить первому игроку, чему равны x, y и z, хотя они должны удовлетворять условиям леммы.
5. Первый игрок "выигрывает", если выбирает k, которое может быть функцией от n, x, y и z и для которого цепочка xy^kz не принадлежит L.

Пример 4.2. Покажем, что язык L_{eq}, состоящий из всех цепочек с одинаковым числом нулей и единиц (расположенных в произвольном порядке), нерегулярен. В терминах игры, описанной во врезке "Лемма о накачке как игра двух противников", мы являемся игроком 1 и должны иметь дело с любыми допустимыми ходами игрока 2. Предположим, что n — это та константа, которая согласно лемме о накачке должна существовать, если язык L_{eq} регулярен, т.е. "игрок 2" выбирает n. Мы выбираем цепочку $w = 0^n1^n$, которая наверняка принадлежит L_{eq}.

Теперь "игрок 2" разбивает цепочку w на xyz. Нам известно лишь, что $y \neq \varepsilon$ и $|xy| \leq n$. Но эта информация очень полезна, и мы "выигрываем" следующим образом. Поскольку $|xy| \leq n$, и цепочка xy расположена в начале цепочки w, то она состоит только из нулей. Если язык L_{eq} регулярен, то по лемме о накачке цепочка xz принадлежит L_{eq} (при $k = 0$ в лемме)[2]. Цепочка xz содержит n единиц, так как все единицы цепочки w попадают в z. Но в xz нулей меньше n, так как потеряны все нули из y. Поскольку $y \neq \varepsilon$, то вместе x и z содержат не более $n - 1$ нулей. Таким образом, предположив, что язык L_{eq} регулярен, приходим к ошибочному выводу, что цепочка xz принадлежит L_{eq}. Следовательно, методом от противного доказано, что язык L_{eq} нерегулярен. □

Пример 4.3. Докажем нерегулярность языка L_{pr}, образованного всеми цепочками из единиц, длины которых — простые числа. Предположим, что язык L_{pr} регулярен. Тогда должна существовать константа n, удовлетворяющая условиям леммы о накачке. Рассмотрим некоторое простое число $p \geq n + 2$. Такое p должно существовать, поскольку множество простых чисел бесконечно. Пусть $w = 1^p$.

Согласно лемме о накачке можно разбить цепочку $w = xyz$ так, что $y \neq \varepsilon$ и $|xy| \leq n$. Пусть $|y| = m$. Тогда $|xz| = p - m$. Рассмотрим цепочку $xy^{p-m}z$, которая по лемме о накачке должна принадлежать языку L_{pr}, если он действительно регулярен. Однако

$$|xy^{p-m}z| = |xz| + (p - m)|y| = p - m + (p - m)m = (m + 1)(p - m).$$

[2] Заметим, что можно выиграть и при k = 2 или любом другом значении k, кроме k = 1.

Похоже, что число $|xy^{p-m}z|$ не простое, так как имеет два множителя $m + 1$ и $p - m$. Однако нужно еще убедиться, что ни один из этих множителей не равен 1, потому что тогда число $(m + 1)(p - m)$ будет простым. Из неравенства $y \neq \varepsilon$ следует $m \geq 1$ и $m + 1 > 1$. Кроме того, $m = |y| \leq |xy| \leq n$, а $p \geq n + 2$, поэтому $p - m \geq 2$.

Мы начали с предположения, что предлагаемый язык регулярен, и пришли к противоречию, доказав, что существует некоторая цепочка, которая не принадлежит этому языку, тогда как по лемме о накачке она должна ему принадлежать. Таким образом, язык L_{pr} нерегулярен. □

4.1.3. Упражнения к разделу 4.1

4.1.1. Докажите нерегулярность следующих языков:

а) $\{0^n1^n \mid n \geq 1\}$. Это язык L_{01}, который рассматривался в начале раздела. Ему принадлежат все цепочки, состоящие из нулей, за которыми следует такое же количество единиц. Здесь для доказательства примените лемму о накачке;

б) множество цепочек, состоящих из "сбалансированных" скобок "(" и ")", которые встречаются в правильно построенном арифметическом выражении;

в) (*) $\{0^n10^n \mid n \geq 1\}$;

г) $\{0^n1^m2^n \mid n$ и m — произвольные целые числа$\}$;

д) $\{0^n1^m \mid n \leq m\}$;

е) $\{0^n1^{2n} \mid n \geq 1\}$.

4.1.2. (!) Докажите нерегулярность следующих языков:

а) (*) $\{0^n \mid n$ — полный квадрат$\}$;

б) $\{0^n \mid n$ — полный куб$\}$;

в) $\{0^n \mid n$ — степень числа 2$\}$;

г) множество цепочек из нулей и единиц, длины которых — полные квадраты;

д) множество цепочек из нулей и единиц вида ww, где некоторая цепочка w повторяется дважды;

е) множество цепочек из нулей и единиц вида ww^R, где за цепочкой следует цепочка, обратная к ней (формальное определение цепочки, обратной к данной, см. в разделе 4.2.2);

ж) множество цепочек из нулей и единиц вида $w\overline{w}$, где цепочка $\overline{w}$ образована из w путем замены всех нулей единицами и наоборот. Например, $\overline{011} = 100$, так что цепочка 011100 принадлежит данному языку;

з) множество цепочек из нулей и единиц вида $w1^n$, где w — цепочка из нулей и единиц длиной n.

4.1.3. (!!) Докажите нерегулярность следующих языков:

а) множество всех цепочек из нулей и единиц, которые начинаются с единицы и удовлетворяют следующему условию: если интерпретировать такую цепочку как целое число, то это число простое;[3]

б) множество цепочек вида 0^i1^j, для которых наибольший общий делитель чисел i и j равен 1.

4.1.4. (!) При попытке применения леммы о накачке к регулярным языкам "противник выигрывает", и закончить доказательство не удается. Определите, что именно происходит не так, как нужно, если в качестве L выбрать следующий язык:

а) (*) пустое множество;

б) (*) {00, 11};

в) (*) $(\mathbf{00} + \mathbf{11})^*$;

г) $\mathbf{01}^*\mathbf{0}^*\mathbf{1}$.

4.2. Свойства замкнутости регулярных языков

В этом разделе доказано несколько теорем вида "если определенные языки регулярны, а язык L построен из них с помощью определенных операций (например, L есть объединение двух регулярных языков), то язык L также регулярен". Эти теоремы часто называют *свойствами замкнутости* регулярных языков, так как в них утверждается, что класс регулярных языков замкнут относительно определенных операций. Свойства замкнутости выражают идею того, что если один или несколько языков регулярны, то языки, определенным образом связанные с ним (с ними), также регулярны. Кроме того, данные свойства служат интересной иллюстрацией того, как эквивалентные представления регулярных языков (автоматы и регулярные выражения) подкрепляют друг друга в нашем понимании этого класса языков, так как часто один способ представления намного лучше других подходит для доказательства некоторого свойства замкнутости.

Основные свойства замкнутости регулярных языков выражаются в том, что эти языки замкнуты относительно следующих операций.

1. Объединение.
2. Пересечение.
3. Дополнение.
4. Разность.
5. Обращение.

[3] Иными словами, это двоичные записи простых чисел. — *Прим. ред.*

6. Итерация (звездочка).
7. Конкатенация.
8. Гомоморфизм (подстановка цепочек вместо символов языка).
9. Обратный гомоморфизм.

4.2.1. Замкнутость регулярных языков относительно булевых операций

Сначала рассмотрим замкнутость для трех булевых операций: объединение, пересечение и дополнение.

1. Пусть L и M — языки в алфавите Σ. Тогда язык $L \cup M$ содержит все цепочки, которые принадлежат хотя бы одному из языков L или M.
2. Пусть L и M — языки в алфавите Σ. Тогда язык $L \cap M$ содержит все цепочки, принадлежащие обоим языкам L и M.
3. Пусть L — некоторый язык в алфавите Σ. Тогда язык $\overline{L}$, *дополнение* L, — это множество тех цепочек в алфавите Σ^*, которые не принадлежат L.

Что делать, если языки имеют разные алфавиты?

При объединении или пересечении двух языков L и M может оказаться, что они определены в разных алфавитах. Например, возможен случай, когда $L_1 \subseteq \{a, b\}$, а $L_2 \subseteq \{b, c, d\}$. Однако, если язык L состоит из цепочек символов алфавита Σ, то L можно также рассматривать как язык в любом конечном алфавите, включающем Σ (надмножестве Σ). Например, можно представить указанные выше языки L_1 и L_2 как языки в алфавите $\{a, b, c, d\}$. То, что ни одна цепочка языка L_1 не содержит символов c или d, несущественно, как и то, что ни одна цепочка языка L_2 не содержит a.

Аналогично, рассматривая дополнение языка L, который является подмножеством множества Σ_1^* для некоторого алфавита Σ_1, можно взять дополнение *относительно* некоторого алфавита Σ_2, включающего Σ_1 (надмножества Σ_1). В этом случае дополнением L будет $\Sigma_2^* - L$, т.е. дополнение языка L относительно алфавита Σ_2 включает (среди прочих) все цепочки из Σ_2^*, которые содержат хотя бы один символ алфавита Σ_2, не принадлежащий Σ_1. Если взять дополнение L относительно Σ_1, то ни одна цепочка, содержащая символы из $\Sigma_2 - \Sigma_1$, не попадет в $\overline{L}$. Таким образом, чтобы избежать неточностей, нужно указывать алфавит, относительно которого берется дополнение. Часто, однако, бывает очевидно, какой алфавит подразумевается в конкретном случае. Например, если язык L определен некоторым автоматом, то в описании этого автомата указывается и алфавит. Итак, во многих ситуациях можно говорить о “дополнении”, не указывая алфавит.

Оказывается, что класс регулярных языков замкнут относительно всех трех булевых операций, хотя, как будет видно, в доказательствах используются совершенно разные подходы.

Замкнутость относительно объединения

Теорема 4.4. Если L и M — регулярные языки, то $L \cup M$ также регулярен.

Доказательство. Поскольку языки L и M регулярны, им соответствуют некоторые регулярные выражения. Пусть $L = L(R)$ и $M = L(S)$. Тогда $L \cup M = L(R + S)$ согласно определению операции + для регулярных выражений. □

Замкнутость относительно регулярных операций

Доказательство замкнутости регулярных выражений относительно объединения было исключительно легким, поскольку объединение является одной из трех операций, определяющих регулярные выражения. Идея доказательства теоремы 4.4 применима также к конкатенации и итерации. Таким образом,

- если L и M — регулярные языки, то язык LM регулярен;
- если L — регулярный язык, то L^* также регулярен.

Замкнутость относительно дополнения

Теорема для объединения языков легко доказывается с помощью регулярных выражений. Теперь рассмотрим дополнение. Знаете ли вы, как преобразовать регулярное выражение, чтобы оно определяло дополнение языка? Мы тоже не знаем. Однако это выполнимо, так как согласно теореме 4.5 можно начать с ДКА и построить ДКА, допускающий дополнение. Таким образом, начав с регулярного выражения для языка, можно найти выражение для его дополнения следующим образом.

1. Преобразовать регулярное выражение в ε-НКА.
2. Преобразовать ε-НКА в ДКА с помощью конструкции подмножеств.
3. Дополнить допускающие состояния этого ДКА.
4. Преобразовать полученный ДКА для дополнения обратно в регулярное выражение, используя методы из разделов 3.2.1 или 3.2.2.

Теорема 4.5. Если L — регулярный язык в алфавите Σ, то язык $\overline{L} = \Sigma^* - L$ также регулярен.

Доказательство. Пусть $L = L(A)$ для некоторого ДКА $A = (Q, \Sigma, \delta, q_0, F)$. Тогда $\overline{L} = L(B)$, где B — это ДКА $(Q, \Sigma, \delta, q_0, Q - F)$, т.е. автоматы A и B одинаковы, за исключением того, что допускающие состояния автомата A стали не допускающими в B, и наоборот. Тогда w принадлежит $L(B)$, если, и только если, $\hat{\delta}(q_0, w)$ принадлежит $Q - F$, т.е. w не принадлежит $L(A)$. □

Заметим, что для приведенного выше доказательства важно, чтобы $\hat{\delta}(q_0, w)$ всегда было некоторым состоянием. Это значит, что в автомате A все переходы определены. Если бы некоторые переходы отсутствовали, то определенные цепочки не вели бы ни в допускающее, ни в недопускающее состояния автомата A, т.е. отсутствовали бы как в $L(A)$, так и $L(B)$. К счастью, ДКА определен так, что в любом состоянии у него есть переход по каждому символу алфавита Σ, так что каждая цепочка приводит в состояние либо из F, либо из $Q - F$.

Пример 4.6. Пусть A — автомат, изображенный на рис. 2.14. Напомним, что этот ДКА допускает те и только те цепочки символов 0 и 1, которые заканчиваются на 01. В терминах регулярных выражений $L(A) = (\mathbf{0} + \mathbf{1})^*\mathbf{01}$. Таким образом, дополнение к $L(A)$ содержит все те цепочки из нулей и единиц, которые *не* заканчиваются на 01. На рис. 4.2 представлен автомат для $\{0, 1\}^* - L(A)$. Он совпадает с автоматом на рис. 2.14, за исключением того, что допускающие состояния стали недопускающими, а два недопускающих состояния стали допускающими. □

Пример 4.7. Используем теорему 4.5 для доказательства нерегулярности определенного языка. В примере 4.2 была доказана нерегулярность языка L_{eq}, состоящего из цепочек с равными количествами символов 0 и 1. Это доказательство было непосредственным применением леммы о накачке. Теперь рассмотрим язык M, состоящий из цепочек с неравными количествами нулей и единиц.

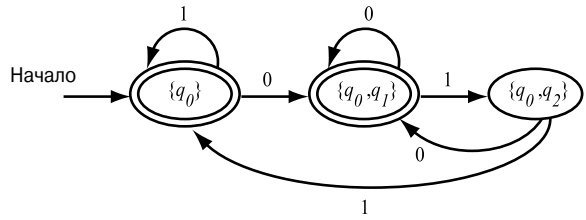


Рис. 4.2. ДКА, допускающий дополнение языка $(\mathbf{0} + \mathbf{1})^*\mathbf{01}$

В данном случае для доказательства нерегулярности языка M лемму о накачке применить трудно. Интуиция подсказывает, что если начать с некоторой цепочки w из M, разбить ее на $w = xyz$ и "накачать" y, то может оказаться, что y — это цепочка вроде 01, в которой поровну символов 0 и 1. В этом случае не существует такого k, для которого цепочка xy^kz имела бы поровну нулей и единиц, поскольку xyz содержит неравные количества нулей и единиц, а при "накачивании" y эти количества изменяются одинаково. Следовательно, мы не можем использовать лемму о накачке для того, чтобы прийти к противоречию с предположением, что язык M регулярен.

Но все-таки язык M нерегулярен. Объясняется это тем, что $M = \overline{L}_{eq}$. Поскольку дополнение к дополнению некоторого множества равно этому же множеству, то $L_{eq} = \overline{M}$.

Если M регулярен, то по теореме 4.5 язык L_{eq} также регулярен. Но мы знаем, что L_{eq} *не* регулярен, и полученное противоречие доказывает нерегулярность языка M. □

Замкнутость относительно пересечения

Рассмотрим пересечение двух регулярных языков. Здесь почти нечего делать, поскольку операции объединения, дополнения и пересечения не являются независимыми. Пересечение языков L и M выражается через объединение и дополнение следующим тождеством.

$$L \cap M = \overline{\overline{L} \cup \overline{M}} \tag{4.1}$$

Вообще, пересечение двух множеств — это множество элементов, не принадлежащих дополнениям каждого из них. Это замечание, записанное в виде равенства (4.1), представляет один из *законов Де Моргана*. Другой закон имеет аналогичный вид, только объединение и пересечение меняются местами, т.е. $L \cup M = \overline{\overline{L} \cap \overline{M}}$.

Вместе с тем, для пересечения двух регулярных языков можно построить ДКА непосредственно. Такая конструкция, в которой, по существу, параллельно работают два ДКА, весьма полезна сама по себе. Например, она использовалась для построения автомата (см. рис. 2.3), представляющего "произведение" действий двух участников — банка и магазина. *Конструкция произведения* формально представлена в следующей теореме.

Теорема 4.8. Если L и M — регулярные языки, то язык $L \cap M$ также регулярен.

Доказательство. Пусть L и M — языки автоматов $A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$ и $A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$. Заметим, что алфавиты автоматов считаются одинаковыми, т.е. Σ есть объединение алфавитов языков L и M, если эти алфавиты различаются. В действительности конструкция произведения работает для НКА точно так же, как и для ДКА, но для максимального упрощения предположим, что A_L и A_M — ДКА.

Для $L \cap M$ построим автомат A, моделирующий автоматы A_L и A_M одновременно. Состояниями автомата A будут пары состояний, первое из которых принадлежит A_L, а второе — A_M. Чтобы построить переходы автомата A, предположим, что он находится в состоянии (p, q), где p — состояние автомата A_L, а q — состояние A_M. Нам известно, как ведет себя автомат A_L, получая на входе символ a. Пусть он переходит в состояние s. Также допустим, что автомат A_M по входному символу a совершает переход в состояние t. Тогда следующим состоянием автомата A будет (s, t). Таким образом, автомат A моделирует работу автоматов A_L и A_M. Эта идея в общих чертах представлена на рис. 4.3.

Остальные детали доказательства очень просты. Начальным состоянием автомата A будет пара начальных состояний автоматов A_L и A_M. Поскольку автомат A допускает тогда и только тогда, когда допускают оба автомата A_L и A_M, в качестве допускающих состояний автомата A выбираем все пары (p, q), где p — допускающее состояние автомата A_L, а q —A_M. Формально автомат A определяется как

$$A = (Q_L \times Q_M, \Sigma, \delta, (q_L, q_M), (F_L \times F_M),$$

где $\delta((p, q), a) = (\delta_L(p, a), \delta_M(q, a))$.

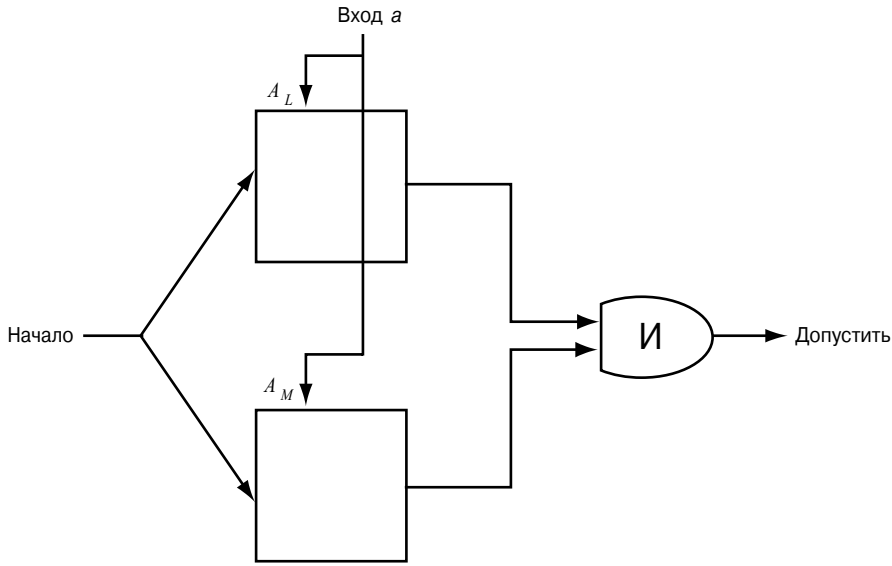


Рис. 4.3. Автомат, имитирующий два других автомата и допускающий тогда и только тогда, когда допускают оба автомата

Чтобы увидеть, почему $L(A) = L(A_L) \cap L(A_M)$, сначала заметим, что индукцией по $|w|$ легко доказать равенство $\hat{\delta}((q_L, q_M), w) = (\hat{\delta}_L(q_L, w), \hat{\delta}_M(q_M, w))$. Но A допускает w тогда и только тогда, когда $\hat{\delta}((q_L, q_M), w)$ является парой допускающих состояний, т.е. $\hat{\delta}_L(q_L, w)$ должно принадлежать F_L, а $\hat{\delta}_M(q_M, w)$ — F_M. Иными словами, цепочка w допускается автоматом A тогда и только тогда, когда ее допускают оба автомата A_L и A_M. Итак, A допускает пересечение языков L и M. □

Пример 4.9. На рис. 4.4 представлены два ДКА. Автомат на рис. 4.4, *а* допускает все цепочки, имеющие 0, а автомат на рис. 4.4, *б* — все цепочки, имеющие 1. На рис. 4.4, *в* представлен автомат — произведение двух данных автоматов. Его состояния помечены как пары состояний исходных автоматов.

Легко доказать, что этот автомат допускает пересечение первых двух языков, т.е. те цепочки, которые содержат как 0, так и 1. Состояние *pr* представляет начальное условие, когда на вход автомата пока не поступили ни 0, ни 1. Состояние *qr* означает, что поступили только нули, а состояние *ps* — только единицы. Допускающее состояние *qs* представляет условие того, что на вход автомата поступили и нули, и единицы. □

Замкнутость относительно разности

Существует еще одна, четвертая, операция, часто применяемая к множествам и связанная с булевыми операциями, а именно, разность множеств. В терминах языков *разностью* $L - M$ языков L и M называют множество цепочек, которые принадлежат L и не принадлежат M. Регулярные языки замкнуты относительно этой операции. Доказательство замкнутости относительно разности следует из доказанных выше теорем.

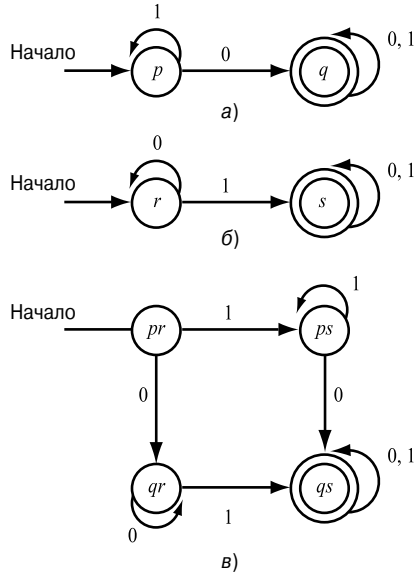


Рис. 4.4. Конструкция произведения

Теорема 4.10. Если L и M — регулярные языки, то язык $L - M$ также регулярен.

Доказательство. Заметим, что $L - M = L \cap \overline{M}$. По теореме 4.5 регулярен язык $\overline{M}$, а по теореме 4.8 — $L \cap \overline{M}$. Следовательно, язык $L - M$ регулярен. □

4.2.2. Обращение

Обращением цепочки $a_1a_2\ldots a_n$ называется цепочка, записанная в обратном порядке, т.е. $a_na_{n-1}\ldots a_1$. Обращение w обозначается w^R. Таким образом, 0010^R есть 0100, а $\varepsilon^R = \varepsilon$.

Обращение языка L, обозначаемое L^R, состоит из всех цепочек, обратных цепочкам языка L. Например, если $L = \{001, 10, 111\}$, то $L^R = \{100, 01, 111\}$.

Обращение является еще одной операцией, сохраняющей регулярность языков, т.е. если язык L регулярен, то L^R также регулярен. Это легко доказать двумя способами, первый из которых основан на автоматах, а второй — на регулярных выражениях. Доказательство, основанное на автоматах, приводится неформально, так что читатель при желании может восполнить детали. Затем приводится формальное доказательство, использующее регулярные выражения.

Если задан язык L, который есть $L(A)$ для некоторого конечного автомата A, вероятно, с недетерминизмом и ε-переходами, то можно построить автомат для L^R следующим образом.

1. Обратить все дуги на диаграмме переходов автомата A.
2. Сделать начальное состояние автомата A единственным допускающим состоянием нового автомата.

3. Создать новое начальное состояние p_0 с ε-переходами во все допускающие состояния автомата A.

В результате получим автомат, имитирующий автомат A "в обратном порядке" и, следовательно, допускающий цепочку w тогда и только тогда, кода A допускает w^R.

Теперь докажем теорему для обращения формально.

Теорема 4.11. Если язык L регулярен, то язык L^R также регулярен.

Доказательство. Предположим, что язык L определяется регулярным выражением E. Доказательство проводится структурной индукцией по длине выражения E. Покажем, что существует еще одно регулярное выражение E^R, для которого $L(E^R) = (L(E))^R$, т.е. язык выражения E^R является обращением языка выражения E.

Базис. Если E равно ε, $\varnothing$ или a, где a — некоторый символ, то E^R совпадает с E, т.е. $\{\varepsilon\}^R = \{\varepsilon\}$, $\varnothing^R = \varnothing$ и $\{a\}^R = \{a\}$.

Индукция. В зависимости от вида выражения E возможны три варианта.

1. $E = E_1 + E_2$. Тогда $E^R = E_1^R + E_2^R$. Доказательство состоит в том, что обращение объединения двух языков получается, если сначала вычислить, а затем объединить обращения этих языков.
2. $E = E_1E_2$. Тогда $E^R = E_2^R E_1^R$. Заметим, что необходимо обратить не только сами языки, но и их порядок. Например, если $L(E_1) = \{01, 111\}$, а $L(E_2) = \{00, 10\}$, то $L(E_1E_2) = \{0100, 0110, 11100, 11110\}$. Обращение этого языка есть $\{0010, 0110, 00111, 01111\}$.

   Если соединить обращения языков $L(E_2)$ и $L(E_1)$ в таком порядке, как они здесь записаны, то получим язык

   $$\{00, 01\}\{10, 111\} = \{0010, 00111, 0110, 01111\},$$

   который равен языку $(L(E_1E_2))^R$. В общем случае, если цепочка w из $L(E)$ является конкатенацией цепочек w_1 из $L(E_1)$ и w_2 из $L(E_2)$, то $w^R = w_2^R w_1^R$.
3. $E = E_1^*$. Тогда $E^R = (E_1^R)^*$. Доказательство состоит в том, что любая цепочка w из $L(E)$ может быть записана как $w_1w_2\ldots w_n$, где каждая w_i принадлежит $L(E)$. Но

   $$w^R = w_n^R w_{n-1}^R \ldots w_1^R.$$

   Каждая w_i^R принадлежит $L(E^R)$, т.е. w^R принадлежит $(E_1^R)^*$. И наоборот, любая цепочка из $L((E_1^R)^*)$ имеет вид $w_1w_2\ldots w_n$, где каждая цепочка w_i является обращением некоторой цепочки из $L(E_1)$. Следовательно, обращение данной цепочки $w_n^R w_{n-1}^R \ldots w_1^R$ принадлежит языку $L(E_1^*)$, который равен $L(E)$. Таким образом, доказано, что цепочка принадлежит $L(E)$ тогда и только тогда, когда ее обращение принадлежит $L((E_1^R)^*)$.

□

Пример 4.12. Пусть язык L определяется регулярным выражением $(\mathbf{0} + \mathbf{1})\mathbf{0}^*$. Тогда согласно правилу для конкатенации L^R — это язык выражения $(\mathbf{0}^*)^R(\mathbf{0} + \mathbf{1})^R$. Если приме-

нить правила для итерации и объединения к двум частям этого выражения, а потом использовать базисное правило, которое говорит, что обратными к **0** и **1** будут эти же выражения, то получим, что язык L^R определяется регулярным выражением $\mathbf{0}^*(\mathbf{0} + \mathbf{1})$. □

4.2.3. Гомоморфизмы

Гомоморфизм цепочек — это такая функция на множестве цепочек, которая подставляет определенную цепочку вместо каждого символа данной цепочки.

Пример 4.13. Функция h, определенная как $h(0) = ab$ и $h(1) = \varepsilon$, является гомоморфизмом. В любой цепочке из символов 0 и 1 h заменяет все нули цепочкой ab, а все единицы — пустой цепочкой. Например, применяя h к цепочке 0011, получим $abab$. □

Формально, если h есть некоторый гомоморфизм на алфавите Σ, а $w = a_1a_2\ldots a_n$ — цепочка символов в Σ, то $h(w) = h(a_1)h(a_2)\ldots h(a_n)$. Таким образом, сначала h применяется к каждому символу цепочки w, а потом полученные цепочки символов соединяются в соответствующем порядке. Например, рассмотрим гомоморфизм h из примера 4.13 и цепочку $w = 0011$: $h(w) = h(0)h(0)h(1)h(1) = (ab)(ab)(\varepsilon)(\varepsilon) = abab$, что и утверждается в этом примере.

Гомоморфизм языка определяется с помощью его применения к каждой цепочке языка, т.е. если L — язык в алфавите Σ, а h — гомоморфизм на Σ, то $h(L) = \{h(w) \mid w$ принадлежит $L\}$. Рассмотрим язык L регулярного выражения $\mathbf{10}^*\mathbf{1}$, т.е. все цепочки, которые начинаются и заканчиваются единицей, а между ними содержат произвольное число нулей. Пусть h — гомоморфизм из примера 4.13. Тогда $h(L)$ — это язык выражения $(\mathbf{ab})^*$. Объясняется это тем, что h исключает все единицы, заменяя их ε, а вместо каждого нуля подставляет цепочку ab. Идея применения гомоморфизма непосредственно к регулярному выражению используется для доказательства замкнутости регулярных языков относительно гомоморфизма.

Теорема 4.14. Если L — регулярный язык в алфавите Σ, а h — гомоморфизм на Σ, то язык $h(L)$ также регулярен.

Доказательство. Пусть $L = L(R)$ для некоторого регулярного выражения R. Вообще, если E есть регулярное выражение с символами из алфавита Σ, то пусть $h(E)$ — выражение, полученное в результате замены каждого символа a в выражении E цепочкой $h(a)$. Утверждается, что выражение $h(R)$ определяет язык $h(L)$.

Это легко доказать с помощью структурной индукции. Если применить гомоморфизм h к любому подвыражению E выражения R, то язык выражения $h(E)$ совпадет с языком, полученным в результате применения этого гомоморфизма к языку $L(E)$. Формально, $L(h(E)) = h(L(E))$.

Базис. Если E есть ε или $\varnothing$, то $h(E)$ совпадает с E, поскольку h не влияет на цепочку ε или язык $\varnothing$. Следовательно, $L(h(E)) = L(E)$. В то же время, если E равно $\varnothing$ или ε, то $L(E)$ либо не содержит ни одной цепочки, либо состоит из цепочки без символов. Таким образом, в обоих случаях $h(L(E)) = L(E)$. Из этого следует, что $L(h(E)) = L(E) = h(L(E))$.

Возможен еще один базисный вариант, когда $E = \mathbf{a}$ для некоторого символа a из Σ. В этом случае $L(E) = \{a\}$, и $h(L(E)) = \{h(a)\}$. Выражение $h(E)$ представляет собой цепочку символов $h(a)$. Таким образом, язык $L(h(E))$ также совпадает с $\{h(a)\}$, и, следовательно, $L(h(E)) = h(L(E))$.

Индукция. В зависимости от операции в регулярном выражении возможны три ситуации. Все они просты, поэтому обоснуем индукцию только для объединения, $E = F + G$. Способ применения гомоморфизмов к регулярным выражениям гарантирует, что $h(E) = h(F + G) = h(F) + h(G)$. Нам также известно, что $L(E) = L(F) \cup L(G)$ и

$$L(h(E)) = L(h(F) + h(G)) = L(h(F)) \cup L(h(G)) \tag{4.2}$$

по определению операции + для регулярных выражений. Наконец,

$$h(L(E)) = h(L(F) \cup L(G)) = h(L(F)) \cup h(L(G)), \tag{4.3}$$

поскольку h применяется к языку путем применения его к каждой цепочке этого языка по отдельности. По индуктивной гипотезе $L(h(F)) = h(L(F))$ и $L(h(G)) = h(L(G))$. Таким образом, правые части выражений (4.2) и (4.3) эквивалентны, и, следовательно, $L(h(E)) = h(L(E))$.

Для случаев, когда выражение E является конкатенацией или итерацией, доказательства не приводятся, поскольку они аналогичны доказательству, представленному выше. Итак, можно сделать вывод, что $L(h(R))$ действительно равняется $h(L(R))$, т.е. применение гомоморфизма к регулярному выражению языка L дает регулярное выражение, определяющее язык $h(L)$. □

4.2.4. Обратный гомоморфизм

Гомоморфизм можно применять “назад”, и это также сохраняет регулярность языков. Предположим, что h — гомоморфизм из алфавита Σ в цепочки, заданные в другом (возможно, том же) алфавите T[4]. Пусть L — язык в алфавите T. Тогда $h^{-1}(L)$, читаемое как “обратное h от L”, — это множество цепочек w из Σ^*, для которых $h(w)$ принадлежит L. На рис. 4.5, *а* представлено применение гомоморфизма к языку L, а на рис. 4.5, *б* — использование обратного гомоморфизма.

Пример 4.15. Пусть L — язык регулярного выражения $(\mathbf{00} + \mathbf{1})^*$, т.е. все цепочки из символов 0 и 1, в которых нули встречаются парами. Таким образом, цепочки 0010011 и 10000111 принадлежат L, а 000 и 10100 — нет.

Пусть h — такой гомоморфизм: $h(a) = 01$, $h(b) = 10$. Утверждается, что $h^{-1}(L)$ — это язык регулярного выражения $(\mathbf{ba})^*$, т.е. все цепочки, в которых повторяются пары ba. Докажем, что $h(w)$ принадлежит L тогда и только тогда, когда цепочка w имеет вид $baba\ldots ba$.

[4] Под “T” подразумевается прописная буква греческого алфавита “тау”, следующая за буквой “сигма”.

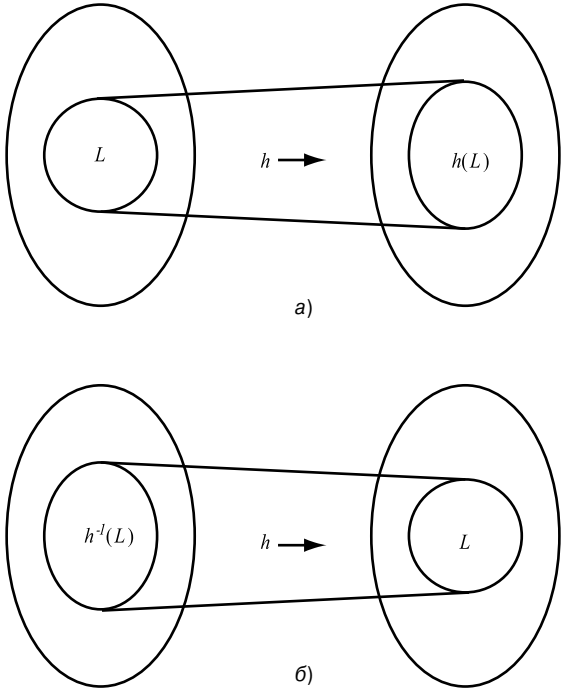


Рис. 4.5. Гомоморфизм, применяемый в прямом и обратном направлении

Достаточность. Предположим, что цепочка w состоит из n повторений ba для некоторого $n \geq 0$. Заметим, что $h(ba) = 1001$, т.е. $h(w)$ — это n повторений цепочки 1001. Поскольку цепочка 1001 построена из двух единиц и пары нулей, то она принадлежит языку L. Следовательно, цепочка, состоящая из любого числа повторений 1001, также образована единицами и парами нулей и принадлежит L. Таким образом, $h(w)$ принадлежит L.

Необходимость. Теперь предположим, что $h(w)$ принадлежит L, и покажем, что цепочка w имеет вид $baba…ba$. Существует четыре условия, при которых цепочка имеет *другой* вид. Покажем, что при выполнении любого из них $h(w)$ не принадлежит L, т.е. докажем утверждение, противоположное тому, что нам нужно доказать.

1. Если w начинается символом a, то $h(w)$ начинается с 01. Следовательно, она содержит отдельный 0 и поэтому не принадлежит L.
2. Если w заканчивается символом b, то в конце $h(w)$ стоит 10, и опять-таки в цепочке $h(w)$ есть изолированный 0.
3. Если в цепочке w дважды подряд встречается a, то $h(w)$ содержит подцепочку 0101. Снова в w есть изолированный нуль.
4. Аналогично, если в w есть два символа b подряд, то $h(w)$ содержит подцепочку 1010 с изолированным 0.

Таким образом, при выполнении хотя бы одного из вышеперечисленных условий цепочка $h(w)$ не принадлежит L. Но если ни одно из условий 1–4 не выполняется, то цепочка w имеет вид $baba…ba$. Чтобы понять, почему это происходит, предположим, что ни одно из этих условий не выполняется. Тогда невыполнение условия 1 означает, что w должна начинаться символом b, а невыполнение 2 — что она должна заканчиваться символом a. Невыполнение условий 3 и 4 говорит, что символы a и b должны чередоваться. Следовательно, логическое **ИЛИ** условий 1–4 эквивалентно утверждению "цепочка w имеет вид, отличный от $baba…ba$". Но выше было доказано, что из логического **ИЛИ** условий 1–4 следует, что $h(w)$ не принадлежит L. Это утверждение противоположно тому, что нужно доказать, а именно, что "если $h(w)$ принадлежит L, то цепочка w имеет вид $baba…ba$". □

Далее докажем, что обратный гомоморфизм регулярного языка также регулярен, и покажем, как эту теорему можно использовать.

Теорема 4.16. Если h — гомоморфизм из алфавита Σ в алфавит T, L — регулярный язык над T, то язык $h^{-1}(L)$ также регулярен.

Доказательство. Начнем с ДКА A для языка L. По A и h строится ДКА для $h^{-1}(L)$ с помощью схемы, представленной на рис. 4.6. Этот ДКА использует состояния автомата A, но переводит входной символ в соответствии с h перед тем, как решить, в какое состояние перейти.

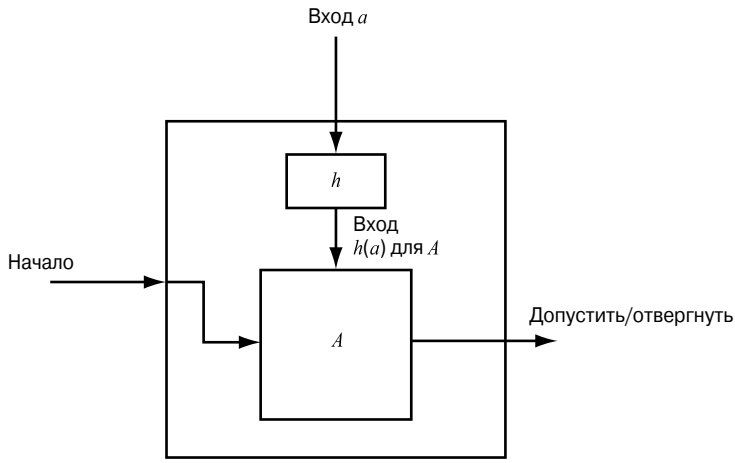


Рис. 4.6. ДКА для $h^{-1}(L)$ *применяет гомоморфизм* h *ко входным символам, а потом имитирует ДКА для* L

Формально, пусть L — это $L(A)$, где ДКА $A = (Q, T, \delta, q_0, F)$. Определим ДКА

$$B = (Q, \Sigma, \gamma, q_0, F),$$

функция переходов γ которого строится по правилу $\gamma(q, a) = \hat{\delta}(q, h(a))$. Таким образом, переход автомата B по входному символу a является результатом последовательности переходов, совершаемых автоматом A при получении цепочки символов $h(a)$. Напомним,

что, хотя $h(a)$ может равняться ε, состоять из одного или нескольких символов, функция $\hat{\delta}$ определена так, чтобы справиться со всеми этими случаями.

С помощью индукции по $|w|$ легко показать, что $\hat{\gamma}(q_0, w) = \hat{\delta}(q_0, h(w))$. Поскольку допускающие состояния автоматов A и B совпадают, то B допускает цепочку w тогда и только тогда, когда A допускает цепочку $h(w)$. Иными словами, B допускает только те цепочки, которые принадлежат языку $h^{-1}(L)$. □

Пример 4.17. В этом примере обратный гомоморфизм и некоторые другие свойства замкнутости регулярных множеств используются для доказательства одного необычного свойства конечных автоматов. Предположим, что, допуская входную цепочку, некоторый автомат должен побывать в каждом состоянии хотя бы по одному разу. Точнее, допустим, что $A = (Q, \Sigma, \delta, q_0, F)$ — ДКА, и нас интересует язык L, состоящий из всех цепочек w в алфавите Σ^*, для которых $\hat{\delta}(q_0, w)$ принадлежит F, и для каждого состояния q из Q существует некоторый префикс x_q цепочки w, для которого $\hat{\delta}(q_0, x_q) = q$. Будет ли язык L регулярным? Докажем, что такой язык регулярен, хотя доказательство довольно сложное.

Начнем с языка $M = L(A)$, т.е. множества цепочек, допускаемых автоматом A обычным путем, независимо от того, в какие состояния он переходит, обрабатывая входную цепочку. Заметим, что $L \subseteq M$, так как определение языка L накладывает дополнительные ограничения на цепочки из $L(A)$. Доказательство регулярности языка L начинается с использования обратного гомоморфизма для вставки состояний автомата A во входные символы. Точнее, определим новый алфавит T как состоящий из символов, которые можно представить в виде троек $[paq]$, где p и q — состояния из Q, a — символ из Σ, и $\delta(p, a) = q$.

Таким образом, символы алфавита T представляют переходы автомата A. Важно понимать, что запись $[paq]$ представляет собой единый символ, а не конкатенацию трех символов. Можно обозначить этот символ одной буквой, но при этом трудно описать его связь с p, q и a.

Теперь определим гомоморфизм $h([paq]) = a$ для всех p, a и q. Это значит, что гомоморфизм h удаляет из каждого символа алфавита T компоненты, представляющие состояния, и оставляет только символ из Σ. Первый шаг доказательства регулярности языка L состоит в построении языка $L_1 = h^{-1}(M)$. Поскольку язык M регулярен, то согласно теореме 4.16 язык L_1 также регулярен. Цепочками языка L_1 будут цепочки из M, к каждому символу которых присоединяется пара состояний, представляющая некоторый переход автомата.

В качестве простого примера рассмотрим автомат с двумя состояниями, представленный на рис. 4.4, *а*. Алфавит $\Sigma = \{0, 1\}$, а алфавит T состоит из четырех символов $[p0q]$, $[q0q]$, $[p1p]$ и $[q1q]$. Например, поскольку по символу 0 есть переход из p в q, то $[p0q]$ — один из символов алфавита T. Так как цепочка 101 допускается этим автоматом, применив к ней обратный гомоморфизм h^{-1}, получим $2^3 = 8$ цепочек, две из которых, например, равны $[p1p][p0q][q1q]$ и $[q1q][q0q][p1p]$.

Теперь по языку L_1 построим язык L с помощью ряда операций, сохраняющих регулярность языков. Наша первая цель — исключить все те цепочки языка L_1, в которых состояния указаны неправильно. Поскольку каждый символ вида $[paq]$ означает, что автомат был в состоянии p, прочитал a и затем перешел в состояние q, последовательность таких символов, представляющая допускающее вычисление в автомате A, должна удовлетворять следующим трем условиям.

1. Первым состоянием в первом символе должно быть q_0 — начальное состояние A.
2. Каждый переход автомата должен начинаться там, где закончился предыдущий, т.е. первое состояние в символе должно равняться второму состоянию в предыдущем символе.
3. Второе состояние в последнем символе должно принадлежать F. Если выполняются первые два условия, то и это условие будет выполнено, поскольку каждая цепочка языка L_1 образована из цепочки, допускаемой автоматом A.

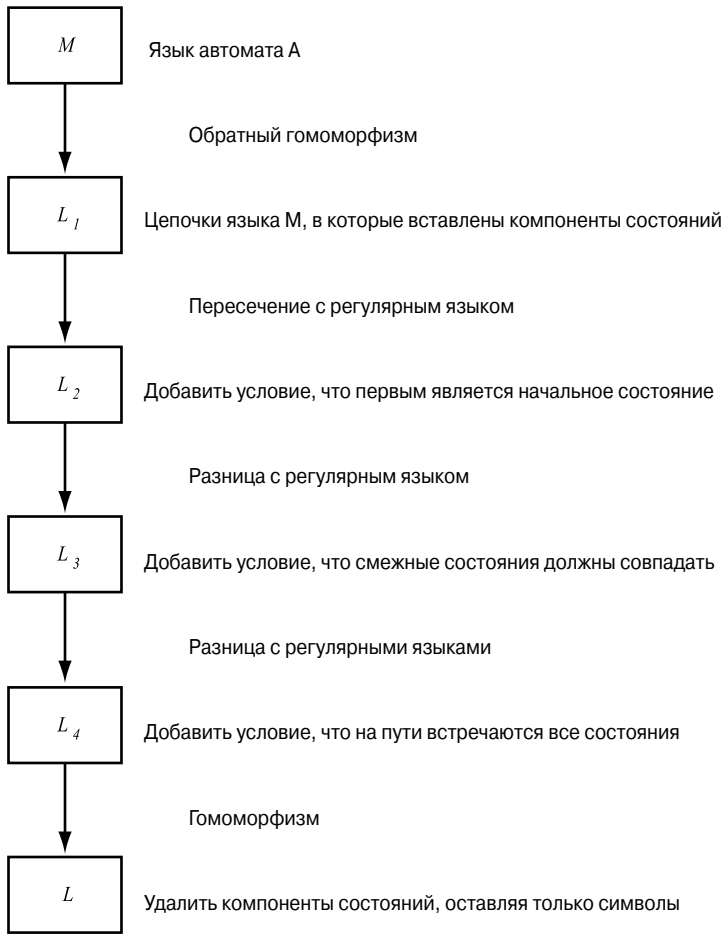


Рис. 4.7. Построение языка L *по языку* M *с помощью операций, сохраняющих регулярность языков*

План построения языка L представлен на рис. 4.7.

Условие 1 обеспечивается пересечением языка L_1 со множеством цепочек, которые начинаются символом вида $[q_0aq]$ для некоторого символа a и состояния q. Пусть E_1 — выражение $[q_0a_1q_1] + [q_0a_2q_2] + \ldots$, где a_iq_i — все пары из $\Sigma \times Q$, для которых $\delta(q_0, a_i) = q_i$. Пусть $L_2 = L_1 \cap L(E_1T^*)$. Регулярное выражение E_1T^* обозначает все цепочки из T^*, которые начинаются стартовым состоянием (здесь T можно рассматривать как сумму всех его символов). Поэтому язык L_2 состоит из всех цепочек, полученных в результате применения обратного гомоморфизма h^{-1} к языку M, у которых первым компонентом в первом символе является начальное состояние, т.е. язык L_2 удовлетворяет условию 1.

Чтобы обеспечить выполнение условия 2, проще всего вычесть из L_2 (используя операцию разности множеств) все цепочки, нарушающие это условие. Пусть E_2 — регулярное выражение, состоящее из суммы (объединения) конкатенаций всех пар символов, которые друг другу не подходят. Это все пары вида $[paq][rbs]$, где $q \neq r$. Тогда регулярное выражение $T^*E_2T^*$ обозначает все цепочки, не удовлетворяющие условию 2.

Теперь можно определить $L_3 = L_2 - L(T^*E_2T^*)$. Цепочки языка L_3 удовлетворяют условию 1, поскольку цепочки языка L_2 начинаются стартовым состоянием. Они также удовлетворяют условию 2, так как в результате вычитания $L(T^*E_2T^*)$ будут удалены все цепочки, для которых это условие не выполняется. Наконец, они удовлетворяют условию 3 (последнее состояние является допускающим), поскольку доказательство было начато с цепочек языка M, допускаемых автоматом A. В результате L_3 состоит из цепочек языка M с состояниями допускающего вычисления такой цепочки, вставленными в каждый символ. Заметим, что язык L_3 регулярен, так как он построен из регулярного языка M с помощью операций обратного гомоморфизма, пересечения и разности множеств, сохраняющих регулярность.

Напомним, что наша цель состоит в том, чтобы допустить только те цепочки из M, при обработке которых автомат проходит через каждое состояние. Выполнение этого условия можно обеспечить с помощью операции разности множеств. Пусть для каждого состояния q регулярное выражение E_q представляет собой сумму всех символов алфавита T, в которые не входит состояние q (q не стоит ни на первой, ни на последней позиции). В результате вычитания языка $L(E_q^*)$ из L_3 получим цепочки, представляющие допускающее вычисление автомата A и проходящие через состояние q, по крайней мере, один раз. Если вычесть из L_3 языки $L(E_q^*)$ для всех q из Q, то получим допускающие вычисления автомата A, проходящие через все состояния. Обозначим этот язык L_4. По теореме 4.10 язык L_4 также регулярен.

Последний шаг состоит в построении языка L из L_4 с помощью исключения компонентов состояний, т.е. $L = h(L_4)$. Теперь L является множеством цепочек в алфавите Σ^*, допускаемых автоматом A, причем при их обработке автомат проходит через каждое состояние, по крайней мере, один раз. Поскольку регулярные языки замкнуты относительно гомоморфизмов, делаем вывод, что язык L регулярен. □

4.2.5. Упражнения к разделу 4.2

4.2.1. Пусть h — гомоморфизм из алфавита $\{0, 1, 2\}$ в алфавит $\{a, b\}$, определенный как $h(0) = a$, $h(1) = ab$ и $h(2) = ba$:

а) (*****) найдите $h(0120)$;

б) найдите $h(21120)$;

в) (*****) найдите $h(L)$ для $L = L(\mathbf{01}^*\mathbf{2})$;

г) найдите $h(L)$ для $L = L(\mathbf{0} + \mathbf{12})$;

д) (*****) найдите $h^{-1}(L)$ для $L = \{ababa\}$, т.е. языка, состоящего из одной-единственной цепочки $ababa$;

е) (**!**) найдите $h^{-1}(L)$ для $L = L(\mathbf{a}(\mathbf{ba})^*)$.

4.2.2. (***!**) Если L — язык, a — символ, то L/a, *частное* L и a, — это множество цепочек w, для которых wa принадлежит L. Например, если $L = \{a, aab, baa\}$, то $L/a = \{\varepsilon, ba\}$. Докажите, что из регулярности L следует регулярность L/a. *Указание*. Начните с ДКА для L и рассмотрите множество допускающих состояний.

4.2.3. (**!**) Если L — язык, a — символ, то $a\backslash L$ — это множество цепочек w, для которых aw принадлежит L. Например, если $L = \{a, aab, baa\}$, то $a\backslash L = \{\varepsilon, ab\}$. Докажите, что из регулярности L следует регулярность $a\backslash L$. *Указание.* Вспомните, что регулярные языки замкнуты относительно обращения и операции деления из упражнения 4.2.2.

4.2.4. (**!**) Какие из следующих тождеств истинны?

а) $(L/a)a = L$ (в левой части представлена конкатенация языков L/a и $\{a\}$).

б) $a(a\backslash L) = L$ (снова представлена конкатенация с языком $\{a\}$, но на этот раз слева).

в) $(La)/a = L$.

г) $a\backslash(aL) = L$.

4.2.5. Операция из упражнения 4.2.3 иногда рассматривается как “производная”, а выражение $a\backslash L$ записывается как $\frac{dL}{da}$. Эти производные применяются к регулярным выражениям аналогично тому, как обычные производные применяются к арифметическим выражениям. Таким образом, если R — регулярное выражение, то $\frac{dR}{da}$ обозначает то же, что и $\frac{dL}{da}$, если $L = L(R)$:

а) докажите, что $\frac{d(R+S)}{da} = \frac{dR}{da} + \frac{dS}{da}$;

б) (***!**) напишите правило для “производной” от RS. *Указание*. Нужно рассмотреть два случая: язык $L(R)$ включает или не включает цепочку ε. Это правило

не совпадает с “правилом произведения” для обычных производных, но похоже на него;

в) (**!**) найдите “производную” от итерации, т.е. $\frac{d(R^*)}{da}$;

г) используя формулы (а)–(в), найдите производную регулярного выражения $(\mathbf{0} + \mathbf{1})^*\mathbf{011}$;

д) (*****) опишите такие языки L, для которых $\frac{dL}{d0} = \varnothing$;

е) (***!**) опишите такие языки L, для которых $\frac{dL}{d0} = L$.

4.2.6. (**!**) Докажите замкнутость регулярных языков относительно следующих операций:

а) $min(L) = \{w \mid w$ принадлежит L, но ни один собственный префикс цепочки w не принадлежит $L\}$;

б) $max(L) = \{w \mid w$ принадлежит L, но для любого $x \neq \varepsilon$ цепочка wx не принадлежит $L\}$;

в) $init(L) = \{w \mid$ для некоторого x цепочка wx принадлежит $L\}$.

Указание. Как и в упражнении 4.2.2, проще всего начать с ДКА для L и преобразовать его для получения нужного языка.

4.2.7. (**!**) Если $w = a_1a_2\ldots a_n$ и $x = b_1b_2\ldots b_n$ — цепочки одинаковой длины, то определим $alt(w, x)$ как цепочку, в которой символы цепочек w и x чередуются, начиная с w, т.е. $a_1b_1a_2b_2\ldots a_nb_n$. Если L и M — языки, определим $alt(L, M)$ как множество цепочек вида $alt(w, x)$, где w — произвольная цепочка из L, а x — любая цепочка из M такой же длины. Докажите, что из регулярности языков L и M следует регулярность языка $alt(L, M)$.

4.2.8. (***!!**) Пусть L — язык. Определим $half(L)$ как множество первых половин цепочек языка L, т.е. множество $\{w \mid$ существует x, для которой wx принадлежит L, причем $|x| = |w|\}$. Например, если $L = \{\varepsilon, 0010, 011, 010110\}$, то $half(L) = \{\varepsilon, 00, 010\}$. Заметим, что цепочки с нечетной длиной не влияют на $half(L)$. Докажите, что если язык L регулярен, то $half(L)$ также регулярен.

4.2.9. (**!!**) Упражнение 4.2.8 можно распространить на многие функции, определяющие длину части цепочки. Если f — функция, определенная на множестве целых чисел, то обозначим через $f(L)$ множество цепочек $\{w \mid$ существует цепочка x, для которой $|x| = f(|w|)$ и wx принадлежит $L\}$. Например, операция $half$ соответствует тождественной функции $f(n) = n$, так как для цепочек из языка $half(L)$ выполняется $|x| = |w|$. Покажите, что если язык L регулярен, то язык $f(L)$ также регулярен, где f — одна из следующих функций:

а) $f(n) = 2n$ (т.е. используются первые трети цепочек);

б) $f(n) = n^2$ (длина выбираемой части цепочки равна квадратному корню длины оставшейся части цепочки);

в) $f(n) = 2^n$ (длина выбираемой части цепочки равна логарифму длины ее остатка).

4.2.10. (!!) Пусть L — язык, не обязательно регулярный, в алфавите {0}, т.е. цепочки языка L состоят из одних нулей. Докажите, что язык L^* регулярен. *Указание.* На первый взгляд эта теорема кажется абсурдной. Чтобы проиллюстрировать истинность утверждения теоремы, приведем один небольшой пример. Рассмотрим язык $L = \{0^i \mid i$ — простое число$\}$, который, как известно, нерегулярен (см. пример 4.3). Нетрудно доказать, что если $j \geq 2$, то 0^j принадлежит L^*. Поскольку числа 2 и 3 — простые, цепочки 00 и 000 принадлежат L. Если j — четное число, то 0^j можно получить, повторив $j/2$ раз цепочку 00, а если j — нечетное, можно взять одну цепочку 000 и $(j - 3)/2$ цепочек 00. Следовательно, $L^* = \varepsilon + \mathbf{000}^*$.

4.2.11. (!!) Докажите замкнутость регулярных языков относительно следующей операции: $cycle(L) = \{w \mid$ цепочку w можно представить в виде $w = xy$, где yx принадлежит $L\}$. Например, если $L = \{01, 011\}$, то $cycle(L) = \{01, 10, 011, 110, 101\}$. *Указание.* Начните с ДКА для языка L и постройте ε-НКА для $cycle(L)$.

4.2.12. (!!) Пусть $w_1 = a_0a_0a_1$, а $w_i = w_{i-1}w_{i-1}a_i$ для всех $i > 0$. Например, $w_3 = a_0a_0a_1a_0a_0a_1a_2a_0a_0a_1a_0a_0a_1a_2a_3$. Кратчайшим регулярным выражением для языка $L_n = \{w_n\}$, т.е. языка, состоящего из цепочки w_n, будет сама w_n, причем длина этого выражения равна $2^{n+1} - 1$. Однако, если применить операцию пересечения, то для языка L_n можно записать выражение длиной $O(n^2)$. Найдите такое выражение. *Указание.* Найдите n языков с регулярными выражениями длины $O(n)$, пересечение которых равно L_n.

4.2.13. Свойства замкнутости можно использовать для доказательства нерегулярности некоторых языков. Докажите, что язык

$L_{0n1n} = \{0^n1^n \mid n \geq 0\}$

нерегулярен. Докажите нерегулярность следующих языков, преобразовав их с помощью операций, сохраняющих регулярность, в язык L_{0n1n}:

а) (*) $\{0^i0^j \mid i \neq j\}$;

б) $\{0^n1^m2^{n-m} \mid n \geq m \geq 0\}$.

4.2.14. В теореме 4.8 представлена “конструкция произведения”, в которой по двум данным ДКА построен ДКА, допускающий пересечение языков данных автоматов:

а) покажите, как построить конструкцию произведения для НКА (без ε-переходов);

б) (!) продемонстрируйте, как построить конструкцию произведения для ε-НКА;

в) (*) покажите, как изменить конструкцию для произведения так, чтобы результирующий ДКА допускал разность языков двух данных ДКА;

г) измените конструкцию для произведения так, чтобы результирующий ДКА допускал объединение языков двух данных ДКА.

4.2.15. В доказательстве теоремы 4.8 утверждалось, что с помощью индукции по длине цепочки w можно доказать следующее равенство:

$\hat{\delta}((q_L, q_M), w) = (\hat{\delta}_L(q_L, w), \hat{\delta}_M(q_M, w))$.

Приведите это доказательство.

4.2.16. Завершите доказательство теоремы 4.14, рассмотрев случаи, когда выражение E является конкатенацией двух подвыражений или итерацией некоторого выражения.

4.2.17. В теореме 4.16 пропущено доказательство индукцией по длине цепочки w того, что $\tilde{\gamma}(q_0, w) = \hat{\delta}(q_0, h(w))$. Восполните этот пробел.

4.3. Свойства разрешимости регулярных языков

В этом разделе мы сформируем важные вопросы, связанные с регулярными языками. Сначала нужно разобраться, что значит задать вопрос о некотором языке. Типичный язык бесконечен, поэтому бессмысленно предъявлять кому-нибудь цепочки этого языка и задавать вопрос, требующий проверки бесконечного множества цепочек. Гораздо разумнее использовать одно из конечных представлений языка, а именно: ДКА, НКА, ε-НКА или регулярное выражение.

Очевидно, что представленные таким образом языки будут регулярными. В действительности не существует способа представления абсолютно произвольных языков. В следующих главах предлагаются конечные методы описания более широких классов, чем класс регулярных языков, и можно будет рассматривать вопросы о языках из них. Однако алгоритмы разрешения многих вопросов о языках существуют только для класса регулярных языков. Эти же вопросы становятся “неразрешимыми” (не существует алгоритмов ответов на эти вопросы), если они поставлены с помощью более “выразительных” обозначений (используемых для выражения более широкого множества языков), чем представления, разработанные для регулярных языков.

Начнем изучение алгоритмов для вопросов о регулярных языках, рассмотрев способы, которыми одно представление языка преобразуется в другое. В частности, рассмотрим временну*ю* сложность алгоритмов, выполняющих преобразования. Затем рассмотрим три основных вопроса о языках.

1. Является ли описываемый язык пустым множеством?
2. Принадлежит ли некоторая цепочка w представленному языку?
3. Действительно ли два разных описания представляют один и тот же язык? (Этот вопрос часто называют “эквивалентностью” языков.)

4.3.1. Преобразования различных представлений языков

Из главы 3 известно, что каждое из четырех представлений регулярных языков можно преобразовать в любое из остальных трех. На рис. 3.1 представлены переходы от одного представления к другому. Хотя существуют алгоритмы для любого из этих преобразований, иногда нас интересует не только осуществимость некоторого преобразования, но и время, необходимое для его выполнения. В частности, важно отличать алгоритмы, которые занимают экспоненциальное время (время как функция от размера входных данных) и, следовательно, могут быть выполнены только для входных данных сравнительно небольших размеров, от тех алгоритмов, время выполнения которых является линейной, квадратичной или полиномиальной с малой степенью функцией от размера входных данных. Последние алгоритмы "реалистичны", так как их можно выполнить для гораздо более широкого класса экземпляров задачи. Рассмотрим временную сложность каждого из обсуждавшихся преобразований.

Преобразование НКА в ДКА

Время выполнения преобразования НКА или ε-НКА в ДКА может быть экспоненциальной функцией от количества состояний НКА. Начнем с того, что вычисление ε-замыкания n состояний занимает время $O(n^3)$. Необходимо найти все дуги с меткой ε, ведущие от каждого из n состояний. Если есть n состояний, то может быть не более n^2 дуг. Разумное использование системных ресурсов и хорошо спроектированные структуры данных гарантируют, что время исследования каждого состояния не превысит $O(n^2)$. В действительности для однократного вычисления всего ε-замыкания можно использовать такой алгоритм транзитивного замыкания, как алгоритм Уоршалла (Warshall)[5].

После вычисления ε-замыкания можно перейти к синтезу ДКА с помощью конструкции подмножеств. Основное влияние на расход времени оказывает количество состояний ДКА, которое может равняться 2^n. Для каждого состояния можно вычислить переходы за время $O(n^3)$, используя ε-замыкание и таблицу переходов НКА для каждого входного символа. Предположим, нужно вычислить $\delta(\{q_1, q_2, \ldots, q_k\}, a)$ для ДКА. Из каждого состояния q_i можно достичь не более n состояний вдоль путей с меткой ε, и каждое из этих состояний может иметь не более, чем n дуг с меткой a. Создав массив, проиндексированный состояниями, можно вычислить объединение не более n множеств, состоящих из не более, чем n состояний, за время, пропорциональное n^2.

Таким способом для каждого состояния q_i можно вычислить множество состояний, достижимых из q_i вдоль пути с меткой a (возможно, включая дуги, отмеченные ε). Поскольку $k \leq n$, то существует не более n таких состояний q_i, и для каждого из них вычис-

---

[5] Обсуждение алгоритмов транзитивного замыкания можно найти в книге A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1984. (А. Ахо, Дж. Хопкрофт, Дж. Ульман. *Структуры данных и алгоритмы*, М.: Издательский дом "Вильямс", 2000.)

ление достижимых состояний занимает время $O(n^2)$. Таким образом, общее время вычисления достижимых состояний равно $O(n^3)$. Для объединения множеств достижимых состояний потребуется только $O(n^2)$ дополнительного времени, следовательно, вычисление одного перехода ДКА занимает время $O(n^3)$.

Заметим, что количество входных символов считается постоянным и не зависит от n. Таким образом, как в этой, так и в других оценках времени работы количество входных символов не рассматривается. Размер входного алфавита влияет только на постоянный коэффициент, скрытый в обозначении "О большого".

Итак, время преобразования НКА в ДКА, включая ситуацию, когда НКА содержит ε-переходы, равно $O(n^3 2^n)$. Конечно, на практике обычно число состояний, которые строятся, намного меньше 2^n. Иногда их всего лишь n. Поэтому можно установить оценку времени работы равной $O(n^3 s)$, где s — это число состояний, которые в действительности есть у ДКА.

Преобразование ДКА в НКА

Это простое преобразование, занимающее время $O(n)$ для ДКА с n состояниями. Все, что необходимо сделать, — изменить таблицу переходов для ДКА, заключив в скобки {} состояния, а также добавить столбец для ε, если нужно получить ε-НКА. Поскольку число входных символов (т.е. ширина таблицы переходов) считается постоянным, копирование и обработка таблицы занимает время $O(n)$.

Преобразование автомата в регулярное выражение

Рассмотрев конструкцию из раздела 3.2.1, заметим, что на каждом из n этапов (где n — число состояний ДКА) размер конструируемого регулярного выражения может увеличиться в четыре раза, так как каждое выражение строится из четырех выражений предыдущего цикла. Таким образом, простая запись n^3 выражений может занять время $O(n^3 4^n)$. Улучшенная конструкция из раздела 3.2.2 уменьшает постоянный коэффициент, но не влияет на экспоненциальность этой задачи (в наихудшем случае).

Аналогичная конструкция требует такого же времени работы, если преобразуется НКА, или даже ε-НКА, но это здесь не доказывается. Однако использование конструкции для НКА имеет большое значение. Если сначала преобразовать НКА в ДКА, а затем ДКА — в регулярное выражение, то на это потребуется время $O(n^3 4^{n^3 2^n})$, которое является дважды экспоненциальным.

Преобразование регулярного выражения в автомат

Для преобразования регулярного выражения в ε-НКА потребуется линейное время. Необходимо эффективно проанализировать регулярное выражение, используя метод, занимающий время $O(n)$ для регулярного выражения длины n[6]. В результате получим де-

[6] Методы анализа, с помощью которых можно выполнить эту задачу за время O(n), обсуждаются в книге A. V. Aho, R. Sethi, and J. D. Ullman, *Compiler Design: Principles, Tools, and Techniques*, Addison-Wesley, 1986. (А. Ахо, Р. Сети, Дж. Ульман. *Компиляторы: принципы, инструменты и технологии*, М.: Издательский дом "Вильямс", 2001.)

рево с одним узлом для каждого символа регулярного выражения (хотя скобки в этом дереве не встречаются, поскольку они только управляют разбором выражения).

Полученное дерево заданного регулярного выражения можно обработать, конструируя ε-НКА для каждого узла. Правила преобразования регулярного выражения, представленные в разделе 3.2.3, никогда не добавляют более двух состояний и четырех дуг для каждого узла дерева выражения. Следовательно, как число состояний, так и число дуг результирующего ε-НКА равны $O(n)$. Кроме того, работа по созданию этих элементов в каждом узле дерева анализа является постоянной при условии, что функция, обрабатывающая каждое поддерево, возвращает указатели в начальное и допускающие состояния этого автомата.

Приходим к выводу, что построение ε-НКА по регулярному выражению занимает время, линейно зависящее от размера выражения. Можно исключить ε-переходы из ε-НКА с n состояниями, преобразовав его в обычный НКА за время $O(n^3)$ и не увеличив числа состояний. Однако преобразование в ДКА может занять экспоненциальное время.

4.3.2. Проверка пустоты регулярных языков

На первый взгляд ответ на вопрос “является ли регулярный язык L пустым?” кажется очевидным: язык $\varnothing$ пуст, а все остальные регулярные языки — нет. Однако, как говорилось в начале раздела 4.3, при постановке задачи явный перечень цепочек языка L не приводится. Обычно задается некоторое представление языка L, и нужно решить, обозначает ли оно язык $\varnothing$, или нет.

Если язык задан с помощью конечного автомата любого вида, то вопрос пустоты состоит в том, есть ли какие-нибудь пути из начального состояния в допускающие. Если есть, то язык непуст, а если все допускающие состояния изолированы от начального, то язык пуст. Ответ на вопрос, можно ли перейти в допускающее состояние из начального, является простым примером достижимости в графах, подобным вычислению ε-замыкания, рассмотренному в разделе 2.5.3. Искомый алгоритм можно сформулировать следующим рекурсивным образом.

Базис. Начальное состояние всегда достижимо из начального состояния.

Индукция. Если состояние q достижимо из начального, и есть дуга из q в состояние p с любой меткой (входным символом или ε, если рассматривается ε-НКА), то p также достижимо.

Таким способом можно вычислить все множество достижимых состояний. Если среди них есть допускающее состояние, то ответом на поставленный вопрос будет “нет” (язык данного автомата *не* пуст), в противном случае ответом будет “да” (язык пуст). Заметим, что если автомат имеет n состояний, то вычисление множества достижимых состояний занимает время не более $O(n^2)$ (практически это время пропорционально числу дуг на диаграмме переходов автомата, которое может быть и меньше n^2).

Если язык L представлен регулярным выражением, а не автоматом, то можно преобразовать это выражение в ε-НКА, а далее продолжить так, как описано выше. Поскольку автомат, полученный в результате преобразования регулярного выражения длины n, содержит не более $O(n)$ состояний и переходов, для выполнения алгоритма потребуется время $O(n)$.

Можно проверить само выражение — пустое оно, или нет. Сначала заметим, что если в данном выражении ни разу не встречается $\varnothing$, то его язык гарантированно не пуст. Если же в выражении встречается $\varnothing$, то язык такого выражения не обязательно пустой. Используя следующие рекурсивные правила, можно определить, представляет ли заданное регулярное выражение пустой язык.

Базис. $\varnothing$ обозначает пустой язык, но ε и **a** для любого входного символа a обозначают не пустой язык.

Индукция. Пусть R — регулярное выражение. Нужно рассмотреть четыре варианта, соответствующие возможным способам построения этого выражения.

1. $R = R_1 + R_2$. $L(R)$ пуст тогда и только тогда, когда оба языка $L(R_1)$ и $L(R_2)$ пусты.
2. $R = R_1R_2$. $L(R)$ пуст тогда и только тогда, когда хотя бы один из языков $L(R_1)$ и $L(R_2)$ пуст.
3. $R = R_1^*$. $L(R)$ не пуст: он содержит цепочку ε.
4. $R = (R_1)$. $L(R)$ пуст тогда и только тогда, кода $L(R_1)$ пуст, так как эти языки равны.

4.3.3. Проверка принадлежности регулярному языку

Следующий важный вопрос состоит в том, принадлежит ли данная цепочка w данному регулярному языку L. В то время, как цепочка w задается явно, язык L представляется с помощью автомата или регулярного выражения.

Если язык L задан с помощью ДКА, то алгоритм решения данной задачи очень прост. Имитируем ДКА, обрабатывающий цепочку входных символов w, начиная со стартового состояния. Если ДКА заканчивает в допускающем состоянии, то цепочка w принадлежит этому языку, в противном случае — нет. Этот алгоритм является предельно быстрым. Если $|w| = n$ и ДКА представлен с помощью подходящей структуры данных, например, двухмерного массива (таблицы переходов), то каждый переход требует постоянного времени, а вся проверка занимает время $O(n)$.

Если L представлен способом, отличным от ДКА, то преобразуем это представление в ДКА и применим описанную выше проверку. Такой подход может занять время, экспоненциально зависящее от размера данного представления и линейное относительно $|w|$. Однако, если язык задан с помощью НКА или ε-НКА, то намного проще и эффективнее непосредственно проимитировать этот НКА. Символы цепочки w обрабатываются по одному, и запоминается множество состояний, в которые НКА может попасть после

прохождения любого пути, помеченного префиксом цепочки w. Идея такой имитации была представлена на рис. 2.10.

Если длина цепочки w равна n, а количество состояний НКА равно s, то время работы этого алгоритма равно $O(ns^2)$. Чтобы обработать очередной входной символ, необходимо взять предыдущее множество состояний, число которых не больше s, и для каждого из них найти следующее состояние. Затем объединяем не более s множеств, состоящих из не более, чем s состояний, для чего нужно время $O(s^2)$.

Если заданный НКА содержит ε-переходы, то перед тем, как начать имитацию, необходимо вычислить ε-замыкание. Такая обработка очередного входного символа a состоит из двух стадий, каждая из которых занимает время $O(s^2)$. Сначала для предыдущего множества состояний находим последующие состояния при входе a. Далее вычисляем ε-замыкание полученного множества состояний. Начальным множеством состояний для такого моделирования будет ε-замыкание начального состояния НКА.

И наконец, если язык L представлен регулярным выражением, длина которого s, то за время $O(s)$ можно преобразовать это выражение в ε-НКА с числом состояний не больше $2s$. Выполняем описанную выше имитацию, что требует $O(ns^2)$ времени для входной цепочки w длиной n.

4.3.4. Упражнения к разделу 4.3

4.3.1. (*) Приведите алгоритм определения, является ли регулярный язык L бесконечным. *Указание.* Используйте лемму о накачке для доказательства того, что если язык содержит какую-нибудь цепочку, длина которой превышает определенную нижнюю границу, то этот язык должен быть бесконечным.

4.3.2. Приведите алгоритм определения, содержит ли регулярный язык L по меньшей мере 100 цепочек.

4.3.3. Пусть L — регулярный язык в алфавите Σ. Приведите алгоритм проверки равенства $L = \Sigma^*$, т.е. содержит ли язык L все цепочки в алфавите Σ.

4.3.4. Приведите алгоритм определения, содержат ли регулярные языки L_1 и L_2 хотя бы одну общую цепочку.

4.3.5. Пусть L_1 и L_2 — два регулярных языка с одним и тем же алфавитом Σ. Приведите алгоритм определения, существует ли цепочка из Σ^*, которая не принадлежит ни L_1, ни L_2.

4.4. Эквивалентность и минимизация автоматов

В отличие от предыдущих вопросов — пустоты и принадлежности, алгоритмы решения которых были достаточно простыми, вопрос о том, определяют ли два представления двух регулярных языков один и тот же язык, требует значительно бо́льших интеллектуальных

усилий. В этом разделе мы обсудим, как проверить, являются ли два описания регулярных языков *эквивалентными* в том смысле, что они задают один и тот же язык. Важным следствием этой проверки является возможность минимизации ДКА, т.е. для любого ДКА можно найти эквивалентный ему ДКА с минимальным количеством состояний. По существу, такой ДКА один: если даны два эквивалентных ДКА с минимальным числом состояний, то всегда можно переименовать состояния так, что эти ДКА станут одинаковыми.

4.4.1. Проверка эквивалентности состояний

Начнем с вопроса об эквивалентности состояний одного ДКА. Наша цель — понять, когда два различных состояния p и q можно заменить одним, работающим одновременно как p и q. Будем говорить, что состояния p и q *эквивалентны*, если

- для всех входных цепочек w состояние $\hat{\delta}(p, w)$ является допускающим тогда и только тогда, когда состояние $\hat{\delta}(q, w)$ — допускающее.

Менее формально, эквивалентные состояния p и q невозможно различить, если просто проверить, допускает ли автомат данную входную цепочку, начиная работу в одном (неизвестно, каком именно) из этих состояний. Заметим, что состояния $\hat{\delta}(p, w)$ и $\hat{\delta}(q, w)$ могут и не совпадать — лишь бы оба они были либо допускающими, либо недопускающими.

Если два состояния p и q не эквивалентны друг другу, то будем говорить, что они *различимы*, т.е. существует хотя бы одна цепочка w, для которой одно из состояний $\hat{\delta}(p, w)$ и $\hat{\delta}(q, w)$ является допускающим, а другое — нет.

Пример 4.18. Рассмотрим ДКА на рис. 4.8. Функцию переходов этого автомата обозначим через δ. Очевидно, что некоторые пары состояний не эквивалентны, например C и G, потому что первое из них допускающее, а второе — нет. Пустая цепочка различает эти состояния, так как $\hat{\delta}(C, \varepsilon)$ — допускающее состояние, а $\hat{\delta}(G, \varepsilon)$ — нет.

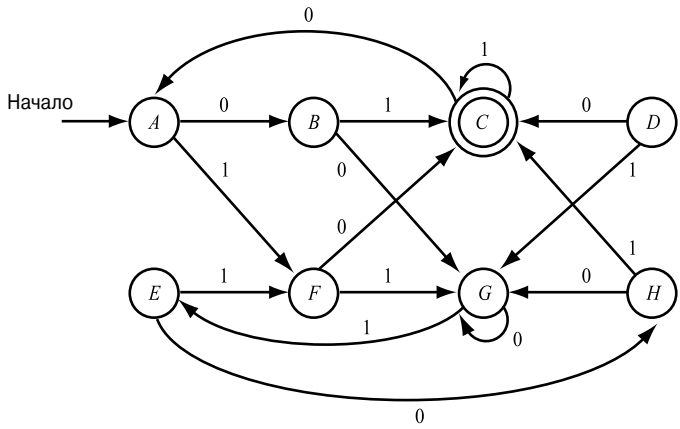


Рис. 4.8. Автомат с эквивалентными состояниями

Рассмотрим состояния A и G. Различить их с помощью цепочки ε невозможно, так как оба они недопускающие. 0 также не различает их, поскольку по входу 0 автомат переходит в состояния B и G, соответственно, а оба эти состояния недопускающие. Однако цепочка 01 различает A и G, так как $\hat{\delta}(A, 01) = C$, $\hat{\delta}(G, 01) = E$, состояние C — допускающее, а E — нет. Для доказательства неэквивалентности A и G достаточно любой входной цепочки, переводящей автомат из состояний A и G в состояния, одно из которых является допускающим, а второе — нет.

Рассмотрим состояния A и E. Ни одно из них не является допускающим, так что цепочка ε не различает их. По входу 1 автомат переходит и из A, и из E в состояние F. Таким образом, ни одна входная цепочка, начинающаяся с 1, не может различить их, поскольку $\hat{\delta}(A, 1x) = \hat{\delta}(E, 1x)$ для любой цепочки x.

Рассмотрим поведение в состояниях A и E на входах, которые начинаются с 0. Из состояний A и E автомат переходит в B и H, соответственно. Так как оба эти состояния недопускающие, сама по себе цепочка 0 не отличает A от E. Однако состояния B и H не помогут: по входу 1 оба эти состояния переходят в C, а по входу 0 — в G. Значит, ни одна входная цепочка, начинающаяся с 0, не может различить состояния A и E. Следовательно, ни одна входная цепочка не различает состояния A и E, т.е. они эквивалентны. □

Для того чтобы найти эквивалентные состояния, нужно выявить все пары различимых состояний. Как ни странно, но если найти все пары состояний, различимых в соответствии с представленным ниже алгоритмом, то те пары состояний, которые найти не удастся, будут эквивалентными. Алгоритм, который называется *алгоритмом заполнения таблицы*, состоит в рекурсивном обнаружении пар различимых состояний ДКА $A = (Q, \Sigma, \delta, q_0, F)$.

Базис. Если состояние p — допускающее, а q — не допускающее, то пара состояний $\{p, q\}$ различима.

Индукция. Пусть p и q — состояния, для которых существует входной символ a, приводящий их в различимые состояния $r = \delta(p, a)$ и $s = \delta(q, a)$. Тогда $\{p, q\}$ — пара различимых состояний. Это правило очевидно, потому что должна существовать цепочка w, отличающая r от s, т.е. только одно из состояний $\hat{\delta}(r, w)$ и $\hat{\delta}(s, w)$ является допускающим. Тогда цепочка aw отличает p от q, так как $\hat{\delta}(p, aw)$ и $\hat{\delta}(q, aw)$ — это та же пара состояний, что и $\hat{\delta}(r, w)$ и $\hat{\delta}(s, w)$.

Пример 4.19. Выполним алгоритм заполнения таблицы для ДКА, представленного на рис. 4.8. Окончательный вариант таблицы изображен на рис. 4.9, где x обозначает пары различимых состояний, а пустые ячейки указывают пары эквивалентных состояний. Сначала в таблице нет ни одного x.

В базисном случае, поскольку C — единственное допускающее состояние, записываем x в каждую пару состояний, в которую входит C. Зная некоторые пары различимых состояний, можно найти другие. Например, поскольку пара $\{C, H\}$ различима, а состоя-

ния E и F по входу 0 переходят в H и C, соответственно, то пара $\{E, F\}$ также различима. Фактически, все x на рис. 4.9, за исключением пары $\{A, G\}$, получаются очень просто: посмотрев на переходы из каждой пары состояний по символам 0 или 1, обнаружим (для одного из этих символов), что одно состояние переходит в C, а другое — нет. Различимость пары состояний $\{A, G\}$ видна в следующем цикле, поскольку по символу 1 они переходят в F и E, соответственно, а различимость состояний $\{E, F\}$ уже установлена.

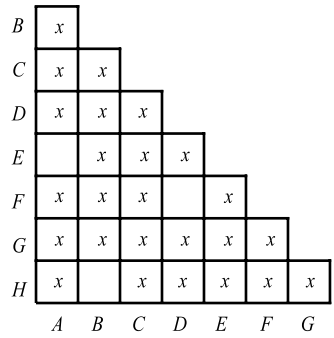


Рис. 4.9. Таблица неэквивалентности состояний

Однако обнаружить другие пары различимых состояний невозможно. Следовательно, оставшиеся три пары состояний $\{A, E\}$, $\{B, H\}$ и $\{D, F\}$ эквивалентны. Выясним, почему нельзя утверждать, что пара состояний $\{A, E\}$ различима. По входному символу 0 состояния A и E переходят в B и H, соответственно, а про эту пару пока неизвестно, различима она, или нет. По символу 1 оба состояния A и E переходят в F, так что нет никакой надежды различить их этим способом. Остальные две пары, $\{B, H\}$ и $\{D, F\}$, различить нельзя, поскольку у них одинаковые переходы как по символу 0, так и по 1. Таким образом, алгоритм заполнения таблицы останавливается на таблице, представленной на рис. 4.9, и корректно определяет эквивалентные и различимые состояния. □

Теорема 4.20. Если два состояния не различаются с помощью алгоритма заполнения таблицы, то они эквивалентны.

Доказательство. Снова рассмотрим ДКА $A = (Q, \Sigma, \delta, q_0, F)$. Предположим, что утверждение теоремы неверно, т.е. существует хотя бы одна пара состояний $\{p, q\}$, для которой выполняются следующие условия.

1. Состояния p и q различимы, т.е. существует некоторая цепочка w, для которой только одно из состояний $\hat{\delta}(p,w)$ и $\hat{\delta}(q,w)$ является допускающим.
2. Алгоритм заполнения таблицы не может обнаружить, что состояния p и q различимы.

Назовем такую пару состояний *плохой парой*.

Если существуют плохие пары, то среди них должны быть такие, которые различимы с помощью кратчайших из всех цепочек, различающих плохие пары. Пусть пара

$\{p, q\}$ — плохая, а $w = a_1a_2\ldots a_n$ — кратчайшая из всех цепочек, различающих p и q. Тогда только одно из состояний $\hat{\delta}(p, w)$ и $\hat{\delta}(q, w)$ является допускающим.

Заметим, что цепочка w не может быть ε, так как, если некоторая пара состояний различается с помощью ε, то ее можно обнаружить, выполнив базисную часть алгоритма заполнения таблицы. Следовательно, $n \geq 1$.

Рассмотрим состояния $r = \delta(p, a_1)$ и $s = \delta(q, a_1)$. Эти состояния можно различить с помощью цепочки $a_2a_3\ldots a_n$, поскольку она переводит r и s в состояния $\hat{\delta}(p, w)$ и $\hat{\delta}(q, w)$. Однако цепочка, отличающая r от s, короче любой цепочки, различающей плохую пару. Следовательно, $\{r, s\}$ не может быть плохой парой, и алгоритм заполнения таблицы должен был обнаружить, что эти состояния различимы.

Но индуктивная часть алгоритма заполнения таблицы не остановится, пока не придет к выводу, что состояния p и q также различимы, поскольку уже обнаружено, что состояние $\delta(p, a_1) = r$ отличается от $\delta(q, a_1) = s$. Получено противоречие с предположением о том, что существуют плохие пары состояний. Но если плохих пар нет, то любую пару различимых состояний можно обнаружить с помощью алгоритма заполнения таблицы, и теорема доказана. □

4.4.2. Проверка эквивалентности регулярных языков

Эквивалентность регулярных языков легко проверяется с помощью алгоритма заполнения таблицы. Предположим, что языки L и M представлены, например, один — регулярным выражением, а второй — некоторым НКА. Преобразуем каждое из этих представлений в ДКА. Теперь представим себе ДКА, состояния которого равны объединению состояний автоматов для языков L и M. Технически этот ДКА содержит два начальных состояния, но фактически при проверке эквивалентности начальное состояние не играет никакой роли, поэтому любое из этих двух состояний можно принять за единственное начальное.

Далее проверяем эквивалентность начальных состояний двух заданных ДКА, используя алгоритм заполнения таблицы. Если они эквивалентны, то $L = M$, а если нет, то $L \neq M$.

Пример 4.21. Рассмотрим два ДКА (рис. 4.10). Каждый ДКА допускает пустую цепочку и все цепочки, которые заканчиваются символом 0, т.е. язык регулярного выражения $\varepsilon + (\mathbf{0} + \mathbf{1})^*\mathbf{0}$. Можно представить, что на рис. 4.10 изображен один ДКА, содержащий пять состояний от A до E. Если применить алгоритм заполнения таблицы к этому автомату, то в результате получим таблицу, представленную на рис. 4.11.

Чтобы заполнить эту таблицу, начнем с размещения x в ячейках, соответствующих тем парам состояний, из которых только одно является допускающим. Оказывается, что больше делать ничего не нужно. Остальные четыре пары $\{A, C\}$, $\{A, D\}$, $\{C, D\}$ и $\{B, E\}$ являются парами эквивалентных состояний. Необходимо убедиться, что в индуктивной части алгоритма заполнения таблицы различимые состояния не обнаружены. Например, с помощью такой таблицы (см. рис. 4.11) нельзя различить пару $\{A, D\}$, так как по символу 0 эти состояния переходят сами в себя, а по 1 — в пару состояний $\{B, E\}$, которая

осталась неразличимой. Поскольку в результате проверки установлено, что состояния A и C эквиваленты и являются начальными у двух заданных автоматов, делаем вывод, что эти ДКА действительно допускают один и тот же язык. □

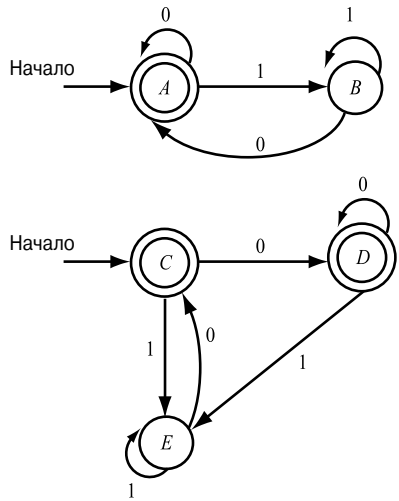


Рис. 4.10. Два эквивалентных ДКА

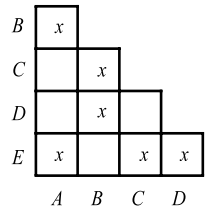


Рис. 4.11. Таблица различимости для автоматов, представленных на рис. 4.10

Время заполнения таблицы, а значит и время проверки эквивалентности двух состояний, полиномиально относительно числа состояний. Если число состояний равно n, то количество пар состояний равно $\binom{n}{2}$, или $n(n-1)/2$. За один цикл рассматриваются все пары состояний, чтобы определить, является ли одна из пар состояний-преемников различимой. Значит, один цикл занимает время не больше $O(n^2)$. Кроме того, если в некотором цикле не обнаружены новые пары различимых состояний, то алгоритм заканчивается. Следовательно, количество циклов не превышает $O(n^2)$, а верхняя граница времени заполнения таблицы равна $O(n^4)$.

Однако с помощью более аккуратно построенного алгоритма можно заполнить таблицу за время $O(n^2)$. С этой целью для каждой пары состояний $\{r, s\}$ необходимо составить список пар состояний $\{p, q\}$, "зависящих" от $\{r, s\}$, т.е., если пара $\{r, s\}$ различима, то $\{p, q\}$ также различима. Вначале такие списки создаются путем рассмотрения каждой

пары состояний $\{p, q\}$, и для каждого входного символа a (а их число фиксировано) пара $\{p, q\}$ вносится в список для пары состояний-преемников $\{\delta(p, a), \delta(q, a)\}$.

Если обнаруживается, что пара $\{r, s\}$ различима, то в списке этой пары каждая пока неразличимая пара отмечается как различимая и помещается в очередь пар, списки которых нужно проверить аналогичным образом.

Общее время работы этого алгоритма пропорционально сумме длин списков, так как каждый раз либо что-то добавляется в списки (инициализация), либо в первый и последний раз проверяется наличие некоторой пары в списке (когда проходим по списку пары, признанной различимой). Так как размер входного алфавита считается постоянным, то каждая пара состояний попадает в $O(1)$ списков. Поскольку всего пар $O(n^2)$, суммарное время также $O(n^2)$.

4.4.3. Минимизация ДКА

Еще одним важным следствием проверки эквивалентности состояний является возможность "минимизации" ДКА. Это значит, что для каждого ДКА можно найти эквивалентный ему ДКА с наименьшим числом состояний. Более того, для данного языка существует единственный минимальный ДКА (с точностью до выбираемого нами обозначения состояний).

Основная идея минимизации ДКА состоит в том, что понятие эквивалентности состояний позволяет объединять состояния в блоки следующим образом.

1. Все состояния в блоке эквиваленты.
2. Любые два состояния, выбранные из разных блоков, неэквивалентны.

Пример 4.22. Рассмотрим рис. 4.9, на котором представлены эквивалентность и различимость для состояний, изображенных на рис. 4.8. Эти состояния разбиваются на эквивалентные блоки следующим образом: $(\{A, E\}, \{B, H\}, \{C\}, \{D, F\}, \{G\})$. Заметим, что каждая пара эквивалентных состояний помещена в отдельный блок, а состояния, отличимые от всех остальных, образуют отдельные блоки.

Для автомата, представленного на рис. 4.10, разбиение на блоки имеет вид $(\{A, C, D\}, \{B, E\})$. Этот пример показывает, что в блоке может быть более двух состояний. Может показаться случайностью, что состояния A, C и D помещены в один блок потому, что каждые два из них эквивалентны и ни одно из этих состояний не эквивалентно еще какому-нибудь состоянию, кроме этих. Однако следующая теорема утверждает, что такая ситуация следует из определения эквивалентности состояний. □

Теорема 4.23. Эквивалентность состояний транзитивна, т.е., если для некоторого ДКА $A = (Q, \Sigma, \delta, q_0, F)$ состояние p эквивалентно q, а q — r, то состояния p и r также эквивалентны.

Доказательство. Естественно ожидать, что любое отношение, называемое "эквивалентностью", обладает свойством транзитивности. Однако, просто назвав какое-то от-

ношение “эквивалентностью”, нельзя гарантировать, что оно транзитивно — это нужно доказать.

Предположим, что $\{p, q\}$ и $\{q, r\}$ — пары эквивалентных состояний, а пара $\{p, r\}$ — различима. Тогда должна существовать такая цепочка w, для которой только одно из состояний $\hat{\delta}(p, w)$ и $\hat{\delta}(r, w)$ является допускающим. Используя симметрию, предположим, что $\hat{\delta}(p, w)$ — допускающее.

Теперь посмотрим, будет ли состояние $\hat{\delta}(q, w)$ допускающим. Если оно допускающее, то пара $\{q, r\}$ различима, так как состояние $\hat{\delta}(q, w)$ —допускающее, а $\hat{\delta}(r, w)$ — нет. Если $\hat{\delta}(q, w)$ не допускающее, то по аналогичным причинам пара $\{p, q\}$ различима. Полученное противоречие доказывает неразличимость пары $\{p, r\}$, т.е. состояния p и r эквивалентны. □

Теорему 4.23 можно использовать для обоснования очевидного алгоритма разбиения состояний. Для каждого состояния q строится блок, состоящий из q и всех эквивалентных ему состояний. Необходимо доказать, что результирующие блоки образуют разбиение множества состояний, т.е. ни одно состояние не принадлежит двум разным блокам.

Сначала заметим, что состояния внутри каждого блока взаимно эквивалентны, т.е., если p и r принадлежат блоку состояний, эквивалентных q, то согласно теореме 4.23 они эквивалентны.

Предположим, что существуют два пересекающихся, но не совпадающих блока, т.е. существует блок B, содержащий состояния p и q, и блок C, который содержит p, но не q. Поскольку состояния p и q принадлежат одному блоку, то они эквивалентны. Рассмотрим возможные варианты построения блока C. Если этот блок образован состоянием p, то q было бы в блоке C, так как эти состояния эквивалентны. Следовательно, существует некоторое третье состояние s, порождающее блок C, т.е. C — это множество состояний, эквивалентных s.

Состояния p и s эквивалентны, так как оба принадлежат C. Также p эквивалентно q, потому что оба эти состояния принадлежат B. Согласно свойству транзитивности, доказанному в теореме 4.23, состояние q эквивалентно s. Но тогда q принадлежит блоку C, что противоречит предположению о существовании пересекающихся, но не совпадающих блоков. Итак, эквивалентность состояний задает их разбиение, т.е. любые два состояния имеют или совпадающие, или не пересекающиеся множества эквивалентных им состояний. Следующая теорема подытоживает результаты проведенного анализа.

Теорема 4.24. Если для каждого состояния q некоторого ДКА создать *блок*, состоящий из q и эквивалентных ему состояний, то различные блоки состояний образуют *разбиение* множества состояний.[7] Это значит, что каждое состояние может принадлежать

[7] Нужно помнить, что один и тот же блок может формироваться несколько раз, начиная с разных состояний. Однако разбиение состоит из *различных* блоков, так что каждый блок встречается в разбиении только один раз.

только одному блоку. Состояния из одного блока эквивалентны, а любые состояния, выбранные из разных блоков, не эквивалентны. □

Теперь можно кратко сформулировать алгоритм минимизации ДКА $A = (Q, \Sigma, \delta, q_0, F)$.

1. Для выявления всех пар эквивалентных состояний применяем алгоритм заполнения таблицы.
2. Разбиваем множество состояний Q на блоки взаимно эквивалентных состояний с помощью описанного выше метода.
3. Строим ДКА B с минимальным числом состояний, используя в качестве его состояний полученные блоки. Пусть γ— функция переходов автомата B. Предположим, что S — множество эквивалентных состояний автомата A, a — входной символ. Тогда должен существовать один блок состояний T, содержащий $\gamma(q, a)$ для всех состояний q из S. Если это не так, то входной символ a переводит два состояния p и q из S в состояния, принадлежащие разным блокам согласно теореме 4.24. Из этого можно сделать вывод, что состояния p и q не были эквивалентными и не могли вместе принадлежать S. В результате определяется функция переходов $\gamma(S, a) = T$. Кроме того:
   а) начальным состоянием ДКА B является блок, содержащий начальное состояние автомата A;
   б) множеством допускающих состояний автомата B является множество блоков, содержащих допускающие состояния ДКА A. Заметим, что если одно состояние в блоке является допускающим, то все остальные состояния этого блока также должны быть допускающими. Причина в том, что любое допускающее состояние отличимо от любого недопускающего, поэтому допускающее и недопускающее состояния не могут принадлежать одному блоку эквивалентных состояний.

Пример 4.25. Минимизируем ДКА, представленный на рис. 4.8. В примере 4.22 установлены блоки разбиения состояний. На рис. 4.12 изображен ДКА с минимальным числом состояний. Пять состояний этого автомата соответствуют пяти блокам эквивалентных состояний автомата на рис. 4.8.

Начальным состоянием минимизированного автомата является $\{A, E\}$, так как A было начальным на рис. 4.8. Единственным допускающим — $\{C\}$, поскольку C — это единственное допускающее состояние на рис. 4.8. Заметим, что переходы на рис. 4.12 правильно отражают переходы на рис. 4.8. Например, на рис. 4.12 есть переход из $\{A, E\}$ в $\{B, H\}$ по символу 0. Это очевидно, так как A на рис. 4.8 переходит в B при чтении 0, а E — в H. Аналогично, при чтении 1 $\{A, E\}$ переходит в $\{D, F\}$. По рис. 4.8 легко увидеть, что оба состояния A и E переходят в F по 1, так что для $\{A, E\}$ состояние-преемник по 1 также выбрано правильно. Тот факт, что ни A, ни E не переходят в D по 1, неважен. Читатель может проверить, что и все остальные переходы изображены правильно. □

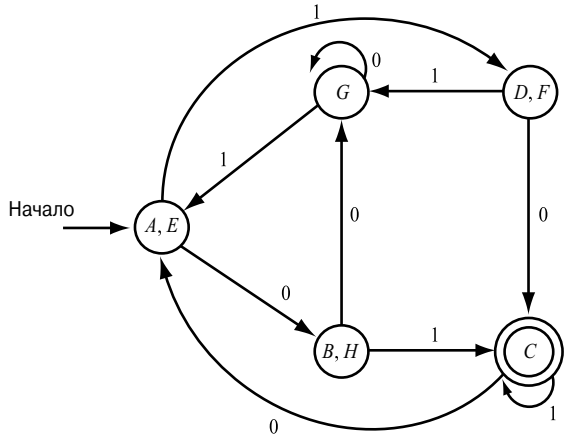


Рис. 4.12. ДКА с минимальным числом состояний, эквивалентный автомату, изображенному на рис. 4.8

4.4.4. Почему минимизированный ДКА невозможно улучшить

Предположим, что задан ДКА A, и мы минимизируем его до ДКА M с помощью метода разбиения из теоремы 4.24. Эта теорема показывает, что невозможно получить эквивалентный ДКА, группируя состояния автомата A в еще меньшее число групп. Но все же, может ли существовать другой ДКА N, не связанный с A, который допускал бы тот же язык, что и автоматы A и M, но имел бы состояний меньше, чем автомат M? Методом от противного докажем, что такого автомата не существует.

Сначала применим алгоритм различимости состояний из раздела 4.4.1 к состояниям автоматов M и N так, как если бы это был один автомат. Можно предположить, что общих обозначений состояний у M и N нет, так что функция переходов комбинированного автомата будет объединением функций переходов автоматов M и N, которые между собой не пересекаются. Состояния комбинированного ДКА будут допускающими тогда и только тогда, когда они являются допускающими в соответствующих им автоматах.

Начальные состояния автоматов M и N неразличимы, так как $L(M) = L(N)$. Далее, если состояния $\{p, q\}$ неразличимы, то для любого входного символа их преемники также неразличимы. Если бы преемников можно было различить, то p и q также можно было различить.

Минимизация состояний НКА

Может показаться, что метод разбиения состояний, минимизирующий ДКА, применим и для построения НКА с минимальным числом состояний, эквивалентного данному НКА или ДКА. Хотя методом полного перебора можно найти НКА с наименьшим количеством состояний, допускающий данный язык, просто сгруппировать состояния некоторого заданного НКА для этого языка нельзя.

Пример приведен на рис. 4.13. Никакие из трех состояний не являются эквивалентными. Очевидно, что допускающее состояние *B* отличается от недопускающих состояний *A* и *C*. Состояния *A* и *C* различаются входным символом 0. *C* переходит в {*A*} (недопускающее), тогда как *A* переходит в {*A*, *B*}, которое включает допускающее состояние. Таким образом, группирование состояний не уменьшает количества состояний на рис. 4.13.

Однако можно построить меньший НКА для этого же языка, просто удалив состояние *C*. Заметим, что *A* и *B* (без *C*) допускают все цепочки, которые заканчиваются нулем, а добавление *C* не позволяет допускать другие цепочки.

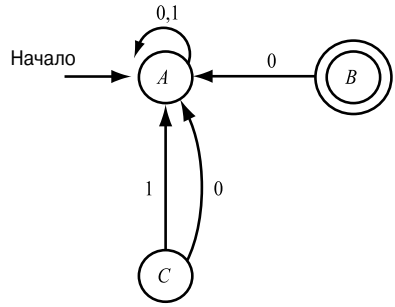


Рис. 4.13. НКА, который невозможно минимизировать с помощью эквивалентности состояний

Ни *M*, ни *N* не могут иметь недостижимых состояний, иначе, исключив это состояние, можно было бы получить еще меньший ДКА для того же языка. Следовательно, каждое состояние автомата *M* неотличимо хотя бы от одного состояния автомата *N*. Выясним, почему это так. Пусть *p* — состояние автомата *M*. Тогда существует цепочка $a_1a_2\ldots a_k$, переводящая *M* из начального состояния в *p*. Эта цепочка также переводит *N* из начального в некоторое состояние *q*. Из того, что начальные состояния этих автоматов неразличимы, следует, что состояния-преемники, соответствующие входному символу a_1, также неразличимы. Состояния, следующие за этими состояниями при чтении a_2, также будут неразличимыми, и так далее, пока мы не придем к заключению, что состояния *p* и *q* неразличимы.

Поскольку автомат *N* содержит меньше состояний, чем *M*, то должны существовать два состояния автомата *M*, которые неотличимы от одного и того же состояния автомата *N*. Значит, эти состояния неразличимы. Но автомат *M* построен таким образом, что все его состояния отличимы друг от друга. Противоречие. Следовательно, предположение о существовании *N* неверно, и *M* действительно является ДКА с наименьшим количеством состояний среди всех ДКА, эквивалентных *A*. Формально доказана следующая теорема.

Теорема 4.26. Если из некоторого ДКА *A* с помощью алгоритма, описанного в теореме 4.24, построен ДКА *M*, то *M* имеет наименьшее число состояний из всех ДКА, эквивалентных *A*. □

Можно сформулировать более сильное утверждение, чем теорема 4.26. Между состояниями любого минимального ДКА *N* и состояниями ДКА *M* должно существовать взаимно

однозначное соответствие. Как уже доказано, каждое состояние M эквивалентно одному состоянию N, и ни одно состояние M не может быть эквивалентным двум состояниям N. Аналогично можно доказать, что ни одно состояние N не может быть эквивалентным двум состояниям M, хотя каждое состояние автомата N должно быть эквивалентно одному из состояний ДКА M. Следовательно, существует только один ДКА с минимальным количеством состояний, эквивалентный A (с точностью до обозначений состояний).

4.4.5. Упражнения к разделу 4.4

4.4.1. (*) На рис. 4.14 представлена таблица переходов некоторого ДКА:

а) составьте таблицу различимости для этого автомата;

б) постройте эквивалентный ДКА с минимальным числом состояний.

	0	*1*
→*A*	*B*	*A*
B	*A*	*C*
C	*D*	*E*
**D*	*D*	*A*
E	*D*	*F*
F	*G*	*E*
G	*F*	*G*
H	*G*	*D*

Рис. 4.14. ДКА, который нужно минимизировать

Выполните упражнение 4.4.1 для ДКА, представленного на рис. 4.15.

	0	1
→A	B	E
B	C	F
*C	D	H
D	E	H
E	F	I
*F	G	B
G	H	B
H	I	C
*I	A	E

Рис. 4.15. Еще один ДКА, который нужно минимизировать

4.4.3. (!!) Пусть p и q — пара различимых состояний заданного ДКА A с n состояниями. Какой может быть самая точная верхняя граница длины кратчайшей цепочки, различающей p и q, как функция от n?

Резюме

- *Лемма о накачке.* Если язык регулярен, то в каждой достаточно длинной цепочке этого языка есть непустая подцепочка, которую можно "накачать", т.е. повторить произвольное число раз; получаемые при этом цепочки будут принадлежать данному языку. Эта лемма используется для доказательства *нерегулярности* многих языков.
- *Операции, сохраняющие регулярность языков.* Существует много операций, результат применения которых к регулярным языкам также является регулярным языком. В их числе объединение, конкатенация, замыкание (итерация), пересечение, дополнение, разность, обращение, гомоморфизм (замена каждого символа соответствующей цепочкой) и обратный гомоморфизм.
- *Проверка пустоты регулярного языка.* Существует алгоритм, который по такому заданному представлению регулярного языка, как автомат или регулярное выражение, определяет, является ли представленный язык пустым множеством.
- *Проверка принадлежности регулярному языку.* Существует алгоритм, который по заданной цепочке и некоторому представлению регулярного языка определяет, принадлежит ли цепочка языку.
- *Проверка различимости состояний.* Два состояния некоторого ДКА различимы, если существует входная цепочка, которая переводит в допускающее только одно из этих состояний. Если начать с того, что все пары, состоящие из допускающего и недопускающего состояний, различимы, и найти дополнительные пары, которые по одному символу переходят в различимые состояния, можно обнаружить все пары различимых состояний.
- *Минимизация детерминированных конечных автоматов.* Состояния любого ДКА можно разбить на группы взаимно неразличимых состояний. Состояния из двух разных групп всегда различимы. Если заменить каждую группу одним состоянием, получим эквивалентный ДКА с наименьшим числом состояний.

Литература

За исключением очевидных свойств замкнутости регулярных выражений (относительно объединения, конкатенации и итерации), которые были доказаны Клини [6], почти все результаты свойств замкнутости воспроизводят аналогичные результаты, полученные для контекстно-свободных языков (этому классу языков посвящены следующие главы). Таким образом, лемма о накачке для регулярных языков является упрощением соответствующего результата для контекстно-свободных языков (Бар-Хиллел, Перлес и Шамир [1]). Из результатов этой работы следуют некоторые другие свойства замкнутости, представленные в данной главе, а замкнутость относительно обратного гомоморфизма обоснована в [2].

Операция деления (см. упражнение 4.2.2) представлена в [3]. В этой работе обсуждается более общая операция, в которой вместо одиночных символов находятся регулярные языки. Ряд операций "частичного удаления", начиная с упражнения 4.2.8, в котором говорилось о первых половинах цепочек регулярного языка, был определен в [8]. Сейферас и Мак-Нотон [9] изучили общий случай, когда операция удаления сохраняет регулярность языков.

Алгоритмы разрешения, такие как проверка пустоты и конечности регулярных языков, а также проверка принадлежности к регулярному языку, берут свое начало в [7]. Алгоритмы минимизации числа состояний ДКА появились в [5]. В работе [4] предложен наиболее эффективный алгоритм нахождения минимального ДКА.

1. Y. Bar-Hillel, M. Perles, and E. Shamir, "On formal properties of simple phrase-structure grammars," *Z. Phonetik. Spachwiss. Kommunikations-forsch.* **14** (1961), pp. 143–172.
2. S. Ginsburg and G. Rose, "Operations which preserve definability in languages," *J. ACM* **10**:2 (1963), pp. 175–195. (Гинзбург С., Роуз Дж. Об инвариантности классов языков относительно некоторых преобразований. — Кибернетический сборник, Новая серия, вып. 5. — М.: Мир, 1968. — С. 138–166.)
3. S. Ginsburg and E. H. Spanier, "Quotients of context-free languages," *J. ACM* **10**:4 (1963), pp. 487–492.
4. J. E. Hopcroft, "An *n* log *n* algorithm for minimizing the states in a finite automaton," in Z. Kohavi (ed.) *The Theory of Machines and Computations*, Academic Press, New York, pp. 189–196. (Хопкрофт Дж. Алгоритм для минимизации конечного автомата. — Кибернетический сборник, Новая серия, вып. 11. — М.: Мир, 1974. — С. 177–184.)
5. D. A. Huffman, "The synthesis of sequential switching circuits," *J. Franklin Inst.* **257**:3-4 (1954), pp. 161–190 and 275–303.
6. S. C. Kleene, "Representation of events in nerve nets and finite automata," in C. E. Shannon and J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956, pp. 3–42. (Клини С.К. Представление событий в нервных сетях. — сб. "*Автоматы*". — М.: ИЛ, 1956. — С. 15–67.)
7. E. F. Moore, "Gedanken experiments on sequential machines," in C. E. Shannon and J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956, pp. 129–153. (Мур Э.Ф. Умозрительные эксперименты с последовательностными машинами. — сб. "*Автоматы*". — М.: ИЛ, 1956. — С. 179–210.)
8. R. E. Stearns and J. Hartmanis, "Regularity-preserving modifications of regular expressions," *Information and Control* **6**:1 (1963), pp. 55–69.
9. J. I. Seiferas and R. McNaughton, "Regularity-preserving modifications," *Theoretical Computer Science* **2**:2 (1976), pp. 147–154.

ГЛАВА 5

Контекстно-свободные грамматики и языки

Перейдем от рассмотрения регулярных языков к более широкому классу языков, которые называются контекстно-свободными. Они имеют естественное рекурсивное описание в виде контекстно-свободных грамматик. Эти грамматики играют главную роль в технологии компиляции с начала 1960-х годов; они превратили непростую задачу реализации синтаксических анализаторов, распознающих структуру программы, из неформальной в рутинную, которую можно решить за один вечер. Позже контекстно-свободные грамматики стали использоваться для описания форматов документов в виде так называемых определений типа документов (document-type definition — DTD), которые применяются в языке XML (extensible markup language) для обмена информацией в Internet.

В этой главе определяется система записи контекстно-свободных грамматик и показывается, каким образом они определяют языки. Обсуждается понятие дерева разбора, изображающего структуру, которую грамматика налагает на цепочки языка. Дерево разбора представляет собой выход синтаксического анализатора языка программирования и одновременно общепринятый способ выражения структуры программы.

Контекстно-свободные языки также описываются с помощью магазинных автоматов, рассматриваемых в главе 6. Эти автоматы не столь важны, как конечные, однако в качестве средства определения языков они эквивалентны контекстно-свободным грамматикам и особенно полезны при изучении свойств замкнутости и разрешимости контекстно-свободных языков (глава 7).

5.1. Контекстно-свободные грамматики

Начнем с неформального представления контекстно-свободных грамматик, затем рассмотрим их некоторые важные свойства. Далее определим их формально и представим процесс “вывода”, с помощью которого грамматика задает цепочки языка.

5.1.1. Неформальный пример

Рассмотрим язык палиндромов. *Палиндром* — это цепочка, читаемая одинаково слева направо и наоборот, например, `otto` или `madamimadam` (“Madam, I'm Adam” — по преданию, первая фраза, услышанная Евой в Райском саду). Другими словами, цепочка w является палиндромом тогда и только тогда, когда $w = w^R$. Для упрощения рассмотрим

описание палиндромов только в алфавите {0, 1}. Этот язык включает цепочки вроде 0110, 11011, ε, но не включает 011 или 0101.

Нетрудно проверить, что язык L_{pal} палиндромов из символов 0 и 1 не является регулярным. Используем для этого лемму о накачке. Если язык L_{pal} регулярен, то пусть n — соответствующая константа из леммы. Рассмотрим палиндром $w = 0^n10^n$. Если L_{pal} регулярен, то w можно разбить на $w = xyz$ так, что y состоит из одного или нескольких нулей из их первой группы. Тогда в слове xz, которое также должно быть в L_{pal}, если L_{pal} регулярен, слева от единицы будет меньше нулей, чем справа. Следовательно, xz не может быть палиндромом, что противоречит предположению о регулярности L_{pal}.

Существует следующее естественное рекурсивное определение того, что цепочка из символов 0 и 1 принадлежит языку L_{pal}. Оно начинается с базиса, утверждающего, что несколько очевидных цепочек принадлежат L_{pal}, а затем использует идею того, что если цепочка является палиндромом, то она должна начинаться и заканчиваться одним и тем же символом. Кроме того, после удаления первого и последнего символа остаток цепочки также должен быть палиндромом.

Базис. ε, 0 и 1 являются палиндромами.

Индукция. Если w — палиндром, то $0w0$ и $1w1$ — также палиндромы. Ни одна цепочка не является палиндромом, если не определяется этими базисом и индукцией.

Контекстно-свободная грамматика представляет собой формальную запись подобных рекурсивных определений языков. Грамматика состоит из одной или нескольких переменных, которые представляют классы цепочек, или языки. В данном примере нужна только одна переменная, представляющая множество палиндромов, т.е. класс цепочек, образующих язык L_{pal}. Имеются правила построения цепочек каждого класса. При построении используются символы алфавита и уже построенные цепочки из различных классов.

Пример 5.1. Правила определения палиндромов, выраженные в виде контекстно-свободной грамматики, представлены на рис. 5.1. В разделе 5.1.2 мы рассмотрим их подробнее.

Первые три правила образуют базис. Они говорят, что класс палиндромов включает цепочки ε, 0 и 1. Эти правила образуют базис, поскольку ни одна из их правых частей (справа от стрелок) не содержит переменных.

Последние два правила образуют индуктивную часть определения. Например, правило 4 гласит, что если взять произвольную цепочку w из класса P, то $0w0$ также будет в классе P. Аналогично, по правилу 5 цепочка $1w1$ также будет в классе P. □

1. $P \to \varepsilon$
2. $P \to 0$
3. $P \to 1$
4. $P \to 0P0$
5. $P \to 1P1$

Рис. 5.1. Контекстно-свободная грамматика для палиндромов

5.1.2. Определение контекстно-свободных грамматик

Описание языка с помощью грамматики состоит из следующих четырех важных компонентов.

1. Есть конечное множество символов, из которых состоят цепочки определяемого языка. В примере о палиндромах это было множество {0, 1}. Его символы называются *терминальными*, или *терминалами*.
2. Существует конечное множество переменных, называемых иногда также *нетерминалами*, или *синтаксическими категориями*. Каждая переменная представляет язык, т.е. множество цепочек. В рассмотренном примере была только одна переменная, P, которая использовалась для представления класса палиндромов в алфавите {0, 1}.
3. Одна из переменных представляет определяемый язык; она называется *стартовым символом*. Другие переменные представляют дополнительные классы цепочек, которые помогают определить язык, заданный стартовым символом.
4. Имеется конечное множество *продукций*, или *правил вывода*, которые представляют рекурсивное определение языка. Каждая продукция состоит из следующих частей:
   а) переменная, (частично) определяемая продукцией. Эта переменная часто называется *головой* продукции;
   б) символ продукции $\rightarrow$;
   в) конечная цепочка, состоящая из терминалов и переменных, возможно, пустая. Она называется *телом* продукции и представляет способ образования цепочек языка, обозначаемого переменной в голове. По этому способу мы оставляем терминалы неизменными и вместо каждой переменной в теле подставляем любую цепочку, про которую известно, что она принадлежит языку этой переменной.

   Пример множества продукций приведен на рис. 5.1.

Описанные четыре компонента образуют *контекстно-свободную грамматику*, или *КС-грамматику*, или просто *грамматику*, или *КСГ*. Мы будем представлять КС-грамматику G ее четырьмя компонентами в виде $G = (V, T, P, S)$, где V — множество переменных, T — терминалов, P — множество продукций, S — стартовый символ.

Пример 5.2. Грамматика G_{pal} для палиндромов имеет вид

$$G_{pal} = (\{P\}, \{0, 1\}, A, P),$$

где A обозначает множество из пяти продукций (см. рис. 5.1). □

Пример 5.3. Рассмотрим более сложную КС-грамматику, которая представляет выражения типичного языка программирования (в несколько упрощенном виде). Во-первых, ограничимся операторами + и *, представляющими сложение и умножение. Во-вторых, допустим, что аргументы могут быть идентификаторами, однако не произвольными, т.е. буквами, за которыми следует любая последовательность букв и цифр, в том числе пустая. Допустим только буквы a и b и цифры 0 и 1. Каждый идентификатор должен начинаться с буквы a или b, за которой следует цепочка из $\{a, b, 0, 1\}^*$.

В нашей грамматике нужны две переменные. Одна обозначается E и представляет выражения определяемого языка. Она является стартовым символом. Другая переменная, I, представляет идентификаторы. Ее язык в действительности регулярен и задается регулярным выражением

$$(\mathbf{a} + \mathbf{b})(\mathbf{a} + \mathbf{b} + \mathbf{0} + \mathbf{1})^*.$$

В грамматиках, однако, регулярные выражения непосредственно не используются. Вместо этого будем обращаться к множеству продукций, задающих в точности то же самое, что и соответствующее регулярное выражение.

1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

Рис. 5.2. Контекстно-свободная грамматика для простых выражений

Грамматика для выражений определяется формально как $G = (\{E, I\}, T, P, E)$, где $T = \{+, *, (,), a, b, 0, 1\}$, а P представляет собой множество продукций, показанное на рис. 5.2. Продукции интерпретируются следующим образом.

Правило 1 является базисным для выражений. Оно гласит, что выражение может быть одиночным идентификатором. Правила 2–4 описывают индуктивное образование выражений. Правила 2 и 3 говорят, что выражение может состоять из двух выражений, соединенных знаком сложения или умножения. Правило 4 — что если заключить произвольное выражение в скобки, то в результате получается также выражение.

Сокращенное обозначение продукций

Продукцию удобно рассматривать как "принадлежащую" переменной в ее голове. Мы будем часто пользоваться словами "продукции для A" или "A-продукции" для обозначения продукций, головой которых является переменная A. Продукции грамматики можно записать, указав каждую переменную один раз и затем перечислив все тела продукций для этой переменной, разделяя их вертикальной чертой. Таким образом, продукции $A \to \alpha_1$, $A \to \alpha_2$, …, $A \to \alpha_n$ можно заменить записью $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$. Например, грамматику для палиндромов (см. рис. 5.1) можно записать в виде $P \to \varepsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1$.

Правила 5–10 описывают идентификаторы I. Правила 5 и 6 образуют базис; они гласят, что a и b — идентификаторы. Остальные четыре правила описывают индуктивный переход: имея произвольный идентификатор, можно приписать к нему справа a, b, 0 или 1 и получить еще один идентификатор. □

5.1.3. Порождения с использованием грамматики

Для того чтобы убедиться, что данные цепочки принадлежат языку некоторой переменной, мы применяем продукции КС-грамматики. Есть два способа действий. Простейший подход состоит в применении правил "от тела к голове". Мы берем цепочки, про которые известно, что они принадлежат языкам каждой из переменных в теле правила, записываем их в соответствующем порядке вместе с терминалами этого тела и убеждаемся, что полученная цепочка принадлежит языку переменной в голове. Такая процедура называется *рекурсивным выводом* (recursive inference).

Есть еще один способ определения языка грамматики, по которому продукции применяются "от головы к телу". Мы разворачиваем стартовый символ, используя одну из его продукций, т.е. продукцию, головой которой является этот символ. Затем разворачиваем полученную цепочку, заменяя одну из переменных телом одной из ее продукций, и так далее, пока не получим цепочку, состоящую из одних терминалов. Язык грамматики — это все цепочки из терминалов, получаемые таким способом. Такое использование грамматики называется *выведением*, или *порождением* (derivation).

Начнем с примера применения первого подхода — рекурсивного вывода, хотя часто естественнее рассматривать грамматики в качестве применяемых для порождений, и далее мы разовьем систему записи таких порождений.

Пример 5.4. Рассмотрим некоторые из выводов, которые можно сделать, используя грамматику для выражений (см. рис. 5.2). Результаты этих выводов показаны на рис. 5.3. Например, строчка (*i*) говорит, что в соответствии с продукцией 5 цепочка a принадлежит языку переменной I. Строчки (*ii*)–(*iv*) свидетельствуют, что цепочка $b00$ является идентификатором, полученным с помощью одного применения продукции 6 и затем двух применений продукции 9.

В строчках (*v*) и (*vi*) использована продукция 1 для того, чтобы прийти к заключению, что цепочки a и $b00$, которые выведены как идентификаторы в строчках (*i*) и (*iv*), принадлежат также и языку переменной E. В строчке (*vii*) применяется продукция 2 для вывода, что сумма этих идентификаторов является выражением, в строчке (*viii*) — продукция 4, и эта же сумма в скобках также представляет собой выражение. В строчке (*ix*) используется продукция 3 для умножения идентификатора a на выражение, исследованное в строчке (*viii*). □

Процесс порождения цепочек путем применения продукций "от головы к телу" требует определения нового символа отношения $\Rightarrow$. Пусть $G = (V, T, P, S)$ — КС-грамматика. Пусть $\alpha A \beta$ — цепочка из терминалов и переменных, где A — переменная. Таким

образом, α и β — цепочки из $(V \cup T)^*$, $A \in V$. Пусть $A \to \gamma$ — продукция из G. Тогда мы говорим, что $\alpha A \beta \underset{G}{\Rightarrow} \alpha\gamma\beta$. Если грамматика G подразумевается, то это записывается просто как $\alpha A \beta \Rightarrow \alpha\gamma\beta$. Заметим, что один шаг порождения заменяет любую переменную в цепочке телом одной из ее продукций.

	Выведенная цепочка	Для языка переменной	Использованная продукция	Использованные цепочки
(i)	*a*	*I*	5	—
(ii)	*b*	*I*	6	—
(iii)	*b*0	*I*	9	(ii)
(iv)	*b*00	*I*	9	(iii)
(v)	*a*	*E*	1	(i)
(vi)	*b*00	*E*	1	(iv)
(vii)	*a* + *b*00	*E*	2	(v), (vi)
(viii)	*(a* + *b*00*)*	*E*	4	(vii)
(ix)	*a*(a* + *b*00*)*	*E*	3	(v), (viii)

Рис. 5.3. Вывод цепочек с использованием грамматики, показанной на рис. 5.2

Для представления нуля, одного или нескольких шагов порождения можно расширить отношение $\Rightarrow$ подобно тому, как функция переходов δ расширялась до $\hat{\delta}$. Для обозначения нуля или нескольких шагов порождения используется символ *.

Базис. Для произвольной цепочки α терминалов и переменных считаем, что $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$. Таким образом, любая цепочка порождает саму себя.

Индукция. Если $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ и $\beta \underset{G}{\Rightarrow} \gamma$, то $\alpha \overset{*}{\underset{G}{\Rightarrow}} \gamma$. Таким образом, если α может породить β за нуль или несколько шагов, и из β еще за один шаг порождается γ, то α может породить γ. Иными словами, $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ означает, что для некоторого $n \geq 1$ существует последовательность цепочек $\gamma_1, \gamma_2, \ldots, \gamma_n$, удовлетворяющая следующим условиям.

1. $\alpha = \gamma_1$.
2. $\beta = \gamma_n$.
3. Для $i = 1, 2, \ldots, n-1$ имеет место отношение $\gamma_i \Rightarrow \gamma_{i+1}$.

Если грамматика G подразумевается, то вместо $\overset{*}{\underset{G}{\Rightarrow}}$ используется обозначение $\overset{*}{\Rightarrow}$.

Пример 5.5. Вывод о том, что $a*(a + b00)$ принадлежит языку переменной E, можно отразить в порождении этой цепочки, начиная с E. Запишем одно из таких порождений.

$$E \Rightarrow E * E \Rightarrow I * E \Rightarrow a * E \Rightarrow$$
$$a * (E) \Rightarrow a * (E + E) \Rightarrow a * (I + E) \Rightarrow a * (a + E) \Rightarrow$$
$$a * (a + I) \Rightarrow a * (a + I0) \Rightarrow a * (a + I00) \Rightarrow a * (a + b00)$$

На первом шаге E заменяется телом продукции 3 (см. рис. 5.2). На втором шаге применяется продукция 1 для изменения E на I и т.д. Заметим, что мы систематически придерживались тактики замены крайней слева переменной в цепочке. На каждом шаге, однако, мы можем произвольно выбирать переменную для замены и использовать любую из продукций для этой переменной. Например, на втором шаге мы могли бы изменить второе E на (E), используя продукцию 4. В этом случае $E * E \Rightarrow E * (E)$. Мы могли бы также выбрать замену, не приводящую к той же самой цепочке терминалов. Простым примером было бы использование продукции 2 на первом же шаге, в результате чего $E \Rightarrow E + E$, но теперь никакая замена переменных E не превратит цепочку $E + E$ в $a * (a + b00)$.

Мы можем использовать отношение $\overset{*}{\Rightarrow}$ для сокращения порождения. На основании базиса нам известно, что $E \overset{*}{\Rightarrow} E$. Несколько использований индукции дают нам утверждения $E \overset{*}{\Rightarrow} E * E$, $E \overset{*}{\Rightarrow} I * E$ и так далее до заключительного $E \overset{*}{\Rightarrow} a * (a + b00)$.

Две точки зрения — рекурсивный вывод и порождение — являются эквивалентными. Таким образом, можно заключить, что цепочка терминалов w принадлежит языку некоторой переменной A тогда и только тогда, когда $A \overset{*}{\Rightarrow} w$. Доказательство этого факта, однако, требует некоторых усилий, и мы отложим его до раздела 5.2. □

5.1.4. Левые и правые порождения

Для ограничения числа выборов в процессе порождения цепочки полезно потребовать, чтобы на каждом шаге мы заменяли крайнюю слева переменную одним из тел ее продукций. Такое порождение называется *левым* (leftmost), и для его указания используются отношения $\underset{lm}{\Rightarrow}$ и $\overset{*}{\underset{lm}{\Rightarrow}}$. Если используемая грамматика G не очевидна из контекста, то ее имя G также добавляется внизу.

Аналогично можно потребовать, чтобы на каждом шаге заменялась крайняя справа переменная на одно из тел. В таком случае порождение называется *правым* (rightmost), и для его обозначения используются символы $\underset{rm}{\Rightarrow}$ и $\overset{*}{\underset{rm}{\Rightarrow}}$. Имя используемой грамматики также при необходимости записывается внизу.

Пример 5.6. Порождение из примера 5.5 в действительности было левым, и его можно представить следующим образом.

$$E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} \mathrm{I} * \mathrm{E} \underset{lm}{\Rightarrow} \mathrm{a} * \mathrm{E} \underset{lm}{\Rightarrow}$$

$$a * (E) \underset{lm}{\Rightarrow} a * (E + E) \underset{lm}{\Rightarrow} a * (I + E) \underset{lm}{\Rightarrow} a * (a + E) \underset{lm}{\Rightarrow}$$
$$a * (a + I) \underset{lm}{\Rightarrow} a * (a + I0) \underset{lm}{\Rightarrow} a * (a + I00) \underset{lm}{\Rightarrow} a * (a + b00)$$

Обозначения для порождений, заданных КС-грамматиками

Существует немало соглашений, напоминающих о роли тех или иных символов, используемых в КС-грамматиках. Будем использовать следующие соглашения.

1. Строчные буквы из начала алфавита (*a*, *b* и т.д.) являются терминальными символами. Будем предполагать, что цифры и другие символы типа знака + или круглых скобок — также терминалы.
2. Прописные начальные буквы (*A*, *B* и т.д.) являются переменными.
3. Строчные буквы из конца алфавита, такие как *w* или *z*, обозначают цепочки терминалов. Это соглашение напоминает, что терминалы аналогичны входным символам конечного автомата.
4. Прописные буквы из конца алфавита, вроде *X* или *Y*, могут обозначать как терминалы, так и переменные.
5. Строчные греческие буквы (α, β и т.д.) обозначают цепочки, состоящие из терминалов и/или переменных.

Для цепочек, состоящих только из переменных, специального обозначения нет, поскольку это понятие не имеет большого значения. Вместе с тем, цепочка, обозначенная α или другой греческой буквой, возможно, состоит только из переменных.

Можно резюмировать левое порождение в виде $E \overset{*}{\underset{lm}{\Rightarrow}} a * (a + b00)$ или записать несколько его шагов в виде выражений типа $E * E \overset{*}{\underset{lm}{\Rightarrow}} a * (E)$.

Следующее правое порождение использует те же самые замены для каждой переменной, хотя и в другом порядке.

$$E \underset{rm}{\Rightarrow} E * E \underset{rm}{\Rightarrow} E * (E) \underset{rm}{\Rightarrow} E * (E + E) \underset{rm}{\Rightarrow}$$
$$E * (E + I) \underset{rm}{\Rightarrow} E * (E + I0) \underset{rm}{\Rightarrow} E * (E + I00) \underset{rm}{\Rightarrow} E * (E + b00) \underset{rm}{\Rightarrow}$$
$$E * (I + b00) \underset{rm}{\Rightarrow} E * (a + b00) \underset{rm}{\Rightarrow} I * (a + b00) \underset{rm}{\Rightarrow} a*(a + b00)$$

Данное порождение также позволяет заключить, что $E \overset{*}{\underset{rm}{\Rightarrow}} a * (a + b00)$.

Любое порождение имеет эквивалентные левое и правое порождения. Это означает, что если *w* — терминальная цепочка, а *A* — переменная, то $A \overset{*}{\Rightarrow} w$ тогда и только тогда, когда $A \overset{*}{\underset{lm}{\Rightarrow}} w$, а также $A \overset{*}{\Rightarrow} w$ тогда и только тогда, когда $A \overset{*}{\underset{rm}{\Rightarrow}} w$. Эти утверждения доказываются также в разделе 5.2.

5.1.5. Язык, задаваемый грамматикой

Если $G = (V, T, P, S)$ — КС-грамматика, то язык, *задаваемый грамматикой* G, или *язык грамматики* G, обозначается $L(G)$ и представляет собой множество терминальных цепочек, порождаемых из стартового символа. Таким образом,

$$L(G) = \{w \in T^* \mid S \overset{*}{\underset{G}{\Rightarrow}} w\}.$$

Если язык L задается некоторой КС-грамматикой, то он называется *контекстно-свободным*, или *КС-языком*. Например, мы утверждали, что грамматика, представленная на рис. 5.1, определяет множество палиндромов в алфавите {0, 1}. Таким образом, множество палиндромов является КС-языком. Можем доказать это утверждение следующим образом.

Теорема 5.7. Язык $L(G_{pal})$, где G_{pal} — грамматика из примера 5.1, является множеством палиндромов над {0, 1}.

Доказательство. Докажем, что цепочка w в $\{0, 1\}^*$ принадлежит $L(G_{pal})$ тогда и только тогда, когда она является палиндромом, т.е. $w = w^R$.

(*Достаточность*) Пусть w — палиндром. Докажем индукцией по $|w|$, что $w \in L(G_{pal})$.

Базис. Если $|w| = 0$ или $|w| = 1$, то w представляет собой ε, 0 или 1. Поскольку в грамматике есть продукции $P \to \varepsilon$, $P \to 0$, $P \to 1$, то во всех случаях $P \overset{*}{\Rightarrow} w$.

Индукция. Пусть $|w| \geq 2$. Так как $w = w^R$, она должна начинаться и заканчиваться одним и тем же символом, т.е. $w = 0x0$ или $w = 1x1$. Кроме того, x является палиндромом, т.е. $x = x^R$. Заметим, что нам необходим факт $|w| \geq 2$, чтобы обеспечить наличие двух различных нулей или единиц на обоих концах w.

Если $w = 0x0$, то по предположению индукции $P \overset{*}{\Rightarrow} x$. Тогда существует порождение w из P: $P \Rightarrow 0P0 \overset{*}{\Rightarrow} 0x0 = w$. Если $w = 1x1$, то рассуждения аналогичны, но с использованием продукции $P \to 1P1$ на первом шаге. В обоих случаях заключаем, что $w \in L(G_{pal})$, тем самым завершая доказательство.

(*Необходимость*) Предположим, что $w \in L(G_{pal})$, т.е. $P \overset{*}{\Rightarrow} w$. Докажем, что w — палиндром. Проведем доказательство индукцией по числу шагов в порождении w из P.

Базис. Если порождение имеет один шаг, то он использует одну из трех продукций, не имеющих P в теле, т.е. порождение представляет собой $P \Rightarrow \varepsilon$, $P \Rightarrow 0$ или $P \Rightarrow 1$. Так как ε, 0 и 1 — палиндромы, то базис доказан.

Индукция. Предположим, что порождение состоит из $n + 1$ шагов, где $n \geq 1$, и что утверждение индукции верно для всех порождений из n шагов. Таким образом, если $P \overset{*}{\Rightarrow} x$ за n шагов, то x является палиндромом.

Рассмотрим $(n + 1)$-шаговое порождение, которое должно иметь вид

$$P \Rightarrow 0P0 \overset{*}{\Rightarrow} 0x0 = w$$

или $P \Rightarrow 1P1 \overset{*}{\Rightarrow} 1x1 = w$, поскольку $n + 1$ шаг — это как минимум два шага, и только использование продукций $P \to 0P0$ или $P \to 1P1$ делает возможными дополнительные шаги порождения. Заметим, что в обоих случаях $P \overset{*}{\Rightarrow} x$ за n шагов.

По предположению индукции x является палиндромом, т.е. $x = x^R$. Но тогда и $0x0$, и $1x1$ — палиндромы. Например, $(0x0)^R = 0x^R0 = 0x0$. Делаем вывод, что w — палиндром и завершаем доказательство. □

5.1.6. Выводимые цепочки

Порождения из стартового символа грамматики приводят к цепочкам, имеющим особое значение. Они называются "выводимыми цепочками" ("sentential form"). Итак, если $G = (V, T, P, S)$ — КС-грамматика, то любая цепочка α из $(V \cup T)^*$, для которой $S \overset{*}{\Rightarrow} \alpha$, называется *выводимой цепочкой*. Если $S \underset{lm}{\overset{*}{\Rightarrow}} \alpha$, то α является *левовыводимой цепочкой*, а если $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha$, то — *правовыводимой*. Заметим, что язык $L(G)$ образуют выводимые цепочки из T^*, состоящие исключительно из терминалов.

Пример 5.8. Рассмотрим грамматику выражений (см. рис. 5.2). Например, $E * (I + E)$ является выводимой цепочкой, поскольку существует следующее порождение.

$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow E * (I + E)$$

Однако это порождение не является ни левым, ни правым, так как на последнем шаге заменяется среднее E.

Примером левовыводимой цепочки может служить $a * E$ со следующим левым порождением.

$$E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E$$

Аналогично, порождение

$$E \underset{rm}{\Rightarrow} E * E \underset{rm}{\Rightarrow} E * (E) \underset{rm}{\Rightarrow} E * (E + E)$$

показывает, что $E * (E + E)$ является правовыводимой цепочкой. □

Способ доказательства теорем о грамматиках

Теорема 5.7 типична для доказательств, показывающих, что та или иная грамматика задает некоторый неформально определенный язык. Сначала строится предположение индукции, говорящее о свойствах цепочек, порождаемых из каждой переменной. В данном примере была только одна переменная P, поэтому нам было достаточно доказать, что ее цепочки были палиндромами.

Доказывается достаточность: если цепочка w удовлетворяет неформальным утверждениям о цепочках переменной A, то $A \stackrel{*}{\Rightarrow} w$. В нашем примере, поскольку P является стартовым символом, мы утверждали "$P \stackrel{*}{\Rightarrow} w$", говоря, что w принадлежит языку грамматики. Обычно достаточность доказывается индукцией по длине слова w. Если в грамматике k переменных, то индуктивное утверждение имеет k частей, которые доказываются с помощью взаимной индукции.

Нужно доказать также необходимость, т.е. что если $A \stackrel{*}{\Rightarrow} w$, то w удовлетворяет неформальным утверждениям о цепочках, порождаемых из переменной A. Поскольку в нашем примере был только стартовый символ P, предположение о том, что $w \in L(G_{pal})$, было равносильно $P \stackrel{*}{\Rightarrow} w$. Для доказательства этой части типична индукция по числу шагов порождения. Если в грамматике есть продукции, позволяющие нескольким переменным появляться в порождаемых цепочках, то n-шаговое порождение нужно разбить на несколько частей, по одному порождению из каждой переменной. Эти порождения могут иметь менее n шагов, поэтому следует предполагать утверждение индукции верным для всех значений, которые не больше n, как это обсуждалось в разделе 1.4.2.

5.1.7. Упражнения к разделу 5.1

5.1.1. Построить КС-грамматики для следующих языков:

а) (*****) множество $\{0^n1^n \mid n \geq 1\}$ всех цепочек из одного и более символов 0, за которыми следуют символы 1 в таком же количестве;

б) (***!**) множество $\{a^ib^jc^k \mid i \neq j$ или $j \neq k\}$ всех цепочек из символов a, за которыми следуют символы b и затем c так, что количества символов a и b или количества b и c различны;

в) (**!**) множество всех цепочек из символов a и b, не имеющих вида ww, т.е. не равных ни одной цепочке-повторению;

г) (**!!**) множество всех цепочек, у которых символов 0 вдвое больше, чем символов 1.

5.1.2. Следующая грамматика порождает язык регулярного выражения $\mathbf{0^*1(0 + 1)^*}$.

$S \to A1B$
$A \to 0A \mid \varepsilon$
$B \to 0B \mid 1B \mid \varepsilon$

Запишите левое и правое порождения следующих цепочек:

а) 00101;

б) 1001;

в) 00011.

5.1.3. Докажите, что каждый регулярный язык является КС-языком. *Указание*. Постройте КС-грамматику с помощью индукции по числу операторов в регулярном выражении.

5.1.4. КС-грамматика называется *праволинейной*, если тело каждой продукции имеет не более одной переменной, причем она находится на правом краю тела. Таким образом, продукции праволинейной грамматики имеют вид $A \to wB$ или $A \to w$, где A и B — переменные, а w — терминальная цепочка, возможно, пустая:

а) докажите, что каждая праволинейная грамматика порождает регулярный язык. *Указание*. Постройте ε-НКА, который имитирует левые порождения, представляя своим состоянием единственную переменную в текущей левовыводимой цепочке;

б) докажите, что каждый регулярный язык имеет праволинейную грамматику. *Указание*. Начните с ДКА, состояния которого представляются переменными грамматики.

5.1.5. (***!**) Пусть $T = \{0, 1, (,), +, *, \varnothing, e\}$. T можно рассматривать как множество символов, используемых в регулярных выражениях над алфавитом $\{0, 1\}$. Единственная разница состоит в том, что во избежание возможной путаницы вместо символа ε используется символ e. Постройте КС-грамматику со множеством терминалов T, которая порождает в точности регулярные выражения в алфавите $\{0, 1\}$.

5.1.6. Отношение $\overset{*}{\Rightarrow}$ было определено с базисом "$\alpha \Rightarrow \alpha$" и индукцией, утверждавшей: "из $\alpha \overset{*}{\Rightarrow} \beta$ и $\beta \Rightarrow \gamma$ следует $\alpha \overset{*}{\Rightarrow} \gamma$". Есть несколько других способов определения отношения $\overset{*}{\Rightarrow}$, также равнозначных фразе: "$\overset{*}{\Rightarrow}$ есть нуль или несколько шагов отношения $\Rightarrow$". Докажите следующие утверждения:

а) $\alpha \overset{*}{\Rightarrow} \beta$ тогда и только тогда, когда существует последовательность из одной или нескольких цепочек $\gamma_1, \gamma_2, \ldots, \gamma_n$ где $\alpha = \gamma_1$, $\beta = \gamma_n$ и для $i = 1, 2, \ldots, n - 1$ имеет место $\gamma_i \Rightarrow \gamma_{i+1}$;

б) если $\alpha \overset{*}{\Rightarrow} \beta$ и $\beta \overset{*}{\Rightarrow} \gamma$, то $\alpha \overset{*}{\Rightarrow} \gamma$. *Указание*. Воспользуйтесь индукцией по числу шагов в порождении $\beta \overset{*}{\Rightarrow} \gamma$.

5.1.7. (**!**) Рассмотрим КС-грамматику G, определяемую следующими продукциями:

$S \to aS \mid Sb \mid a \mid b$

а) докажите индукцией по длине цепочки, что ни одна цепочка в $L(G)$ не содержит ba как подцепочку;

б) дайте неформальное описание $L(G)$. Уточните ответ, используя часть (а).

5.1.8. (!!) Рассмотрим КС-грамматику G, определяемую следующими продукциями.

$S \to aSbS \mid bSaS \mid \varepsilon$

Докажите, что $L(G)$ представляет собой множество всех цепочек, в которых поровну символов a и b.

5.2. Деревья разбора

Для порождений существует чрезвычайно полезное представление в виде дерева. Это дерево наглядно показывает, каким образом символы цепочки группируются в подцепочки, каждая из которых принадлежит языку одной из переменных грамматики. Возможно, более важно то, что дерево, известное в компиляции как "дерево разбора", является основной структурой данных для представления исходной программы. В компиляторе древовидная структура исходной программы облегчает ее трансляцию в исполняемый код за счет того, что допускает естественные рекурсивные функции для выполнения этой трансляции.

В данном разделе представлено понятие дерева разбора и показано, что существование дерева разбора тесно связано с существованием порождений и рекурсивных выводов. Далее изучается сущность неоднозначности в грамматиках и языках, являющейся важным свойством деревьев разбора. Некоторые грамматики допускают, что терминальная цепочка имеет несколько деревьев разбора. Такое свойство делает грамматику непригодной для описания языков программирования, поскольку в этом случае компилятор не мог бы распознать структуру некоторых исходных программ, и, как следствие, не мог бы однозначно определить исполняемый код, соответствующий программе.

5.2.1. Построение деревьев разбора

Будем рассматривать грамматику $G = (V, T, P, S)$. *Деревья разбора* для G — это деревья со следующими свойствами.

1. Каждый внутренний узел отмечен переменной из V.
2. Каждый лист отмечен либо переменной, либо терминалом, либо ε. При этом, если лист отмечен ε, он должен быть единственным сыном своего родителя.
3. Если внутренний узел отмечен A, и его сыновья отмечены слева направо $X_1, X_2, \ldots, X_k$, соответственно, то $A \to X_1X_2 \cdots X_k$ является продукцией в P. Отметим, что X может быть ε лишь в одном случае — если он отмечает единственного сына, и $A \to \varepsilon$ — продукция грамматики G.

Обзор терминов, связанных с деревьями

Мы предполагаем, что читатель знаком с понятием дерева и основными определениями для деревьев. Тем не менее, напомним их вкратце.

- Деревья представляют собой множества узлов с отношением *родитель-сын*. Узел имеет не более одного родителя, изображаемого над узлом, и нуль или несколько сыновей, изображаемых под ним. Родителей и их сыновей соединяют линии. Примеры деревьев представлены на рис. 5.4–5.6.
- Один узел, *корень*, не имеет родителя; он появляется на вершине дерева. Узлы без сыновей называются *листьями*. Узлы, не являющиеся листьями, называются *внутренними узлами*.
- Сын сына и так далее узла называется его *потомком*; соответственно, родитель родителя и так далее — *предком*. Узлы считаются потомками и предками самих себя.
- Сыновья узла упорядочиваются слева направо и изображаются в этом порядке. Если узел N находится слева от узла M, то считается, что все потомки узла N находятся слева от всех потомков M.

Пример 5.9. На рис. 5.4 показано дерево разбора, которое использует грамматику выражений (см. рис. 5.2). Корень отмечен переменной E. В корне применена продукция $E \to E + E$, поскольку три сына корня отмечены слева направо как E, +, E. В левом сыне корня применена продукция $E \to I$, так как у этого узла один сын, отмеченный переменной I. □

Пример 5.10. На рис. 5.5 показано дерево разбора для грамматики палиндромов (см. рис. 5.1). В корне применена продукция $P \to 0P0$, а в среднем сыне корня — $P \to 1P1$. Отметим, что внизу использована продукция $P \to \varepsilon$. Это использование, при котором у узла есть сын с отметкой ε, является единственным случаем, когда в дереве может быть узел, отмеченный ε. □

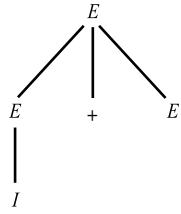


Рис. 5.4. Дерево разбора, показывающее порождение I + E *из* E

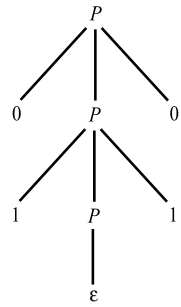


Рис. 5.5. Дерево разбора, показывающее порождение $P \overset{*}{\Rightarrow} 0110$

5.2.2. Крона дерева разбора

Если мы посмотрим на листья любого дерева разбора и выпишем их отметки слева направо, то получим цепочку, которая называется *кроной дерева* и всегда является цепочкой, выводимой из переменной, отмечающей корень. Утверждение о том, что крона выводима из отметки корня, будет доказано далее. Особый интерес представляют деревья разбора со следующими свойствами.

1. Крона является терминальной цепочкой, т.е. все листья отмечены терминалами или ε.
2. Корень отмечен стартовым символом.

Кроны таких деревьев разбора представляют собой цепочки языка рассматриваемой грамматики. Мы докажем также, что еще один способ описания языка грамматики состоит в определении его как множества крон тех деревьев разбора, у которых корень отмечен стартовым символом, а крона является терминальной цепочкой.

Пример 5.11. На рис. 5.6 представлен пример дерева с терминальной цепочкой в качестве кроны и стартовым символом в корне. Оно основано на грамматике для выражений (см. рис. 5.2). Крона этого дерева образует цепочку $a * (a + b00)$, выведенную в примере 5.6. В действительности, как мы увидим далее, это дерево разбора представляет порождение данной цепочки. □

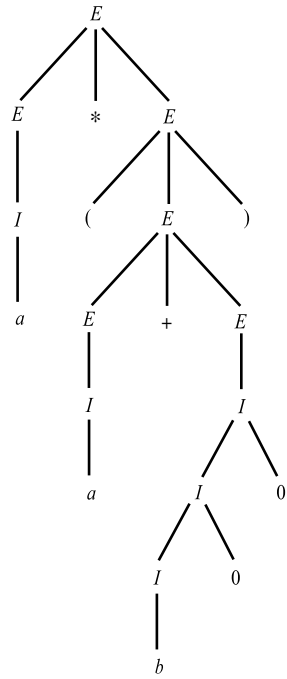


Рис. 5.6. Дерево разбора для a * (a + b*00*) *в языке для грамматики выражений*

5.2.3. Вывод, порождение и деревья разбора

Каждый из способов, определенных ранее для описания работы грамматики, приводит по существу к одним и тем же утверждениям о цепочках. Итак, покажем, что при любой грамматике $G = (V, T, P, S)$ следующие утверждения равносильны.

1. Процедура рекурсивного вывода определяет, что цепочка w принадлежит языку переменной A.
2. $A \overset{*}{\Rightarrow} w$.
3. $A \underset{lm}{\overset{*}{\Rightarrow}} w$.
4. $A \underset{rm}{\overset{*}{\Rightarrow}} w$.
5. Существует дерево разбора с корнем A и кроной w.

В действительности, за исключением использования рекурсивного вывода, определенного только для терминальных цепочек, все остальные условия (существование порождений, левых и правых порождений или деревьев разбора) также равносильны, если w имеет переменные.

Указанные равносильности доказываются в соответствии с планом, приведенным на рис. 5.7. Для каждой стрелки в этой диаграмме доказывается теорема, которая утверждает, что если w удовлетворяет условию в начале стрелки, то удовлетворяет и условию в ее конце. Например, мы покажем в теореме 5.12, что если w принадлежит языку A в соответствии с рекурсивным выводом, то существует дерево разбора с корнем A и кроной w.

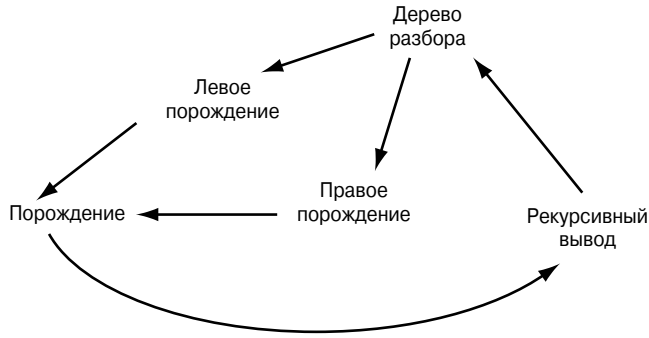


Рис. 5.7. Доказательство равносильности утверждений о грамматиках

Отметим, что две стрелки весьма просты и не будут обоснованы формально. Если цепочка w имеет левое порождение из A, то она безусловно порождается из A, поскольку левое порождение является порождением. Аналогично, если цепочка w имеет правое порождение, то она имеет и порождение. Приступим к доказательству более трудных шагов описанной равносильности.

5.2.4. От выводов к деревьям разбора

Теорема 5.12. Пусть $G = (V, T, P, S)$ — КС-грамматика. Если рекурсивный вывод утверждает, что терминальная цепочка w принадлежит языку переменной A, то существует дерево разбора с корнем A и кроной w.

Доказательство. Проведем доказательство индукцией по числу шагов, используемых в выводе того, что w принадлежит языку A.

Базис. Вывод содержит один шаг. В этом случае использован только базис процедуры вывода, и в грамматике должна быть продукция $A \to w$. В дереве на рис. 5.8 существует лист для каждого символа цепочки w, оно удовлетворяет условиям, налагаемым на дерево разбора для грамматики G, и, очевидно, имеет корень A и крону w. В особом случае, когда $w = \varepsilon$, дерево имеет только один лист, отмеченный ε, и является допустимым деревом разбора с корнем A и кроной w.

Индукция. Предположим, что факт принадлежности w языку переменной A выводится за $n + 1$ шаг, и что утверждение теоремы справедливо для всех цепочек x и переменных B, для которых принадлежность x языку B была выведена с использованием n или менее шагов вывода. Рассмотрим последний шаг вывода того, что w принадлежит языку A. Этот вывод использует некоторую продукцию для A, например $A \to X_1X_2\ldots X_k$, где каждый X_i есть либо переменная, либо терминал.

Цепочку w можно разбить на подцепочки $w_1w_2\ldots w_k$, для которых справедливо следующее.

1. Если X_i является терминалом, то $w_i = X_i$, т.е. w_i представляет собой одинединственный терминал из продукции.
2. Если X_i — переменная, то w_i представляет собой цепочку, о которой уже сделан вывод, что она принадлежит языку переменной X_i. Таким образом, этот вывод относительно w_i содержит не более n из $n + 1$ шагов вывода, что w принадлежит языку A. Этот вывод не может содержать все $n + 1$ шагов, поскольку заключительный шаг, использующий продукцию $A \to X_1X_2\ldots X_k$, безусловно, не является частью вывода относительно w_i. Следовательно, мы можем применить предположение индукции к w_i и заключить, что существует дерево разбора с кроной w_i и корнем X_i.

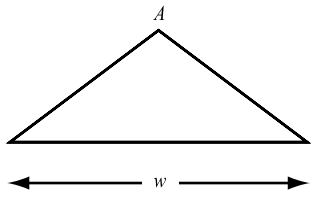


Рис. 5.8. Дерево, построенное для базиса теоремы 5.12

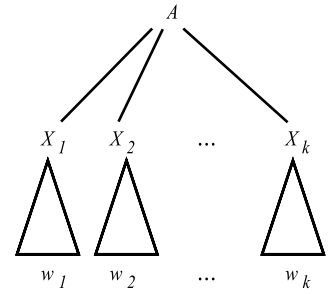


Рис. 5.9. Дерево, использованное в индуктивной части доказательства теоремы 5.12

Далее мы построим дерево с корнем A и кроной w в соответствии с рис. 5.9. Там показан корень с отметкой A и сыновьями $X_1, X_2, \ldots, X_k$. Такой выбор отметок возможен, поскольку $A \to X_1X_2\ldots X_k$ является продукцией грамматики G.

Узел для каждого X_i становится корнем поддерева с кроной w_i. В ситуации 1, где X_i — терминал, это поддерево состоит из одного листа, отмеченного X_i. Так как в данной ситуации $w_i = X_i$, условия того, что кроной поддерева является w_i, выполнены.

Во второй ситуации X_i является переменной. Тогда по предположению индукции существует дерево с корнем X_i и кроной w_i. Оно присоединяется к узлу для X_i (см. рис. 5.9).

Построенное таким образом дерево имеет корень A. Его крона состоит из крон поддеревьев, приписанных друг к другу слева направо, т.е. $w = w_1w_2\ldots w_k$. □

5.2.5. От деревьев к порождениям

Покажем, как построить левое порождение по дереву разбора (метод построения правого порождения аналогичен и не приводится). Для того чтобы понять, каким образом можно построить левое порождение, сначала нужно увидеть, как одно порождение цепочки из переменной можно вставить внутрь другого порождения. Проиллюстрируем это на примере.

Пример 5.13. Рассмотрим еще раз грамматику выражений (см. рис. 5.2). Нетрудно убедиться, что существует следующее порождение.

$$E \Rightarrow I \Rightarrow Ib \Rightarrow ab$$

Отсюда для произвольных цепочек α и β возможно следующее порождение.

$$\alpha E\beta \Rightarrow \alpha I\beta \Rightarrow \alpha Ib\beta \Rightarrow \alpha ab\beta$$

Доказательством служит то, что головы продукций можно заменять их телами в контексте α и β точно так же, как и вне его.[1]

Например, если порождение начинается заменами $E \Rightarrow E + E \Rightarrow E + (E)$, то можно было бы применить порождение цепочки ab из второго E, рассматривая "E+(" в качестве α и ")" — β. Указанное порождение затем продолжалось бы следующим образом.

$$E + (E) \Rightarrow E + (I) \Rightarrow E + (Ib) \Rightarrow E + (ab)$$

□

Теперь можно доказать теорему, позволяющую преобразовывать дерево разбора в левое порождение. Доказательство проводится индукцией по *высоте* дерева, которая представля-

[1] В действительности, именно эта возможность подстановки строки вместо переменной независимо от контекста и породила название "контекстно-свободная" (context-free). Существует более мощный класс грамматик, называемых "контекстно-зависимыми" (context-sensitive), в которых подстановки разрешены, только если определенные строки находятся слева и/или справа от заменяемой переменной. В современной практике контекстно-зависимые грамматики большого значения не имеют.

ет собой максимальную из длин путей, ведущих от корня через его потомков к листьям. Например, высота дерева, изображенного на рис. 5.6, равна 7. Самый длинный из путей от корня к листьям ведет к листу, отмеченному b. Заметим, что длины путей обычно учитывают ребра, а не узлы, поэтому путь, состоящий из единственного узла, имеет длину 0.

Теорема 5.14. Пусть $G = (V, T, P, S)$ — КС-грамматика. Предположим, что существует дерево разбора с корнем, отмеченным A, и кроной w, где $w \in T^*$. Тогда в грамматике G существует левое порождение $A \underset{lm}{\overset{*}{\Rightarrow}} w$.

Доказательство. Используем индукцию по высоте дерева.

Базис. Базисом является высота 1, наименьшая из возможных для дерева разбора с терминальной кроной. Дерево должно выглядеть, как на рис. 5.8, с корнем, отмеченным A, и сыновьями, образующими цепочку w. Поскольку это дерево является деревом разбора, $A \to w$ должно быть продукцией. Таким образом, $A \underset{lm}{\Rightarrow} w$ есть одношаговое левое порождение w из A.

Индукция. Если высота дерева равна n, где $n > 1$, то оно должно иметь вид, как на рис. 5.9. Таким образом, существует корень с отметкой A и сыновьями, отмеченными слева направо $X_1X_2\ldots X_k$. Символы X могут быть как терминалами, так и переменными.

1. Если X_i — терминал, то определим w_i как цепочку, состоящую из одного X_i.
2. Если X_i — переменная, то она должна быть корнем некоторого поддерева с терминальной кроной, которую обозначим w_i. Заметим, что в этом случае высота поддерева меньше n, поэтому к нему применимо предположение индукции. Следовательно, существует левое порождение $X_i \underset{lm}{\overset{*}{\Rightarrow}} w_i$.

Заметим, что $w = w_1w_2\ldots w_k$.

Построим левое порождение цепочки w следующим образом. Начнем с шага $A \underset{lm}{\Rightarrow} X_1X_2\ldots X_k$. Затем для $i = 1, 2, \ldots, k$ покажем, что имеет место следующее порождение.

$$A \underset{lm}{\overset{*}{\Rightarrow}} w_1w_2\ldots w_iX_{i+1}X_{i+2}\ldots X_k$$

Данное доказательство использует в действительности еще одну индукцию, на этот раз по i. Для базиса $i = 0$ мы уже знаем, что $A \underset{lm}{\Rightarrow} X_1X_2\ldots X_k$. Для индукции предположим, что существует следующее порождение.

$$A \underset{lm}{\overset{*}{\Rightarrow}} w_1w_2\ldots w_{i-1}X_iX_{i+1}\ldots X_k$$

1. Если X_i — терминал, то не делаем ничего, но в дальнейшем рассматриваем X_i как терминальную цепочку w_i. Таким образом, приходим к существованию следующего порождения.

$$A \underset{lm}{\overset{*}{\Rightarrow}} w_1 w_2 \ldots w_i X_{i+1} X_{i+2} \ldots X_k$$

2. Если X_i является переменной, то продолжаем порождением w_i из X_i в контексте уже построенного порождения. Таким образом, если этим порождением является

$$X_i \underset{lm}{\Rightarrow} \alpha_1 \underset{lm}{\Rightarrow} \alpha_2 \ldots \underset{lm}{\Rightarrow} w_i,$$

то продолжаем следующими порождениями.

$$w_1 w_2 \ldots w_{i-1} X_i X_{i+1} \ldots X_k \underset{lm}{\Rightarrow}$$
$$w_1 w_2 \ldots w_{i-1} \alpha_1 X_{i+1} \ldots X_k \underset{lm}{\Rightarrow}$$
$$w_1 w_2 \ldots w_{i-1} \alpha_2 X_{i+1} \ldots X_k \underset{lm}{\Rightarrow}$$
$$\ldots$$
$$w_1 w_2 \ldots w_i X_{i+1} X_{i+2} \ldots X_k$$

Результатом является порождение $A \underset{lm}{\overset{*}{\Rightarrow}} w_1 w_2 \ldots w_i X_{i+1} X_{i+2} \ldots X_k$.

Когда $i = k$, результат представляет собой левое порождение w из A. □

Пример 5.15. Рассмотрим левое порождение для дерева, изображенного на рис. 5.6. Продемонстрируем лишь заключительный шаг, на котором строится порождение по целому дереву из порождений, соответствующих поддеревьям корня. Итак, предположим, что с помощью рекурсивного применения техники из теоремы **5.14** мы убедились, что поддерево, корнем которого является левый сын корня дерева, имеет левое порождение $E \underset{lm}{\Rightarrow} I \underset{lm}{\Rightarrow} a$, а поддерево с корнем в третьем сыне корня имеет следующее левое порождение.

$$E \underset{lm}{\Rightarrow} (E) \underset{lm}{\Rightarrow} (E + E) \underset{lm}{\Rightarrow} (I + E) \underset{lm}{\Rightarrow} (a + E) \underset{lm}{\Rightarrow}$$
$$(a + I) \underset{lm}{\Rightarrow} (a + I0) \underset{lm}{\Rightarrow} (a + I00) \underset{lm}{\Rightarrow} (a + b00)$$

Чтобы построить левое порождение для целого дерева, начинаем с шага в корне: $A \underset{lm}{\Rightarrow} E * E$. Затем заменяем первую переменную E и в соответствии с ее порождением приписываем на каждом шаге $*E$, чтобы учесть контекст, в котором это порождение используется. Левое порождение на текущий момент представляет собой следующее.

$$E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E$$

Символ * в продукции, использованной в корне, не требует порождения, поэтому указанное левое порождение также учитывает первые два сына корня. Дополним левое порождение, используя порождение $E \underset{lm}{\overset{*}{\Rightarrow}} (a + b00)$, левый контекст которого образован цепочкой $a*$, а правый пуст. Это порождение в действительности появлялось в примере 5.6 и имело следующий вид.

$$E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E \underset{lm}{\Rightarrow}$$
$$a * (E) \underset{lm}{\Rightarrow} a * (E + E) \underset{lm}{\Rightarrow} a * (I + E) \underset{lm}{\Rightarrow} a * (a + E) \underset{lm}{\Rightarrow}$$
$$a * (a + I) \underset{lm}{\Rightarrow} a * (a + I0) \underset{lm}{\Rightarrow} a * (a + I00) \underset{lm}{\Rightarrow} a * (a + b00)$$

Аналогичная теорема позволяет нам преобразовать дерево в правое порождение. Построение правого порождения по дереву почти такое же, как и построение левого. Здесь, однако, после первого шага $A \underset{rm}{\Rightarrow} X_1X_2\ldots X_k$ мы заменяем сначала X_k, используя правое порождение, затем X_{k-1} и так далее до X_1. Таким образом, примем без доказательства следующее утверждение.

Теорема 5.16. Пусть $G = (V, T, P, S)$ — КС-грамматика. Предположим, что существует дерево разбора с корнем, отмеченным A, и кроной w, где $w \in T^*$. Тогда в грамматике G существует правое порождение $A \overset{*}{\underset{rm}{\Rightarrow}} w$.

5.2.6. От порождений к рекурсивным выводам

Теперь завершим цикл, представленный на рис. 5.7, доказав, что если существует порождение $A \overset{*}{\Rightarrow} w$ для некоторой КС-грамматики, то факт принадлежности w языку A доказывается путем процедуры рекурсивного вывода. Перед тем как приводить теорему и ее доказательство, сделаем важные замечания о порождениях.

Предположим, у нас есть порождение $A \Rightarrow X_1X_2\ldots X_k \overset{*}{\Rightarrow} w$. Тогда w можно разбить на подцепочки $w = w_1w_2\ldots w_k$, где $X_i \overset{*}{\Rightarrow} w_i$. Заметим, что если X_i является терминалом, то $w_i = X_i$, и порождение имеет 0 шагов. Доказать это замечание несложно. Вы можете доказать индукцией по числу шагов порождения, что если $X_1X_2\ldots X_k \overset{*}{\Rightarrow} \alpha$, то все позиции в α, происходящие от расширения X_i, находятся слева от всех позиций, происходящих от расширения X_j, если $i < j$.

Если X_i является переменной, то можно получить порождение $X_i \overset{*}{\Rightarrow} w_i$, начав с порождения $A \overset{*}{\Rightarrow} w$ и отбрасывая следующее:

а) все шаги, не относящиеся к порождению w_i из X_i;

б) все позиции выводимой цепочки, которые находятся либо справа, либо слева от позиций, порождаемых из X_i.

Этот процесс поясняется примером.

Пример 5.17. Используем грамматику выражений и рассмотрим следующее порождение.

$$E \Rightarrow E * E \Rightarrow E * E + E \Rightarrow I * E + E \Rightarrow I * I + E \Rightarrow$$
$$I * I + I \Rightarrow a * I + I \Rightarrow a * b + I \Rightarrow a * b + a$$

Рассмотрим третью выводимую цепочку, $E * E + E$, и среднее E в ней.[2]

Начав с $E * E + E$, можно пройти по шагам указанного выше порождения, выбрасывая позиции, порожденные из $E*$ слева от центрального E и из $+E$ справа от него. Шагами порождения тогда становятся E, E, I, I, I, b, b. Таким образом, следующий шаг не меняет центральное E, следующий за ним меняет его на I, два шага за ними сохраняют I, следующий меняет его на b, и заключительный шаг не изменяет того, что порождено из центрального E.

Если мы рассмотрим только шаги, которые изменяют то, что порождается из центрального E, то последовательность E, E, I, I, I, b, b превращается в порождение $E \Rightarrow I \Rightarrow b$. Оно корректно описывает, как центральное E эволюционирует в полном порождении. □

Теорема 5.18. Пусть $G = (V, T, P, S)$ — КС-грамматика, и пусть существует порождение $A \overset{*}{\underset{G}{\Rightarrow}} w$, где $w \in T$. Тогда процедура рекурсивного вывода, примененная к G, определяет, что w принадлежит языку переменной A.

Доказательство. Доказательство проведем индукцией по длине порождения $A \overset{*}{\Rightarrow} w$.

Базис. Если порождение состоит из одного шага, то $A \to w$ должно быть продукцией. Так как w состоит только из терминалов, то факт принадлежности w языку A устанавливается на основе базисной части процедуры рекурсивного вывода.

Индукция. Пусть порождение состоит из $n + 1$ шагов и пусть для любого порождения из n и менее шагов утверждение выполняется. Запишем порождение в виде $A \Rightarrow X_1X_2\ldots X_k \overset{*}{\Rightarrow} w$. Тогда, как обсуждалось перед теоремой, можно представить w как $w = w_1w_2\ldots w_k$, где:

а) если X_i — терминал, то $w_i = X_i$;

б) если X_i — переменная, то $X_i \overset{*}{\Rightarrow} w_i$. Так как первый шаг порождения $A \overset{*}{\Rightarrow} w$ действительно не является частью порождения $X_i \overset{*}{\Rightarrow} w_i$, нам известно, что это порождение состоит из n или менее шагов. Таким образом, к нему применимо предположение индукции, и можно сделать вывод, что w принадлежит языку X_i.

Теперь у нас есть продукция $A \to X_1X_2\ldots X_k$, и нам известно, что w_i или равно X_i, или принадлежит языку X_i. На следующем шаге процедуры рекурсивного вывода мы обнаружим, что $w_1w_2\ldots w_k$ принадлежит языку A. Так как $w_1w_2\ldots w_k = w$, выводимость того, что $w \in L(A)$, доказана. □

---

[2] Наше обсуждение нахождения подпорождений из больших порождений предполагало, что мы имели дело с переменными второй выводимой цепочки некоторого порождения. Однако идея применима к переменной на любом шаге порождения.

5.2.7. Упражнения к разделу 5.2

5.2.1. Приведите деревья разбора для грамматики и каждой из цепочек в упражнении 5.1.2.

5.2.2. Пусть G — КС-грамматика без продукций с ε в правой части. Доказать, что если $w \in L(G)$, длина w равна n, и w порождается за m шагов, то для w существует дерево разбора с $n + m$ узлами.

5.2.3. Пусть действуют все предположения упражнения 5.2.2, но G может иметь несколько продукций с ε справа. Доказать, что дерево разбора для w может иметь до $n + 2m - 1$ узлов, но не более.

5.2.4. В разделе 5.2.6 мы заметили, что если $X_1X_2\ldots X_k \stackrel{*}{\Rightarrow} \alpha$, то все позиции в α, происходящие от расширения X_i, находятся слева от всех позиций, происходящих от расширения X_j, если $i < j$. Доказать этот факт. *Указание.* Провести индукцию по числу шагов в порождении.

5.3. Приложения контекстно-свободных грамматик

Контекстно-свободные грамматики были придуманы Н. Хомским (N. Chomsky) как способ описания естественных языков, но их оказалось недостаточно. Однако по мере того, как множились примеры использования рекурсивно определяемых понятий, возрастала и потребность в КС-грамматиках как в способе описания примеров таких понятий. Мы рассмотрим вкратце два применения КС-грамматик, одно старое и одно новое.

1. Грамматики используются для описания языков программирования. Более важно здесь то, что существует механический способ превращения описания языка, вроде КС-грамматики, в синтаксический анализатор — часть компилятора, которая изучает структуру исходной программы и представляет ее с помощью дерева разбора. Это приложение является одним из самых ранних использований КС-грамматик; в действительности, это один из первых путей, по которым теоретические идеи компьютерной науки пришли в практику.
2. Развитие XML (Extensible Markup Language) призвано облегчить электронную коммерцию тем, что ее участникам доступны соглашения о форматах ордеров, описаний товаров, и многих других видов документов. Существенной частью XML является *определение типа документа* (DTD — Document Type Defifnition), представляющее собой КС-грамматику, которая описывает допустимые дескрипторы (tags) и способы их вложения друг в друга. Дескрипторы являются привычными ключевыми словами в угловых скобках, которые читателю, возможно, известны по языку HTML, например, <EM> и </EM> для указания текста, который нужно выделить. Однако дескрипторы XML связаны не с форматированием текста, а с тем, что он означает. Например, можно было бы заключить в скобки <PHONE> и </PHONE> последовательности символов, интерпретируемые как телефонные номера.

5.3.1. Синтаксические анализаторы

Многие объекты языка программирования имеют структуру, которая может быть описана с помощью регулярных выражений. Например, мы обсуждали в примере 3.9, как идентификаторы можно представлять регулярными выражениями. Вместе с тем, существует также несколько весьма важных объектов в языках программирования, которые нельзя представить с помощью только лишь регулярных выражений. Приведем два примера.

Пример 5.19. Обычные языки программирования используют круглые и/или квадратные скобки во вложенном и сбалансированном виде, т.е. так, что можно некоторой левой скобке поставить в соответствие правую, которая записана непосредственно за ней, удалить их и повторять эти действия вплоть до удаления всех скобок. Например, (()), ()(), (()()) и ε являются сбалансированными скобками, а)(и (() — нет.

Все цепочки сбалансированных скобок (и только они) порождаются грамматикой $G_{bal} = (\{B\}, \{(,)\}, P, B)$, где P состоит из продукций

$$B \to BB \mid (B) \mid \varepsilon.$$

Первая продукция, $B \to BB$, гласит, что конкатенация двух цепочек сбалансированных скобок сбалансирована. Это утверждение очевидно, поскольку можно сопоставить скобки в двух цепочках независимо друг от друга. Вторая продукция, $B \to (B)$, говорит, что если поместить пару скобок вокруг сбалансированной цепочки, то новая цепочка также будет сбалансированной. Это утверждение тоже очевидно, так как если скобки внутренней цепочки соответствуют друг другу, их можно удалить, и новые скобки становятся соседними. Третья продукция, $B \to \varepsilon$, является базисной, гласящей, что пустая цепочка сбалансирована.

Приведенные выше неформальные доводы должны убедить нас, что G_{bal} порождает только цепочки сбалансированных скобок. Нам еще нужно доказать обратное: что каждая цепочка сбалансированных скобок порождается этой грамматикой. Однако доказательство индукцией по длине сбалансированной цепочки весьма просто и оставляется в качестве упражнения.

Мы отмечали, что множество цепочек сбалансированных скобок не является регулярным языком, и теперь докажем это. Если бы $L(G_{bal})$ был регулярным, то для него по лемме о накачке для регулярных языков существовала бы константа n. Рассмотрим сбалансированную цепочку $w = (^n)^n$, т.е. n левых скобок, за которыми следуют n правых. Если разбить $w = xyz$ в соответствии с леммой, то y состоит только из левых скобок, и цепочка xz содержит больше правых скобок, чем левых. Эта цепочка несбалансированна, т.е. получено противоречие с предположением, что язык сбалансированных скобок регулярен. □

Языки программирования содержат, конечно же, не только скобки, но скобки составляют существенную часть арифметических и условных выражений. Грамматика, изображенная на рис. 5.2, более типична для структуры арифметических выражений, хотя там использованы всего два оператора, сложения и умножения, и включена детальная структура идентификаторов, которая, вероятней всего, обрабатывалась бы лексическим анали-

затором компилятора, как мы упоминали в разделе 3.3.2. Однако язык, представленный на рис. 5.2, также нерегулярен. Например, в соответствии с этой грамматикой, $(^n a)^n$ является правильным выражением. Мы можем использовать лемму о накачке для того, чтобы показать, что если бы язык был регулярным, то цепочка с некоторыми удаленными левыми скобками, символом a и всеми нетронутыми правыми скобками также была бы правильным выражением, что неверно.

Многие объекты типичного языка программирования ведут себя подобно сбалансированным скобкам. Обычно это сами скобки в выражениях всех типов, а также начала и окончания блоков кода, например, слова **`begin`** и **`end`** в языке Pascal, или фигурные скобки { и } в C. Таким образом, любое появление фигурных скобок в C-программе должно образовывать сбалансированную последовательность с { в качестве левой скобки и } — правой.

Есть еще один способ балансирования "скобок", отличающийся тем, что левые скобки могут быть несбалансированны, т.е. не иметь соответствующих правых. Примером является обработка **`if`** и **`else`** в C. Произвольная **`if`**-часть может быть как сбалансирована, так и не сбалансирована некоторой **`else`**-частью. Грамматика, порождающая возможные последовательности слов **`if`** и **`else`**, представленных i и e соответственно, имеет следующие продукции.

$$S \to \varepsilon \mid SS \mid iS \mid iSeS$$

Например, *ieie*, *iie* и *iei* являются возможными последовательностями слов **`if`** и **`else`**, и каждая из этих цепочек порождается данной грамматикой. Примерами неправильных последовательностей, не порождаемых грамматикой, являются *ei* и *ieeii*.

Простая проверка (доказательство ее корректности оставляется в качестве упражнения) того, что последовательность символов i и e порождается грамматикой, состоит в рассмотрении каждого e по очереди слева направо. Найдем первое i слева от рассматриваемого e. Если его нет, цепочка не проходит проверку и не принадлежит языку. Если такое i есть, вычеркнем его и рассматриваемое e. Затем, если больше нет символов e, цепочка проходит проверку и принадлежит языку. Если символы e еще есть, то проверка продолжается.

Пример 5.20. Рассмотрим цепочку *iee*. Первое e соответствует i слева от него. Оба удаляются. Оставшееся e не имеет i слева, и проверка не пройдена; слово *iee* не принадлежит языку. Отметим, что это заключение правильно, поскольку в C-программе слов **`else`** не может быть больше, чем **`if`**.

В качестве еще одного примера рассмотрим *iieie*. Соответствие первого e и i слева от него оставляет цепочку *iie*. Соответствие оставшегося e и i слева оставляет i. Символов e больше нет, и проверка пройдена. Это заключение также очевидно, поскольку последовательность *iieie* соответствует C-программе, структура которой подобна приведенной на рис. 5.10. В действительности, алгоритм проверки соответствия (и компилятор C) говорит нам также, какое именно **`if`** совпадает с каждым данным **`else`**. Это знание существенно, если компилятор должен создавать логику потока управления, подразумеваемую программистом. □

```
if (Условие) {
   ...
   if (Условие) Инструкция;
   else Инструкция;
   ...
   if (Условие) Инструкция;
   else Инструкция;
   ...
}
```

Рис. 5.10. Структура if-else; два слова ***else*** *соответствуют предыдущим* ***if****, а первое* ***if*** *несбалансированно*

5.3.2. Генератор синтаксических анализаторов YACC

Генерация синтаксического анализатора (функция, создающая деревья разбора по исходным программам) воплощена в программе YACC, реализованной во всех системах UNIX. На вход YACC подается КС-грамматика, запись которой отличается от используемой здесь только некоторыми деталями. С каждой продукцией связывается *действие* (action), представляющее собой фрагмент С-кода, который выполняется всякий раз, когда создается узел дерева разбора, соответствующий (вместе со своими сыновьями) этой продукции. Обычно действием является код для построения этого узла, хотя в некоторых приложениях YACC дерево разбора не создается, и действие задает что-то другое, например, выдачу порции объектного кода.

Пример 5.21. На рис. 5.11 показан пример КС-грамматики в нотации YACC. Грамматика совпадает с приведенной на рис. 5.2. Мы опустили действия, показав лишь их (требуемые нотацией) фигурные скобки и расположение во входной последовательности YACC.

```
Exp     : Id              {...}
        | Exp '+' Exp     {...}
        | Exp '*' Exp     {...}
        | '(' Exp ')'     {...}
        ;
Id      : 'a'             {...}
        'b'               {...}
        | Id 'a'          {...}
        | Id 'b'          {...}
        | Id '0'          {...}
        | Id '1'          {...}
        ;
```

Рис. 5.11. Пример грамматики в нотации YACC

Отметим следующие соответствия между нотацией YACC и нашими грамматиками.

- Двоеточие используется в качестве символа продукции $\rightarrow$.
- Все продукции с данной головой группируются вместе, и их тела разделены вертикальной чертой.
- Список тел для данной головы заканчивается точкой с запятой. Завершающий символ не используется.
- Терминалы записываются в апострофах. Некоторые буквы могут появляться в одиночных апострофах. Хотя у нас они не показаны, YACC позволяет пользователю определять также и символические терминалы. Появление таких терминалов в исходной программе обнаруживает лексический анализатор и сигнализирует об этом синтаксическому анализатору через свое возвращаемое значение.
- Цепочки символов и цифр, не взятые в апострофы, являются именами переменных. Мы воспользовались этой возможностью для того, чтобы дать нашим переменным более выразительные имена — `Exp` и `Id`, хотя можно было использовать *E* и *I*.

□

5.3.3. Языки описания документов

Рассмотрим семейство "языков", которые называются языками *описания документов* (markup languages). "Цепочками" этих языков являются документы с определенными метками, которые называются *дескрипторами* (tags). Дескрипторы говорят о семантике различных цепочек внутри документа.

Читатель, возможно, знаком с таким языком описания документов, как HTML (HyperText Markup Language). Этот язык имеет две основные функции: создание связей между документами и описание формата ("вида") документа. Мы дадим лишь упрощенный взгляд на структуру HTML, но следующие примеры должны показать его структуру и способ использования КС-грамматики как для описания правильных HTML-документов, так и для управления обработкой документа, т.е. его отображением на мониторе или принтере.

Пример 5.22. На рис. 5.12, *а* показан текст, содержащий список пунктов, а на рис. 5.12, *б* — его выражение в HTML. На рис. 5.12, *б*, показано что HTML состоит из обычного текста, перемежаемого дескрипторами. Соответствующие друг другу, т.е. парные дескрипторы имеют вид <*x*> и </*x*> для некоторой цепочки *x*.[3] Например, мы видим парные дескрипторы `<EM>` и `</EM>`, которые сигнализируют, что текст между ними должен быть выделен, т.е. напечатан курсивом или другим подходящим шрифтом. Мы

[3] Иногда введение признака <*x*> несет в себе больше информации, чем просто имя *x* для признака. Однако мы не рассматриваем в примерах эту возможность.

видим также парные дескрипторы <OL> и </OL>, указывающие на упорядоченный список, т.е. на нумерацию элементов списка.
Вещи, которые я *ненавижу*.

1. Заплесневелый хлеб.
2. Людей, которые ведут машину по узкой дороге слишком медленно.

а) видимый текст

```
<P>Вещи, которые я <EM>ненавижу</EM>:
<OL>
<LI>Заплесневелый хлеб.
<LI>Людей, которые ведут машину по узкой дороге слишком медленно.
</OL>
```

б) исходный HTML-текст

Рис. 5.12. HTML-документ и его видимая версия

Мы видим также два примера непарных дескрипторов: <P> и <LI>, которые вводят абзацы и элементы списка, соответственно. HTML допускает, а в действительности поощряет, чтобы эти дескрипторы сопровождались парными им </P> и </LI> на концах абзацев и списков, однако не требует этого. Поэтому эти парные дескрипторы не рассматриваются, что обеспечивает некоторую сложность нашей HTML-грамматике, развиваемой далее. □

Существует много классов цепочек, связанных с HTML-документом. Мы не будем стремиться перечислить их все, а представим только существенные для понимания текстов, подобных приведенному в примере 5.22. Для каждого класса мы введем переменную с содержательным именем.

1. *Text* (текст) — это произвольная цепочка символов, которая может быть проинтерпретирована буквально, т.е. не имеющая дескрипторов. Примером *элемента-текста* служит “Заплесневелый хлеб” (см. рис. 5.12).
2. *Char* (символ) — цепочка, состоящая из одиночного символа, допустимого в HTML-тексте. Заметим, что пробелы рассматриваются как символы.
3. *Doc* (документ) представляет документы, которые являются последовательностями “элементов”. Мы определим элементы следующими, и это определение будет взаимно рекурсивным с определением класса *Doc*.
4. *Element* (элемент) — это или цепочка типа *Text*, или пара соответствующих друг другу дескрипторов и документ между ними, или непарный дескриптор, за которым следует документ.
5. *ListItem* (элемент списка) есть дескриптор <LI> со следующим за ним документом, который представляет собой одиночный элемент списка.

6. *List* (список) есть последовательность из нуля или нескольких элементов списка.

1.	*Char*	→	*a* | *A* | …
2.	*Text*	→	ε | *Char Text*
3.	*Doc*	→	ε | *Element Doc*
4.	*Element*	→	*Text* | <EM> *Doc* </EM> | <P> *Doc* | <OL> *List* </OL> | …
5.	*ListItem*	→	<LI> *Doc*
6.	*List*	→	ε | *ListItem List*

Рис. 5.13. Часть грамматики HTML

На рис. 5.13 представлена КС-грамматика, которая описывает часть структуры языка HTML, рассмотренную нами в примерах. В строке 1 подразумевается, что символами могут быть “a”, “A” или многие другие символы из набора HTML. Строка 2 с использованием двух продукций гласит, что *Text* может быть либо пустой цепочкой, либо любым допустимым символом с текстом, следующим за ним. Иными словами, *Text* есть последовательность символов, возможно, пустая. Заметим, что символы < и > не являются допустимыми, хотя их можно представить последовательностями `<` и `>` соответственно. Таким образом, мы не сможем случайно вставить дескриптор в *Text*.

Строка 3 гласит, что документ является последовательностью из нуля или нескольких “элементов”. Элемент, в свою очередь, согласно строке 4 есть либо текст, либо выделенный документ, либо начало абзаца с документом, либо список. Мы также предполагаем, что существуют и другие продукции для элементов, соответствующие другим видам дескрипторов HTML. Далее, в строке 5 мы находим, что элемент списка представляет собой дескриптор `<LI>`, за которым следует произвольный документ, а строка 6 гласит, что список есть последовательность из нуля или нескольких элементов списка.

Для некоторых объектов HTML мощность КС-грамматик не нужна; достаточно регулярных выражений. Например, строки 1 и 2 (см. рис. 5.13) просто говорят, что *Text* представляет тот же язык, что и регулярное выражение $(\mathbf{a} + \mathbf{A} + \ldots)^*$. Однако для некоторых объектов мощность КС-грамматик необходима. Например, каждая пара соответствующих друг другу дескрипторов, вроде `<EM>` и `</EM>`, подобна сбалансированным скобкам, которые, как мы уже знаем, нерегулярны.

5.3.4. XML и определения типа документа

Тот факт, что HTML описывается грамматикой, сам по себе не является значительным. Практически все языки программирования можно описать соответствующими им КС-грамматиками, поэтому более удивительным было бы, если бы мы *не смогли* описать

HTML. Однако если мы обратимся к другому важному языку описания документов, XML (eXtensible Markup Language), то обнаружим, что КС-грамматики играют более существенную роль как часть процесса использования этого языка.

Цель XML состоит не в описании форматирования документа; это работа для HTML. Вместо этого XML стремится описать "семантику" текста. Например, текст наподобие "Кленовая ул., 12" выглядит как адрес, но является ли им? В XML дескрипторы окружали бы фразу, представляющую адрес, например:

```
<ADDR>Кленовая ул., 12</ADDR>
```

Однако сразу не очевидно, что `<ADDR>` означает адрес дома. Например, если бы документ говорил о распределении памяти, мы могли бы предполагать, что дескриптор `<ADDR>` ссылается на адрес в памяти. Ожидается, что стандарты описания различных типов дескрипторов и структур, которые могут находиться между парами таких дескрипторов, будут развиваться в различных сферах деятельности в виде определений типа документа (DTD — Document-Type Definition).

DTD, по существу, является КС-грамматикой с собственной нотацией для описания переменных и продукций. Приведем простое DTD и представим некоторые средства, используемые в языке описания DTD. Язык DTD сам по себе имеет КС-грамматику, но не она интересует нас. Мы хотим увидеть, как КС-грамматики выражаются в этом языке.

DTD имеет следующий вид.

```
<!DOCTYPE имя-DTD [
   список определений элементов
]>
```

Определение элемента, в свою очередь, имеет вид

```
<!ELEMENT имя-элемента (описание элемента)>
```

Описания элементов являются, по существу, регулярными выражениями. Их базис образуется следующими выражениями.

1. Имена других элементов, отражающие тот факт, что элементы одного типа могут появляться внутри элементов другого типа, как в HTML мы могли бы найти выделенный текст в списке.
2. Специальное выражение `\#PCDATA`, обозначающее любой текст, который не включает дескрипторы XML. Это выражение играет роль переменной *Text* в примере 5.22.

Допустимы следующие знаки операций.

1. |, обозначающий объединение, как в записи регулярных выражений, обсуждавшихся в разделе 3.3.1.
2. Запятая для обозначения конкатенации.

3. Три варианта знаков операции замыкания, как в разделе 3.3.1. Знак * означает “нуль или несколько появлений”, + — “не менее одного появления”, ? — “нуль или одно появление”.

Скобки могут группировать операторы и их аргументы; в их отсутствие действуют обычные приоритеты регулярных операций.

Пример 5.23. Представим себе, что продавцы компьютеров собрались, чтобы создать общедоступный стандарт DTD для описания разнообразных ПК (персональных компьютеров), которыми они торгуют. Каждое описание ПК будет иметь номер модели и спецификацию ее свойств, например, объем памяти, количество и размер дисков и т.д. На рис. 5.14 представлено гипотетическое, весьма упрощенное DTD для ПК.

```
<!DOCTYPE PcSpec [
   <!ELEMENT PCS (PC*)>
   <!ELEMENT PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)>
   <!ELEMENT MODEL (\#PCDATA)>
   <!ELEMENT PRICE (\#PCDATA)>
   <!ELEMENT PROCESSOR (MANF, MODEL, SPEED)>
   <!ELEMENT MANF (\#PCDATA)>
   <!ELEMENT MODEL (\#PCDATA)>
   <!ELEMENT SPEED (\#PCDATA)>
   <!ELEMENT RAM (\#PCDATA)>
   <!ELEMENT DISK (HARDDISK | CD | DVD)>
   <!ELEMENT HARDDISK (MANF, MODEL, SIZE)>
   <!ELEMENT SIZE (\#PCDATA)>
   <!ELEMENT CD (SPEED)>
   <!ELEMENT DVD (SPEED)>
]>
```

Рис. 5.14. DTD для персональных компьютеров

Именем DTD является `PcSpec. PCS` (список спецификаций) является первым элементом, аналогичным стартовому символу КС-грамматики. Его определение, `PC*`, гласит, что `PCS` — это нуль или несколько элементов `PC` (ПК).

Далее мы видим определение элемента `PC`. Оно состоит из конкатенации пяти компонентов. Первые четыре — это другие элементы, соответствующие модели (`MODEL`), цене (`PRICE`), типу процессора (`PROCESSOR`) и памяти (`RAM`). Каждый из них должен появляться один раз в указанном порядке, поскольку запятая обозначает конкатенацию. Последний компонент, `DISK+`, говорит, что у ПК будет один или несколько дисководов.

Многие компоненты представляют собой просто текст; к этому типу относятся `MODEL`, `PRICE` и `RAM`. Однако `PROCESSOR` имеет структуру. Из его определения видно, что он состоит из названия производителя (manufacturer, `MANF`), модели и скорости (`SPEED`), в указанном порядке. Каждый из этих элементов является простым текстом.

Элемент DISK наиболее сложен. Во-первых, диск — это либо жесткий диск (HARDDISK), либо CD, либо DVD, что указано в правиле для элемента DISK операциями "логического или". Жесткие диски, в свою очередь, имеют структуру, в которой определяются производитель (MANF), модель (MODEL) и размер (SIZE), тогда как CD и DVD представлены только их скоростью.

На рис. 5.15 показан пример XML-документа, соответствующего определению на рис. 5.14. Заметим, что каждый элемент представлен в документе дескриптором с именем элемента и парным дескриптором в конце с дополнительной чертой "/", как и в HTML. Таким образом, на внешнем уровне (см. рис. 5.15) виден дескриптор <PCS>...</PCS>. Между этими дескрипторами появляется список элементов, по одному на каждый продаваемый ПК; только один из этих списков показан полностью.

В пределах представленного входа <PC> мы видим, что модель имеет номер 4560, цена ее $2295, и она имеет процессор Intel Pentium 800MHz. Она также имеет 256Mb памяти, жесткий диск 30.5Gb Maxtor Diamond и читающее устройство 32x CD-ROM. Важно не то, что мы можем распознать все эти сведения, а то, чтобы читать и правильно интерпретировать числа и имена (см. рис. 5.15) этого документа могла программа под управлением DTD (см. рис. 5.14), которое должно быть ею прочитано вначале. □

```
<PCS>
   <PC>
      <MODEL>4560</MODEL>
      <PRICE>$2295</PRICE>
      <PROCESSOR>
         <MANF>Intel</MANF>
         <MODEL>Pentium</MODEL>
         <SPEED>800mhZ</SPEED>
      </PROCESSOR>
      <RAM>256</RAM>
      <DISK><HARDDISK>
         <MANF>Maxtor</MANF>
         <MODEL>Diamond</MODEL>
         <SIZE>30.5Gb</SIZE>
      </HARDDISK></DISK>
      <DISK><CD>
         <SPEED>32x</SPEED>
      </CD></DISK>
   </PC>
   <PC>
      ...
   </PC>
</PCS>
```

Рис. 5.15. Часть документа, удовлетворяющая структуре DTD (см. рис. 5.14)

Возможно, читатель заметил, что правила для элементов в DTD (см. рис. 5.14) не полностью совпадают с продукциями в КС-грамматиках. Многие правила имеют корректный вид, например, правило

```
<!ELEMENT PROCESSOR (MANF, MODEL, SPEED)>
```

аналогично продукции

Processor → *Manf Model Speed*.

Однако в правиле

```
<!ELEMENT DISK (HARDDISK | CD | DVD)>
```

определение для `DISK` не похоже на тело продукции. Расширение в этом случае является простым: это правило можно интерпретировать как три продукции, у которых вертикальная черта играет ту же роль, что и в продукциях обычного вида. Таким образом, это правило эквивалентно следующим трем продукциям.

Disk → *Harddisk* | *Cd* | *Dvd*

Труднее всего следующее правило.

```
<!ELEMENT PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)>
```

Здесь "тело" содержит оператор замыкания. Решение состоит в замене `DISK+` новой переменной, скажем, *Disks*, которая порождает с помощью пары продукций один или несколько экземпляров переменной *Disk*. Итак, мы получаем следующие эквивалентные продукции.

Pc → *Model Price Processor Ram Disks*
Disks → *Disk* | *Disk Disks*

Существует общая техника преобразования КС-грамматики с регулярными выражениями в качестве тела продукций в обычные КС-грамматики. Идею этого преобразования опишем неформально; возможно, читатель захочет уточнить как смысл КС-грамматик с регулярными выражениями, так и доказательство того, что такое расширение не приводит к порождению языков, не являющихся КС-языками. Мы покажем, как продукция с регулярным выражением в качестве тела преобразуется в совокупность обычных продукций. Для этого применим индукцию по размеру выражения в теле.

Базис. Если тело представляет собой конкатенацию элементов, то продукция уже имеет допустимый в КС-грамматиках вид, поэтому преобразовывать нечего.

Индукция. В зависимости от старшего оператора возможны пять ситуаций.

1. При конкатенации продукция имеет вид $A \to E_1, E_2$, где E_1 и E_2 — выражения, допустимые в языке DTD. Введем две переменные, B и C, не используемые больше нигде. Заменим $A \to E_1, E_2$:

   $A \to BC$
   $B \to E_1$
   $C \to E_2$

Первая продукция, $A \to BC$, допустима в КС-грамматиках. Две последние могут быть как допустимыми, так и недопустимыми. Однако их тела короче, чем тело исходной продукции, поэтому на основании индукции их можно преобразовать в форму КС-грамматик.

2. Для оператора объединения продукция имеет вид $A \to E_1 \mid E_2$. Заменим ее следующей парой продукций.

   $A \to E_1$
   $A \to E_2$

   Аналогично, эти продукции могут иметь недопустимый вид, но их тела короче, чем тело исходной. Применяем эти же правила рекурсивно и преобразуем их к виду КС-грамматик.

3. Продукция имеет вид $A \to (E_1)^*$. Введем новую переменную B, не используемую больше нигде, и заменим продукцию следующими тремя.

   $A \to BA$
   $A \to \varepsilon$
   $B \to E_1$

4. Для продукции $A \to (E_1)^+$ вводим новую переменную B, не используемую больше нигде, и заменяем продукцию следующими тремя.

   $A \to BA$
   $A \to B$
   $B \to E_1$

5. Продукция имеет вид $A \to (E_1)?$. Заменим ее:

   $A \to \varepsilon$
   $A \to E_1$

Пример 5.24. Рассмотрим преобразование DTD-правила

```
<!ELEMENT PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)>
```

в обычные для КС-грамматик продукции. Тело этого правила рассмотрим как конкатенацию двух выражений, первое из которых есть `MODEL, PRICE, PROCESSOR, RAM`, а второе — `DISK+`. Создав для этих подвыражений переменные A и B, соответственно, используем следующие продукции.

$Pc \to AB$
$A \to Model\ Price\ Processor\ Ram$
$B \to Disk+$

Только последняя не имеет нужного вида. Введем еще одну переменную C и следующие продукции.

$B \to CB \mid C$
$C \to Disk$

В данном частном случае переменные A и C в действительности не нужны, поскольку выражение, порождаемое из A, есть просто конкатенация переменных, а *Disk* — одиночная переменная. Вместо приведенных продукций можно было бы использовать следующие.

$Pc \rightarrow Model\ Price\ Processor\ Ram\ B$

$B \rightarrow Disk\ B \mid Disk$

□

5.3.5. Упражнения к разделу 5.3

5.3.1. Докажите, что если цепочка скобок сбалансирована (как в примере 5.19), то она порождается грамматикой $B \rightarrow BB \mid (B) \mid \varepsilon$. *Указание*. Проведите индукцию по длине цепочки.

5.3.2. Рассмотрим множество всех цепочек сбалансированных скобок двух типов, круглых и квадратных. Следующий пример показывает их происхождение. Если взять выражения в языке C, которые используют круглые скобки для группирования и для вызовов функций и квадратные скобки для индексов массивов, и удалить из них все, кроме скобок, то получим цепочки сбалансированных скобок этих двух типов. Например,

```
f(a[i]*(b[i][j]+c[g(x)]),d[i])
```

превращается в сбалансированную цепочку `([]([][][()])[])`. Построить грамматику для всех сбалансированных цепочек из круглых и квадратных скобок, и только для них.

5.3.3. В разделе 5.3 рассматривалась грамматика $S \rightarrow \varepsilon \mid SS \mid iS \mid iSeS$ и утверждалось, что принадлежность цепочки w языку L этой грамматики можно проверить путем повторения следующих действий, начиная с w.

1. Если текущая цепочка начинается с e, то проверка не пройдена; $w \notin L$.
2. Если текущая цепочка не содержит e, то проверка пройдена; $w \in L$.
3. В противном случае удалить первое e и i непосредственно слева от него. Повторить эти три шага с полученной цепочкой.

Докажите, что этот процесс правильно распознает цепочки языка L.

5.3.4. Добавьте к грамматике HTML (см. рис. 5.13) следующие формы:

а) (*****) элемент списка должен заканчиваться закрывающим дескриптором `</LI>`;

б) элемент может быть как неупорядоченным, так и упорядоченным списком. Неупорядоченные списки заключаются в парные дескрипторы `<UL>` и `</UL>`;

в) (**!**) элемент может быть таблицей, которая заключается в парные дескрипторы `<TABLE>` и `</TABLE>`. Между ними находятся одна или несколько цепочек, каждая из которых заключается в `<TR>` и `</TR>`. Первая цепочка яв-

ляется заголовком с одним или несколькими полями, каждое из которых начинается дескриптором <TH> (будем предполагать, что эти дескрипторы не закрываются, хотя в действительности они парные). Поля в следующих цепочках начинаются дескриптором <TD>.

```
<!DOCTYPE CourseSpec[
   <!ELEMENT COURSES (COURSE+)>
   <!ELEMENT COURSE (CNAME, PROF, STUDENT*, TA?)>
   <!ELEMENT CNAME (\#PCDATA)>
   <!ELEMENT STUDENT (\#PCDATA)>
   <!ELEMENT TA (\#PCDATA)>
]>
```

Рис. 5.16. DTD для курсов

5.3.5. Преобразуйте DTD (рис. 5.16) в КС-грамматику.

5.4. Неоднозначность в грамматиках и языках

Как мы увидели, в приложениях КС-грамматики часто служат основой для обеспечения структуры различного рода файлов. Например, в разделе 5.3 грамматики использовались для придания структуры программам и документам. Там действовало неявное предположение, что грамматика однозначно определяет структуру каждой цепочки своего языка. Однако мы увидим, что не каждая грамматика обеспечивает уникальность структуры.

Иногда, когда грамматика не может обеспечить уникальность структуры, ее можно преобразовать, чтобы структура была единственной для каждой цепочки. К сожалению, это возможно не всегда, т.е. существуют “существенно неоднозначные” языки; каждая грамматика для такого языка налагает несколько структур на некоторые его цепочки.

5.4.1. Неоднозначные грамматики

Вернемся к грамматике выражений (см. рис. 5.2). Эта грамматика дает возможность порождать выражения с любой последовательностью операторов + и *, а продукции $E \to E + E \mid E * E$ позволяют порождать эти выражения в произвольно выбранном порядке.

Пример 5.25. Рассмотрим выводимую цепочку $E + E * E$. Она имеет два порождения из E:

1. $E \Rightarrow E + E \Rightarrow E + E * E$
2. $E \Rightarrow E * E \Rightarrow E + E * E$

Заметим, что в порождении 1 второе E заменяется на $E * E$, тогда как в порождении 2 — первое E на $E + E$. На рис. 5.17 показаны два действительно различных дерева разбора.

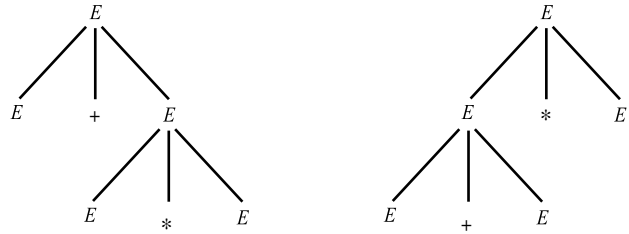


Рис. 5.17. Два дерева разбора с одной и той же кроной

Разница между этими двумя порождениями значительна. Когда рассматривается структура выражений, порождение 1 говорит, что второе и третье выражения перемножаются, и результат складывается с первым. Вместе с тем, порождение 2 задает сложение первых двух выражений и умножение результата на третье. Более конкретно, первое порождение задает, что 1 + 2 * 3 группируется как 1 + (2 * 3) = 7, а второе — что группирование имеет вид (1 + 2) * 3 = 9. Очевидно, что первое из них (но не второе) соответствует нашему понятию о правильном группировании арифметических выражений.

Поскольку грамматика, представленная на рис. 5.2, дает две различные структуры любой цепочке терминалов, порождаемой заменой трех выражений в $E + E * E$ идентификаторами, для обеспечения уникальности структуры она не подходит. В частности, хотя она может давать цепочкам как арифметическим выражениям правильное группирование, она также дает им и неправильное. Для того чтобы использовать грамматику выражений в компиляторе, мы должны изменить ее, обеспечив только правильное группирование. □

С другой стороны, само по себе существование различных порождений цепочки (что не равносильно различным деревьям разбора) еще не означает порочности грамматики. Рассмотрим пример.

Пример 5.26. Используя ту же грамматику выражений, мы находим, что цепочка $a + b$ имеет много разных порождений. Вот два из них.

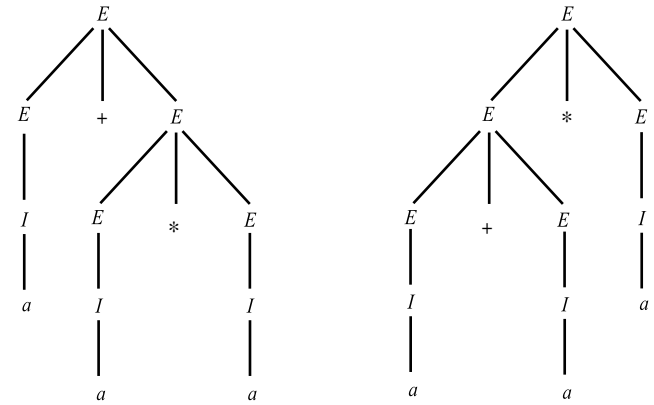
1. $E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b$
2. $E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b$

Заметим, что настоящей разницы между структурами, заданными этими двумя порождениями, нет. Каждая из них говорит, что a и b — идентификаторы, и что их значения нужно сложить. В действительности, оба эти порождения приводят к одному и тому же дереву разбора, если применяются конструкции теорем 5.18 и 5.12. □

Два примера, приведенные выше, показывают, что неоднозначность происходит не от множественности порождений, а от существования двух и более деревьев разбора. Итак, мы говорим, что КС-грамматика $G = (V, T, P, S)$ является *неоднозначной*, если найдется хотя бы одна цепочка w в T^*, для которой существуют два разных дерева разбора, каждое

с корнем, отмеченным S, и кроной w. Если же каждая цепочка имеет не более одного дерева разбора в грамматике, то грамматика *однозначна*.

Пример 5.25 почти показал неоднозначность грамматики, изображенной на рис. 5.2. Нам нужно лишь доказать, что деревья разбора на рис. 5.17 можно пополнить так, чтобы они имели терминальные кроны. На рис. 5.18 приведен пример такого пополнения.



а) *б*)

Рис. 5.18. Деревья с кроной a + a * a, *показывающие неоднозначность грамматики выражений*

5.4.2. Исключение неоднозначности из грамматик

В идеальном мире мы смогли бы дать алгоритм исключения неоднозначности из КС-грамматик, почти как в разделе 4.4, где был приведен алгоритм удаления несущественных состояний конечного автомата. Однако, как будет показано в разделе 9.5.2, не существует даже алгоритма, способного различить, является ли КС-грамматика неоднозначной. Более того, в разделе 5.4.4 мы увидим, что существуют КС-языки, имеющие только неоднозначные КС-грамматики; исключение неоднозначности для них вообще невозможно.

К счастью, положение на практике не настолько мрачное. Для многих конструкций, возникающих в обычных языках программирования, существует техника устранения неоднозначности. Проблема с грамматикой выражений типична, и мы исследуем устранение ее неоднозначности в качестве важной иллюстрации.

Сначала заметим, что есть следующие две причины неоднозначности в грамматике, изображенной на рис. 5.2.

1. Не учитываются приоритеты операторов. В то время как на рис. 5.17, *а* оператор * правильно группируется перед оператором +, на рис. 5.17, *б* показано также допустимое дерево разбора, группирующее + перед *. Необходимо обеспечить, чтобы в однозначной грамматике была допустимой только структура, показанная на рис. 5.17, *а*.

2. Последовательность одинаковых операторов может группироваться как слева, так и справа. Например, если бы операторы * (см. рис. 5.17) были заменены операторами +, то мы увидели бы два разных дерева разбора для цепочки $E + E + E$. Поскольку оба оператора ассоциативны, не имеет значения, группируем ли мы слева или справа, но для исключения неоднозначности нам нужно выбрать что-то одно. Обычный подход состоит в группировании слева, поэтому только структура, изображенная на рис. 5.17, *б*, представляет правильное группирование двух операторов +.

Разрешение неоднозначности в YACC

Если используемая грамматика выражений неоднозначна, нас может удивить реалистичность YACC-программы, приведенной на рис. 5.11. Действительно, данная грамматика неоднозначна, однако генератор синтаксических анализаторов YACC обеспечивает пользователя простыми механизмами разрешения большинства общих причин неоднозначности. Для грамматики выражений достаточно потребовать следующее.

1. Приоритет у оператора * выше, чем у +, т.е. операторы * должны группироваться раньше, чем соседние с обеих сторон операторы +. Это правило говорит нам использовать порождение 1 из примера 5.25, а не порождение 2.
2. И *, и + левоассоциативны, т.е. последовательности выражений, связанных только знаком *, группируются слева, и это же относится к последовательностям, связанным +.

YACC позволяет нам устанавливать приоритеты операторов путем перечисления их в порядке возрастания приоритета. Технически приоритет оператора применяется к использованию любой продукции, в теле которой этот оператор является крайним справа терминалом. Мы можем также объявить операторы как лево- или правоассоциативные с помощью ключевых слов `%left` и `%right`. Например, для того, чтобы объявить оба оператора * и + левоассоциативными и с более высоким приоритетом у *, в начале грамматики (см. рис. 5.11) можно поместить следующие инструкции.

```
%left '+'
%left '*'
```

Решение проблемы установления приоритетов состоит в том, что вводится несколько разных переменных, каждая из которых представляет выражения, имеющие один и тот же уровень "связывающей мощности". В частности, для грамматики выражений это решение имеет следующий вид.

1. *Сомножитель*, или *фактор* (factor), — это выражение, которое не может быть разделено на части никаким примыкающим оператором, ни *, ни +. Сомножителями в нашем языке выражений являются только следующие выражения:

а) идентификаторы. Буквы идентификатора невозможно разделить путем присоединения оператора;

б) выражения в скобках, независимо от того, что находится между ними. Именно для предохранения операндов в скобках от действия внешних операторов и предназначены скобки.

2. *Терм* (term), или *слагаемое*, — это выражение, которое не может быть разорвано оператором +. В нашем примере, где операторами являются только + и *, терм представляет собой произведение одного или несколько сомножителей. Например, терм $a * b$ может быть "разорван", если мы используем левую ассоциативность * и поместим $a1$ * слева, поскольку $a1 * a * b$ группируется слева как $(a1 * a) * b$, разрывая $a * b$. Однако помещение аддитивного выражения слева, типа $a1+$, или справа, типа $+a1$, не может разорвать $a * b$. Правильным группированием выражения $a1 + a * b$ является $a1 + (a * b)$, а выражения $a * b + a1$ — $(a * b) + a1$.

3. *Выражение* (expression) будет обозначать любое возможное выражение, включая те, которые могут быть разорваны примыкающими + и *. Таким образом, выражение для нашего примера представляет собой сумму одного или нескольких термов.

$$
\begin{aligned}
I &\rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\
F &\rightarrow I \mid (E) \\
T &\rightarrow F \mid T * F \\
E &\rightarrow T \mid E + T
\end{aligned}
$$

Рис. 5.19. Однозначная грамматика выражений

Пример 5.27. На рис. 5.19 приведена однозначная грамматика, порождающая тот же язык, что и грамматика, изображенная на рис. 5.2. Посмотрим на F, T и E как на переменные, языками которых являются сомножители, слагаемые и выражения в описанном выше смысле. Например, эта грамматика допускает только одно дерево разбора для цепочки $a + a * a$; оно показано на рис. 5.20.

То, что данная грамматика однозначна, может быть далеко не очевидно. Приведем основные утверждения, поясняющие, почему ни одна цепочка языка не имеет двух разных деревьев разбора.

- Цепочка, порождаемая из T, т.е. терм, должна быть последовательностью из одного или нескольких сомножителей, связанных знаками *. Сомножитель, по определению и как это следует из продукций для F (см. рис. 5.19), есть либо одиночный идентификатор, либо выражение в скобках.
- Вследствие вида продукций для T единственным деревом разбора для последовательности сомножителей будет такое, которое разрывает $f_1 * f_2 * \ldots * f_n$, где $n > 1$, на терм $f_1 * f_2 * \ldots * f_{n-1}$ и сомножитель f_n. Причина в том, что F не может поро-

дить выражение вида $f_{n-1} * f_n$ без введения скобок вокруг него. Таким образом, при использовании продукции $T \to T * F$ из F невозможно породить ничего, кроме последнего из сомножителей, т.е. дерево разбора для терма может выглядеть только так, как на рис. 5.21.

- Аналогично, выражение есть последовательность термов, связанных знаками +. Когда используется продукция $E \to E + T$ для порождения $t_1 + t_2 + \ldots + t_n$, из T должно порождаться только t_n, а из E в теле — $t_1 + t_2 + \ldots + t_{n-1}$. Причина этого опять-таки в том, что из T невозможно породить сумму двух и более термов без заключения их в скобки.

□

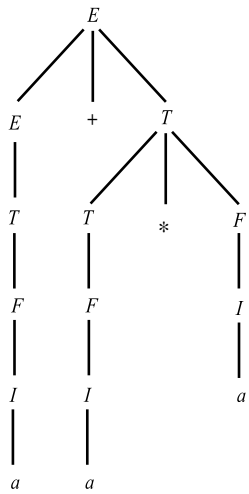


Рис. 5.20. Единственное дерево разбора для цепочки a + a * a

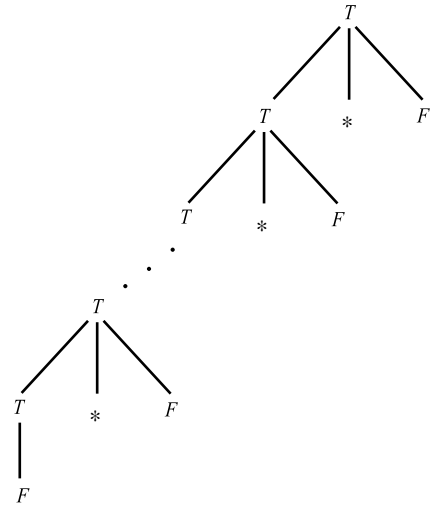


Рис. 5.21. Форма всех деревьев разбора для термов

5.4.3. Левые порождения как способ выражения неоднозначности

Хотя порождения не обязательно уникальны, даже если грамматика однозначна, оказывается, что в однозначной грамматике и левые, и правые порождения уникальны. Рассмотрим только левые.

Пример 5.28. Заметим, что оба дерева разбора, представленные на рис. 5.18, имеют крону $E + E * E$. По ним получаются следующие соответствующие им левые порождения.

а) $E \underset{lm}{\Rightarrow} E + E \underset{lm}{\Rightarrow} I + E \underset{lm}{\Rightarrow} a + E \underset{lm}{\Rightarrow} a + E * E \underset{lm}{\Rightarrow} a + I * E \underset{lm}{\Rightarrow}$

$a + a * E \underset{lm}{\Rightarrow} a + a * I \underset{lm}{\Rightarrow} a + a * a$

б) $E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} E + E * E \underset{lm}{\Rightarrow} I + E * E \underset{lm}{\Rightarrow} a + E * E \underset{lm}{\Rightarrow}$

$a + I * E \underset{lm}{\Rightarrow} a + a * E \underset{lm}{\Rightarrow} a + a * I \underset{lm}{\Rightarrow} a + a * a$

Заметим, что эти левые порождения различаются. Данный пример не доказывает теорему, но демонстрирует, как различия в деревьях разбора влияют на выбор шагов в левом порождении. □

Теорема 5.29. Для каждой грамматики $G = (V, T, P, S)$ и w из T^* цепочка w имеет два разных дерева разбора тогда и только тогда, когда w имеет два разных левых порождения из S.

Доказательство. (*Необходимость*) Внимательно рассмотрим построение левого порождения по дереву разбора в доказательстве теоремы 5.14. В любом случае, если у двух деревьев разбора впервые появляется узел, в котором применяются различные продукции, левые порождения, которые строятся, также используют разные продукции и, следовательно, являются различными.

(*Достаточность*) Хотя мы предварительно не описали непосредственное построение дерева разбора по левому порождению, идея его проста. Начнем построение дерева с корня, отмеченного стартовым символом. Рассмотрим порождение пошагово. На каждом шаге заменяется переменная, и эта переменная будет соответствовать построенному крайнему слева узлу дерева, не имеющему сыновей, но отмеченному этой переменной. По продукции, использованной на этом шаге левого порождения, определим, какие сыновья должны быть у этого узла. Если существуют два разных порождения, то на первом шаге, где они различаются, построенные узлы получат разные списки сыновей, что гарантирует различие деревьев разбора. □

5.4.4. Существенная неоднозначность

Контекстно-свободный язык L называется *существенно неоднозначным*, если все его грамматики неоднозначны. Если хотя бы одна грамматика языка L однозначна, то L является однозначным языком. Мы видели, например, что язык выражений, порождаемый грамматикой, приведенной на рис. 5.2, в действительности однозначен. Хотя данная грамматика и неоднозначна, этот же язык задается еще одной грамматикой, которая однозначна — она изображена на рис. 5.19.

Мы не будем доказывать, что существуют неоднозначные языки. Вместо этого рассмотрим пример языка, неоднозначность которого можно доказать, и объясним неформально, почему любая грамматика для этого языка должна быть неоднозначной:

$$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \bigcup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}.$$

Из определения видно, что L состоит из цепочек вида $\mathbf{a}^+\mathbf{b}^+\mathbf{c}^+\mathbf{d}^+$, в которых поровну символов a и b, а также c и d, либо поровну символов a и d, а также b и c.

L является КС-языком. Очевидная грамматика для него показана на рис. 5.22. Для порождения двух видов цепочек в ней используются два разных множества продукций.

Эта грамматика неоднозначна. Например, у цепочки *aabbccdd* есть два следующих левых порождения.

1. $S \underset{lm}{\Rightarrow} AB \underset{lm}{\Rightarrow} aAbB \underset{lm}{\Rightarrow} aabbB \underset{lm}{\Rightarrow} aabbccBd \underset{lm}{\Rightarrow} aabbccdd$
2. $S \underset{lm}{\Rightarrow} C \underset{lm}{\Rightarrow} aCd \underset{lm}{\Rightarrow} aaDdd \underset{lm}{\Rightarrow} aabDcdd \underset{lm}{\Rightarrow} aabbccdd$

Соответствующие деревья разбора показаны на рис. 5.23.

$$
\begin{aligned}
S &\rightarrow AB \mid C \\
A &\rightarrow aAb \mid ab \\
B &\rightarrow cBd \mid cd \\
C &\rightarrow aCd \mid aDd \\
D &\rightarrow bDc \mid bc
\end{aligned}
$$

Рис. 5.22. Грамматика для существенно неоднозначного языка

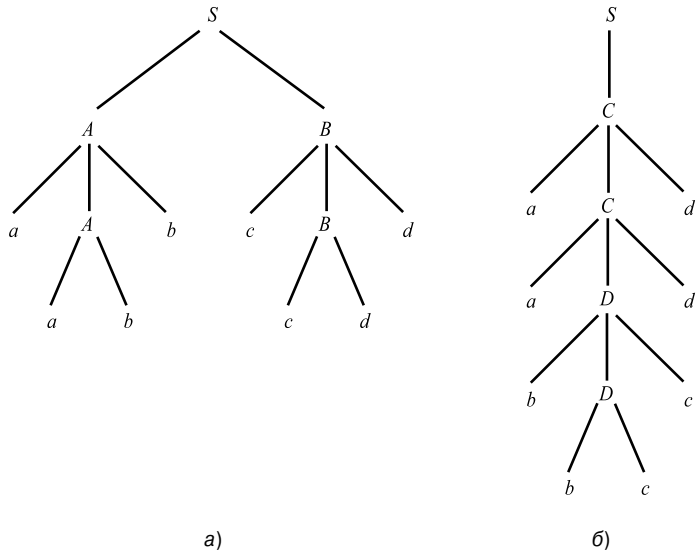


Рис. 5.23. Два дерева разбора для aabbccdd

Доказательство того, что все грамматики для языка L неоднозначны, весьма непросто, однако сущность его такова. Нужно обосновать, что все цепочки (за исключением конечного их числа), у которых поровну всех символов, должны порождаться двумя различными путями. Первый путь — порождение их как цепочек, у которых поровну символов *a* и *b*, а также *c* и *d*, второй путь — как цепочек, у которых поровну символов *a* и *d*, как и *b* и *c*.

Например, единственный способ породить цепочки, у которых поровну *a* и *b*, состоит в использовании переменной, подобной *A* в грамматике, изображенной на рис. 5.22. Ко-

нечно же, возможны варианты, но они не меняют картины в целом, как это видно из следующих примеров.

- Некоторые короткие цепочки могут не порождаться после изменения, например, базисной продукции $A \to ab$ на $A \to aaabbb$.
- Мы могли бы организовать продукции так, чтобы переменная A делила свою работу с другими, например, используя переменные A_1 и A_2, из которых A_1 порождает нечетные количества символов a, а A_2 — четные: $A_1 \to aA_2b \mid ab$; $A_2 \to aA_1b \mid ab$.
- Мы могли бы организовать продукции так, чтобы количества символов a и b, порождаемые переменной A, были не равны, но отличались на некоторое конечное число. Например, можно начать с продукции $S \to AaB$ и затем использовать $A \to aAb \mid a$ для порождения символов a на один больше, чем b.

В любом случае, нам не избежать некоторого способа порождения символов a, при котором соблюдается их соответствие с символами b.

Аналогично можно обосновать, что должна использоваться переменная, подобная B, которая порождает соответствующие друг другу символы c и d. Кроме того, в грамматике должны быть переменные, играющие роль переменных C и D, порождающих соответственно парные a и d и парные b и c. Приведенные аргументы, если их формализовать, доказывают, что независимо от изменений, которые можно внести в исходную грамматику, она будет порождать *хотя бы одну* цепочку вида $a^nb^nc^nd^n$ двумя способами, как и грамматика, изображенная на рис. 5.22.

5.4.5. Упражнения к разделу 5.4

5.4.1. Рассмотрим грамматику $S \to aS \mid aSbS \mid \varepsilon$. Она неоднозначна. Докажите, что для цепочки aab справедливо следующее:

а) для нее существует два дерева разбора;

б) она имеет два левых порождения;

в) она имеет два правых порождения.

5.4.2. (!) Докажите, что грамматика из упражнения 5.4.1 порождает те, и только те цепочки из символов a и b, у которых в любом префиксе символов a не меньше, чем b.

5.4.3. (*!) Найдите однозначную грамматику для языка из упражнения 5.4.1.

5.4.4. (!!) В грамматике из упражнения 5.4.1 некоторые цепочки имеют только одно дерево разбора. Постройте эффективную проверку, является ли цепочка одной из указанных. Проверка типа “проверить все деревья разбора, чтобы увидеть, сколько их у данной цепочки” не является достаточно эффективной.

5.4.5. (!) Воспроизведем грамматику из упражнения 5.1.2:

$$S \rightarrow A1B$$

$$A \rightarrow 0A \mid \varepsilon$$

$$B \rightarrow 0B \mid 1B \mid \varepsilon$$

а) докажите, что данная грамматика неоднозначна;

б) найдите однозначную грамматику для этого же языка и докажите ее однозначность.

5.4.6. (*!) Однозначна ли ваша грамматика из упражнения 5.1.5? Если нет, измените ее так, чтобы она стала однозначной.

5.4.7. Следующая грамматика порождает префиксные выражения с операндами *x* и *y* и операторами +, - и *:

$E \rightarrow + EE \mid * EE \mid - EE \mid x \mid y$

а) найдите левое и правое порождения, а также дерево разбора для цепочки `+*-xyxy`;

б) (!) докажите, что данная грамматика однозначна.

Резюме

- *Контекстно-свободные грамматики*. КС-грамматика — это способ описания языка с помощью рекурсивных правил, называемых продукциями. КС-грамматика состоит из множества переменных, терминальных символов, стартовой переменной, а также продукций. Каждая продукция состоит из головной переменной и тела — цепочки переменных и/или терминалов, возможно, пустой.
- *Порождения и языки*. Начиная со стартового символа, мы порождаем терминальные цепочки, повторяя замены переменных телами продукций с этими переменными в голове. Язык КС-грамматики — это множество терминальных цепочек, которые можно породить; он называется КС-языком.
- *Левые и правые порождения*. Если мы всегда заменяем крайнюю слева (крайнюю справа) переменную, то такое порождение является левым (правым). Каждая цепочка в языке КС-грамматики имеет, по крайней мере, одно левое и одно правое порождения.
- *Выводимые цепочки*. Любой шаг порождения дает цепочку переменных и/или терминалов. Она называется выводимой цепочкой. Если порождение является левым (правым), то цепочка называется левовыводимой (правовыводимой).
- *Деревья разбора*. Дерево разбора — это дерево, показывающее сущность порождения. Внутренние узлы отмечены переменными, листья — терминалами или ε.

Для каждого внутреннего узла должна существовать продукция, голова которой является отметкой узла, а отметки сыновей узла, прочитанные слева направо, образуют ее тело.

- *Эквивалентность деревьев разбора и порождений*. Терминальная цепочка принадлежит языку грамматики тогда и только тогда, когда она является кроной, по крайней мере, одного дерева разбора. Таким образом, существование левых порождений, правых порождений и деревьев разбора является равносильным условием того, что все они определяют в точности цепочки языка КС-грамматики.
- *Неоднозначные грамматики*. Для некоторых грамматик можно найти терминальную цепочку с несколькими деревьями разбора, или (что равносильно) с несколькими левыми или правыми порождениями. Такая грамматика называется неоднозначной.
- *Исключение неоднозначности*. Для многих полезных грамматик, в частности, описывающих структуру программ в обычных языках программирования, можно найти эквивалентные однозначные грамматики. К сожалению, однозначная грамматика часто оказывается сложнее, чем простейшая неоднозначная грамматика для данного языка. Существуют также некоторые КС-языки, обычно специально сконструированные, которые являются существенно неоднозначными, т.е. все грамматики для этих языков неоднозначны.
- *Синтаксические анализаторы*. Контекстно-свободная грамматика является основным понятием для реализации компиляторов и других процессоров языков программирования. Инструментальные средства, вроде YACC, получают на вход КС-грамматику и порождают синтаксический анализатор — часть компилятора, распознающую структуру компилируемых программ.
- *Определения типа документа*. Развивающийся стандарт XML для распределения информации посредством Web-документов имеет нотацию, называемую DTD, для описания структуры таких документов. Для этого в документ записываются вложенные семантические дескрипторы. DTD является, по существу, КС-грамматикой, язык которой — это класс связанных с этим определением документов.

Литература

Контекстно-свободная грамматика была впервые предложена Хомским как способ описания естественных языков в [4]. Бэкус и Наур вскоре использовали подобную идею для описания машинных языков Фортран [2] и Алгол [7]. В результате КС-грамматики иногда называются “формами Бэкуса-Наура”, или БНФ.

Неоднозначность в грамматиках была выделена в качестве проблемы почти одновременно Кантором [3] и Флойдом [5]. Существенная неоднозначность была впервые указана Гроссом [6].

О приложениях КС-грамматик в компиляции рекомендуем прочитать в [1]. DTD описаны в документе о стандартах XML [8].

1. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading MA, 1986. (Ахо А. В., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструменты. — М.: Издательский дом “Вильямс”, 2001.)
2. J. W. Backus, “The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference”, *Proc. Intl. Conf. on Information Processing* (1959), UNESCO, pp. 125–132.
3. D. C. Cantor, “On the ambiguity problem of Backus systems”, *J. ACM* **9**:4 (1962), pp. 477–479.
4. N. Chomsky, “Three models for the description of language”, *IRE Trans. on Information Theory* **2**:3 (1956), pp. 113–124. (Хомский Н. Три модели для описания языка. — Кибернетический сборник, вып. 2. — М.: ИЛ, 1961. — С. 237–266.)
5. R. W. Floyd, “On ambiguity in phrase-structure languages”, *Comm. ACM* **5**:10 (1962), pp. 526–534.
6. M. Gross, “Inherent ambiguity of minimal linear grammars”, *Information and Control* **7**:3 (1964), pp. 366–368.
7. P. Naur et al., “Report on the algorithmic language ALGOL 60”, *Comm. ACM* **3**:5 (1960), pp. 299-314. См. также *Comm. ACM* **6**:1 (1963), pp. 1–17. (Алгоритмический язык Алгол 60. — М.: Мир, 1965.)
8. World-Wide-Web Consortium, `http://www.w3.org/TR/REC-xml` (1998).

ГЛАВА 6

Автоматы с магазинной памятью

Контекстно-свободные языки задаются автоматами определенного типа. Такой автомат называется автоматом с магазинной памятью, или магазинным автоматом, и является расширением недетерминированного конечного автомата с ε-переходами, представляющего собой один из способов определения регулярных языков. Магазинный автомат — это, по существу, ε-НКА с добавлением магазина. Магазин может читаться, в его вершину могут добавляться (заталкиваться, помещаться, заноситься) или с его вершины могут сниматься символы точно так же, как в структуре данных типа "магазин".

В этой главе определяются две различные версии магазинных автоматов: одна из них допускает при достижении допускающего состояния, как это делают конечные автоматы, а другая — при опустошении магазина независимо от состояния. Показывается, что эти два варианта допускают контекстно-свободные языки, т.е. грамматики могут быть преобразованы в эквивалентные магазинные автоматы, и наоборот. Также вкратце рассматривается подкласс детерминированных магазинных автоматов. Множество допускаемых ими языков включает все регулярные языки, но является собственным подмножеством КС-языков. Поскольку механизмом своей работы они весьма похожи на синтаксические анализаторы, важно знать, какие языковые конструкции могут быть распознаны ими, а какие — нет.

6.1. Определение автоматов с магазинной памятью

В этом разделе магазинный автомат представлен сначала неформально, а затем — как формальная конструкция.

6.1.1. Неформальное введение

Магазинный автомат — это, по существу, недетерминированный конечный автомат с ε-переходами и одним дополнением — магазином, в котором хранится цепочка "магазинных символов". Присутствие магазина означает, что в отличие от конечного автомата магазинный автомат может "помнить" бесконечное количество информации. Однако в отличие от универсального компьютера, который также способен запоминать неограниченные объемы информации, магазинный автомат имеет доступ к информации в

магазине только с одного его конца в соответствии с принципом “последним пришел — первым ушел” (“last-in-first-out”).

Вследствие этого существуют языки, распознаваемые некоторой программой компьютера и нераспознаваемые ни одним магазинным автоматом. В действительности, магазинные автоматы распознают в точности КС-языки. Многие языки контекстно-свободны, включая, как мы видели, некоторые нерегулярные, однако существуют языки, которые просто описываются, но не являются контекстно-свободными. Мы увидим это в разделе 7.2. Примером такого языка является $\{0^n1^n2^n \mid n \geq 1\}$, т.е. множество цепочек, состоящих из одинаковых групп символов 0, 1 и 2.

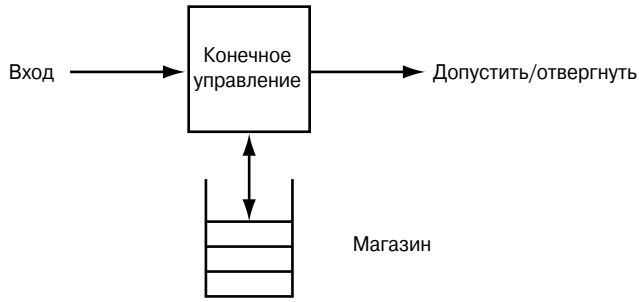


Рис. 6.1. Магазинный автомат как конечный автомат с магазинной структурой данных

Магазинный автомат можно рассматривать неформально как устройство, представленное на рис. 6.1. “Конечное управление” читает входные символы по одному. Магазинный автомат может обозревать символ на вершине магазина и совершать переход на основе текущего состояния, входного символа и символа на вершине магазина. Он может также выполнить “спонтанный” переход, используя ε в качестве входного символа. За один переход автомат совершает следующие действия.

1. Прочитывает и пропускает входной символ, используемый при переходе. Если в качестве входа используется ε, то входные символы не пропускаются.
2. Переходит в новое состояние, которое может и не отличаться от предыдущего.
3. Заменяет символ на вершине магазина некоторой цепочкой. Цепочкой может быть ε, что соответствует снятию с вершины магазина. Это может быть тот же символ, который был ранее на вершине магазина, т.е. магазин не изменяется. Автомат может заменить магазинный символ, что равносильно изменению вершины без снятий и заталкиваний. Наконец, символ может быть заменен несколькими символами — это равносильно тому, что (возможно) изменяется символ на вершине, а затем туда помещаются один или несколько новых символов.

Пример 6.1. Рассмотрим язык $L_{wwr} = \{ww^R \mid w \in (\mathbf{0} + \mathbf{1})^*\}$. Этот язык образован палиндромами четной длины над алфавитом $\{0, 1\}$ и порождается КС-грамматикой (см. рис. 5.1) с исключенными продукциями $P \to 0$ и $P \to 1$.

Дадим следующие неформальное описание магазинного автомата, допускающего L_{wwr}.[1]

1. Работа начинается в состоянии q_0, представляющем "догадку", что не достигнута середина входного слова, т.е. конец слова w, за которым должно следовать его отражение. В состоянии q_0 символы читаются и их копии по очереди записываются в магазин.
2. В любой момент можно предположить, что достигнута середина входа, т.е. конец слова w. В этот момент слово w находится в магазине: левый конец слова на дне магазина, а правый — на вершине. Этот выбор отмечается спонтанным переходом в состояние q_1. Поскольку автомат недетерминирован, в действительности предполагаются обе возможности, т.е. что достигнут конец слова w, но можно оставаться в состоянии q_0 и продолжать читать входные символы и записывать их в магазин.
3. В состоянии q_1 входные символы сравниваются с символами на вершине магазина. Если они совпадают, то входной символ пропускается, магазинный удаляется, и работа продолжается. Если же они не совпадают, то предположение о середине слова неверно, т.е. за предполагаемым w не следует w^R. Эта ветвь вычислений отбрасывается, хотя другие могут продолжаться и вести к тому, что цепочка допускается.
4. Если магазин опустошается, то действительно обнаружен вход w, за которым следует w^R. Прочитанный к этому моменту вход допускается.

□

6.1.2. Формальное определение автомата с магазинной памятью

Формальная запись *магазинного автомата* (МП-автомата) содержит семь компонентов и выглядит следующим образом.

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

Компоненты имеют такой смысл.

Q: конечное множество состояний, как и у конечного автомата.

Σ: конечное множество входных символов, такое же, как у конечного автомата.

Γ: конечный магазинный алфавит. Он не имеет конечноавтоматного аналога и является множеством символов, которые можно помещать в магазин.

δ: функция переходов. Как и у конечных автоматов, δ управляет поведением автомата. Формально, аргументами δ являются тройки $\delta(q, a, X)$, в которых q — состояние из множества Q, a — либо входной символ, либо пустая цепочка ε, которая, по пред-

[1] Можно было бы также построить магазинный автомат для L_{pal}, языка, грамматика которого представлена на рис. 5.1. Однако L_{wwr} несколько проще и позволит нам сосредоточиться на идеях, касающихся магазинных автоматов.

положению, не принадлежит входному алфавиту, X — магазинный символ из Г. Выход δ образуют пары (p, γ), где p — новое состояние, а γ— цепочка магазинных символов, замещающая X на вершине магазина. Например, если $\gamma = \varepsilon$, то магазинный символ снимается, если $\gamma = X$, то магазин не изменяется, а если $\gamma = YZ$, то X заменяется на Z, и Y помещается в магазин.

q_0: начальное состояние. МП-автомат находится в нем перед началом работы.

Z_0: начальный магазинный символ ("маркер дна"). Вначале магазин содержит только этот символ и ничего более.

F: множество допускающих, или заключительных, состояний.

Никаких "смешиваний и сочетаний"

В некоторых случаях МП-автомат может иметь несколько пар на выбор. Например, пусть $\delta(q, a, X) = \{(p, YZ), (r, \varepsilon)\}$. Совершая переход, автомат должен выбрать пару целиком, т.е. нельзя взять состояние из одной пары, а цепочку для замещения в магазине — из другой. Таким образом, имея состояние q, символ X на вершине магазина и a на входе, автомат может либо перейти в состояние p и изменить X на YZ, перейти либо в r и вытолкнуть X. Однако перейти в состояние p и вытолкнуть X или перейти в r и изменить X на YZ нельзя.

Пример 6.2. Построим МП-автомат P, допускающий язык L_{wwr} из примера 6.1. Вначале уточним одну деталь, необходимую для правильной организации работы с магазином. Магазинный символ Z_0 используется для отметки дна магазина. Этот символ должен присутствовать в магазине, чтобы, удалив из магазина w и обнаружив на входе ww^R, можно было совершить переход в допускающее состояние q_2. Итак, наш МП-автомат для языка L_{wwr} можно представить в следующем виде.

$$P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\})$$

Функция δ определяется такими правилами.

1. $\delta(q_0, 0, Z_0) = \{(q_0, 0Z_0)\}$ и $\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}$. Одно из этих правил применяется вначале, когда автомат находится в состоянии q_0 и обозревает начальный символ Z_0 на вершине магазина. Читается первый символ и помещается в магазин; Z_0 остается под ним для отметки дна.
2. $\delta(q_0, 0, 0) = \{(q_0, 00)\}$, $\delta(q_0, 0, 1) = \{(q_0, 01)\}$, $\delta(q_0, 1, 0) = \{(q_0, 10)\}$ и $\delta(q_0, 1, 1) = \{(q_0, 11)\}$. Эти четыре аналогичные правила позволяют оставаться в состоянии q_0 и читать входные символы, помещая каждый из них на вершину магазина над предыдущим верхним символом.

3. $\delta(q_0, \varepsilon, Z_0) = \{(q_1, Z_0)\}$, $\delta(q_0, \varepsilon, 0) = \{(q_1, 0)\}$ и $\delta(q_0, \varepsilon, 1) = \{(q_1, 1)\}$. Эти правила позволяют автомату спонтанно (без чтения входа) переходить из состояния q_0 в состояние q_1, не изменяя верхний символ магазина, каким бы он ни был.
4. $\delta(q_1, 0, 0) = \{(q_1, \varepsilon)\}$ и $\delta(q_1, 1, 1) = \{(q_1, \varepsilon)\}$. В состоянии q_1 входные символы проверяются на совпадение с символами на вершине магазина. При совпадении последние выталкиваются.
5. $\delta(q_1, \varepsilon, Z_0) = \{(q_2, Z_0)\}$. Наконец, если обнаружен маркер дна магазина Z_0 и автомат находится в состоянии q_1, то обнаружен вход вида ww^R. Автомат переходит в состояние q_2 и допускает.

□

6.1.3. Графическое представление МП-автоматов

Функцию δ, заданную списком, как в примере 6.2, отследить нелегко. Иногда особенности поведения МП-автомата становятся более понятными по диаграмме, обобщающей диаграмму переходов конечного автомата. Введем в рассмотрение и используем в дальнейшем *диаграммы переходов* МП-автоматов со следующими свойствами.

1. Вершины соответствуют состояниям МП-автомата.
2. Стрелка, отмеченная словом *Начало*, указывает на начальное состояние, а обведенные двойным кружком состояния являются заключительными, как и у конечных автоматов.
3. Дуги соответствуют переходам МП-автомата в следующем смысле. Дуга, отмеченная $a, X/\alpha$ и ведущая из состояния q в состояние p, означает, что $\delta(q, a, X)$ содержит пару (p, α) (возможно, и другие пары). Таким образом, отметка дуги показывает, какой входной символ используется, а также, что было и что будет на вершине магазина.

Диаграмма не говорит лишь о том, какой магазинный символ является стартовым. По соглашению им будет Z_0, если не оговаривается иное.

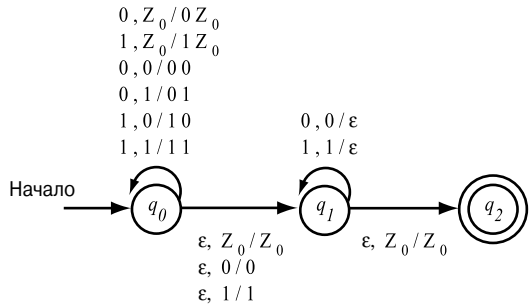


Рис. 6.2. Представление МП-автомата в виде обобщенной диаграммы переходов

Пример 6.3. МП-автомат из примера 6.2 представлен в виде диаграммы на рис. 6.2. □

6.1.4. Конфигурации МП-автомата

Сейчас у нас есть лишь неформальное понятие того, как МП-автомат "вычисляет". Интуитивно МП-автомат переходит от конфигурации к конфигурации в соответствии с входными символами (или ε), но в отличие от конечного автомата, о котором известно только его состояние, конфигурация МП-автомата включает как состояние, так и содержимое магазина. Поскольку магазин может быть очень большим, он часто является наиболее важной частью конфигурации. Полезно также представлять в качестве части конфигурации непрочитанную часть входа.

Таким образом, конфигурация МП-автомата представляется тройкой (q, w, γ), где q — состояние, w — оставшаяся часть входа, γ— содержимое магазина. По соглашению вершина магазина изображается слева, а дно — справа. Такая тройка называется *конфигурацией* МП-автомата, или его *мгновенным описанием*, сокращенно МО (instantaneous description — ID).

Поскольку МО конечного автомата — это просто его состояние, для представления последовательностей конфигураций, через которые он проходил, было достаточно использовать $\hat{\delta}$. Однако для МП-автоматов нужна нотация, описывающая изменения состояния, входа и магазина. Таким образом, используются пары конфигураций, связи между которыми представляют переходы МП-автомата.

Пусть $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — МП-автомат. Определим отношение $\vdash_P$, или просто $\vdash$, когда P подразумевается, следующим образом. Предположим, что $\delta(q, a, X)$ содержит (p, α). Тогда для всех цепочек w из Σ^* и β из Γ^* полагаем $(q, aw, X\beta) \vdash (p, w, \alpha\beta)$.

Этот переход отражает идею того, что, прочитывая на входе символ a, который может быть ε, и заменяя X на вершине магазина цепочкой α, можно перейти из состояния q в состояние p. Заметим, что оставшаяся часть входа (w) и содержимое магазина под его вершиной (β) не влияют на действие МП-автомата; они просто сохраняются, возможно, для того, чтобы влиять на события в дальнейшем.

Используем также символ $\vdash_P^*$, или просто $\vdash^*$, когда МП-автомат P подразумевается, для представления нуля или нескольких переходов МП-автомата. Итак, имеем следующее индуктивное определение.

Базис. $I \vdash^* I$ для любого МО I.

Индукция. $I \vdash^* J$, если существует некоторое МО K, удовлетворяющее условиям $I \vdash K$ и $K \vdash^* J$.

Таким образом, $I \overset{*}{\vdash} J$, если существует такая последовательность МО $K_1, K_2, \ldots, K_n$, у которой $I = K_1$, $J = K_n$, и $K_i \vdash K_{i+1}$ для всех $i = 1, 2, \ldots, n - 1$.

Пример 6.4. Рассмотрим работу МП-автомата из примера 6.2 со входом 1111. Поскольку q_0 — начальное состояние, а Z_0 — стартовый символ, то начальным МО будет $(q_0, 1111, Z_0)$. На этом входе МП-автомат имеет возможность несколько раз делать ошибочные предположения. Вся последовательность МО, достижимых из начальной конфигурации, показана на рис. 6.3. Стрелки представляют отношение $\vdash$.

Из начального МО можно выбрать два перехода. Первый предполагает, что середина не достигнута и ведет к МО $(q_0, 111, 1Z_0)$. В результате символ 1 перемещается в магазин.

Второй выбор основан на предположении, что достигнута середина входа. Без прочитывания очередного символа МП-автомат переходит в состояние q_1, что приводит к МО $(q_1, 1111, Z_0)$. Поскольку МП-автомат может допустить вход, если он находится в состоянии q_1 и обозревает Z_0 на вершине магазина, он переходит к МО $(q_2, 1111, Z_0)$. Это МО не является в точности допускающим, так как вход не дочитан до конца. Если бы входом вместо 1111 было ε, то та же самая последовательность переходов привела к МО (q_0, ε, Z_0), показывающему, что ε допущено.

МП-автомат может также предположить, что он увидел середину после чтения первой 1, т.е. находясь в конфигурации $(q_0, 111, 1Z_0)$. Это предположение также ведет к ошибке, поскольку вход не может быть дочитан до конца. Правильное предположение, что середина достигается после прочтения 11, дает последовательность МО $(q_0, 1111, Z_0) \vdash (q_0, 111, 1Z_0) \vdash (q_0, 11, 11Z_0) \vdash (q_1, 11, 11Z_0) \vdash (q_1, 1, 1Z_0) \vdash (q_1, \varepsilon, Z_0) \vdash (q_2, \varepsilon, Z_0)$. □

Соглашения по записи МП-автоматов

Продолжим соглашения об использовании символов, введенные для конечных автоматов и грамматик. Придерживаясь системы записи, полезно представлять себе, что магазинные символы играют роль, аналогичную объединению терминалов и переменных в КС-грамматиках.

1. Символы входного алфавита представлены строчными буквами из начала алфавита, например, a или b.
2. Состояния обычно представляются буквами p и q или другими, близкими к ним в алфавитном порядке.
3. Цепочки входных символов обозначаются строчными буквами из конца алфавита, например, w или z.
4. Магазинные символы представлены прописными буквами из конца алфавита, например, X или Y.
5. Цепочки магазинных символов обозначаются греческими буквами, например, α или β.

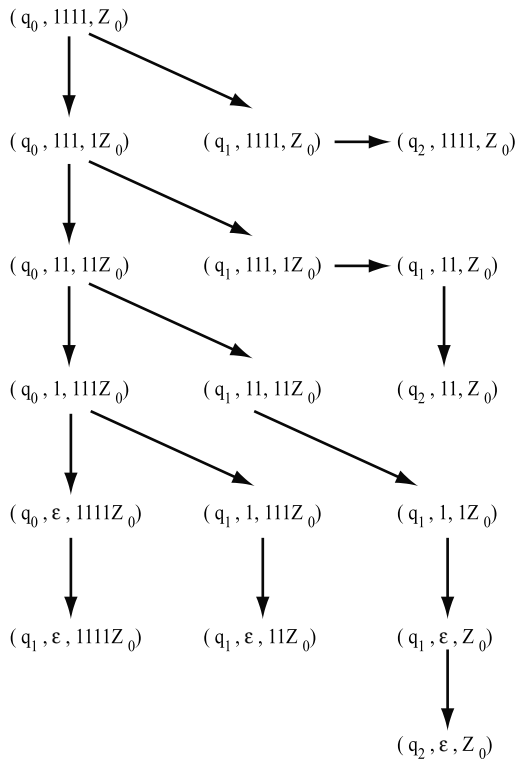


Рис. 6.3. Конфигурации МП-автомата из примера 6.2 при входе 1111

Для дальнейших рассуждений о МП-автоматах понадобятся следующие три важных принципа.

1. Если последовательность конфигураций (*вычисление*) является допустимой (legal) для МП-автомата *P* (с точки зрения его определения), то вычисление, образованное путем дописывания одной и той же цепочки к концам входных цепочек всех его конфигураций, также допустимо.
2. Если вычисление допустимо для МП-автомата *P*, то вычисление, образованное дописыванием одних и тех же магазинных символов внизу магазина в каждой конфигурации, также допустимо.
3. Если вычисление допустимо для МП-автомата *P*, и в результате некоторый суффикс (tail) входной цепочки не прочитан, то вычисление, полученное путем удаления этого суффикса из входных цепочек каждой конфигурации, также допустимо.

Интуитивно данные, которые никогда не читаются МП-автоматом, не влияют на его вычисления. Формализуем приведенные пункты 1 и 2 в виде следующей теоремы.

Теорема 6.5. Если $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — МП-автомат и $(q, x, \alpha) \overset{*}{\vdash}_P (p, y, \beta)$, то для любой цепочки w из Σ^* и γ из Γ^* верно следующее утверждение.

$$(q, xw, \alpha\gamma) \overset{*}{\vdash}_P (p, yw, \beta\gamma)$$

Заметим, что если $\gamma = \varepsilon$, то получается формальное утверждение приведенного выше принципа 1, а при $w = \varepsilon$ — принципа 2.

Доказательство. Доказательство проводится очень просто индукцией по числу шагов в последовательности МО, приводящих $(q, xw, \alpha\gamma)$ к $(p, yw, \beta\gamma)$. Каждый из переходов в последовательности $(q, x, \alpha) \overset{*}{\vdash}_P (p, y, \beta)$ обосновывается переходами P без какого-либо использования w и/или γ. Следовательно, каждый переход обоснован и в случае, когда эти цепочки присутствуют на входе и в магазине. □

Заметим, что обращение этой теоремы неверно. Существуют действия, которые МП-автомат мог бы совершить с помощью выталкивания из стека, т.е. используя некоторые символы γ и заменяя их в магазине, что невозможно без обработки γ. Однако, как утверждает принцип 3, мы можем удалить неиспользуемый вход, так как МП-автомат не может прочитать входные символы, а затем восстановить их на входе. Сформулируем принцип 3 следующим образом.

Нужны ли конфигурации конечных автоматов?

Можно было бы удивиться, почему понятие конфигурации не было введено для конечных автоматов. Хотя конечный автомат не имеет магазина, в качестве МО для него можно было бы использовать пару (q, w), где q — состояние, а w — остаток входа. Определить такие конфигурации можно, но их достижимость не дает никакой новой информации по сравнению с тем, что давало использование $\hat{\delta}$. Иными словами, для любого конечного автомата можно показать, что $\hat{\delta}(q, w) = p$ тогда и только тогда, когда $(q, wx) \overset{*}{\vdash} (p, x)$ для всех цепочек x. Тот факт, что x может быть чем угодно, не влияющим на поведение конечного автомата, является теоремой, аналогичной теоремам 6.5 и 6.6.

Теорема 6.6. Если $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — МП-автомат и $(q, xw, \alpha) \overset{*}{\vdash}_P (p, yw, \beta)$, то верно также, что $(q, x, \alpha) \overset{*}{\vdash}_P (p, y, \beta)$. □

6.1.5. Упражнения к разделу 6.1

6.1.1. Предположим, что МП-автомат $P = (\{q, p\}, \{0, 1\}, \{Z_0, X\}, \delta, q, Z_0, \{p\})$ имеет следующую функцию переходов.

1. $\delta(q, 0, Z_0) = \{(q, XZ_0)\}$.

2. $\delta(q, 0, X) = \{(q, XX)\}$.
3. $\delta(q, 1, X) = \{(q, X)\}$.
4. $\delta(q, \varepsilon, X) = \{(p, \varepsilon)\}$.
5. $\delta(p, \varepsilon, X) = \{(p, \varepsilon)\}$.
6. $\delta(p, 1, X) = \{(p, XX)\}$.
7. $\delta(p, 1, Z_0) = \{(p, \varepsilon)\}$.

Приведите все конфигурации, достижимые из начального МО (q, w, Z_0), если входным словом w является:

а) 01;

б) 0011;

в) 010.

6.2. Языки МП-автоматов

Мы предполагали, что МП-автомат допускает свой вход, прочитывая его и достигая заключительного состояния. Такой подход называется “допуск по заключительному состоянию”. Существует другой способ определения языка МП-автомата, имеющий важные приложения. Для любого МП-автомата мы можем определить язык, “допускаемый по пустому магазину”, т.е. множество цепочек, приводящих МП-автомат в начальной конфигурации к опустошению магазина.

Эти два метода эквивалентны в том смысле, что для языка L найдется МП-автомат, допускающий его по заключительному состоянию тогда и только тогда, когда для L найдется МП-автомат, допускающий его по пустому магазину. Однако для конкретных МП-автоматов языки, допускаемые по заключительному состоянию и по пустому магазину, обычно различны. В этом разделе показывается, как преобразовать МП-автомат, допускающий L по заключительному состоянию, в другой МП-автомат, который допускает L по пустому магазину, и наоборот.

6.2.1. Допустимость по заключительному состоянию

Пусть $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — МП-автомат. Тогда $L(P)$, языком, допускаемым P по заключительному состоянию, является

$$\{w \mid (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \alpha)\}$$

для некоторого состояния q из F и произвольной магазинной цепочки α. Таким образом, начиная со стартовой конфигурации с w на входе, P прочитывает w и достигает допускающего состояния. Содержимое магазина в этот момент не имеет значения.

Пример 6.7. Мы утверждали, что МП-автомат из примера 6.2 допускает язык L_{wwr}, состоящий из цепочек вида ww^R в алфавите {0, 1}. Выясним, почему это утверждение верно. Оно имеет вид "тогда и только тогда, когда": МП-автомат P из примера 6.2 допускает цепочку x по заключительному состоянию тогда и только тогда, когда x имеет вид ww^R.

(*Достаточность*) Эта часть проста; нужно показать лишь допускающее вычисление P. Если $x = ww^R$, то заметим, что справедливы следующие соотношения.

$$(q_0, ww^R, Z_0) \overset{*}{\vdash} (q_0, w^R, w^RZ_0) \vdash (q_1, w^R, w^RZ_0) \overset{*}{\vdash} (q_1, \varepsilon, Z_0) \vdash (q_2, \varepsilon, Z_0)$$

Таким образом, МП-автомат имеет возможность прочитать цепочку w на входе и записать ее символы в свой магазин в обратном порядке. Затем он спонтанно переходит в состояние q_1 и проверяет совпадение w^R на входе с такой же цепочкой в магазине, и наконец, спонтанно переходит в состояние q_2.

(*Необходимость*) Эта часть труднее. Во-первых, заметим, что единственный путь достижения состояния q_2 состоит в том, чтобы находиться в состоянии q_1 и иметь Z_0 на вершине магазина. Кроме того, любое допускающее вычисление P начинается в состоянии q_0, совершает один переход в q_1 и никогда не возвращается в q_0. Таким образом, достаточно найти условия, налагаемые на x, для которых $(q_0, x, Z_0) \overset{*}{\vdash} (q_1, \varepsilon, Z_0)$; именно такие цепочки и допускает P по заключительному состоянию. Покажем индукцией по $|x|$ следующее несколько более общее утверждение.

- Если $(q_0, x, \alpha) \overset{*}{\vdash} (q_1, \varepsilon, \alpha)$, то x имеет вид ww^R.

Базис. Если $x = \varepsilon$, то x имеет вид ww^R, с $w = \varepsilon$. Таким образом, заключение верно, и утверждение истинно. Отметим, что нам не обязательно доказывать истинность гипотезы $(q_0, \varepsilon, \alpha) \overset{*}{\vdash} (q_1, \varepsilon, \alpha)$, хотя она и верна.

Индукция. Пусть $x = a_1a_2\ldots a_n$ для некоторого $n > 0$. Существуют следующие два перехода, которые P может совершить из МО (q_0, x, α).

1. $(q_0, x, \alpha) \vdash (q_1, x, \alpha)$. Теперь P может только выталкивать из магазина, находясь в состоянии q_1. P должен вытолкнуть символ из магазина с чтением каждого входного символа, и $|x| > 0$. Таким образом, если $(q_1, x, \alpha) \overset{*}{\vdash} (q_1, \varepsilon, \beta)$, то цепочка β короче, чем α, и не может ей равняться.
2. $(q_0, a_1a_2\ldots a_n, \alpha) \vdash (q_0, a_2\ldots a_n, a_1\alpha)$. Теперь последовательность переходов может закончиться в $(q_1, \varepsilon, \alpha)$, только если последний переход является следующим выталкиванием.

   $(q_1, a_n, a_1\alpha) \vdash (q_1, \varepsilon, \alpha)$

   В этом случае должно выполняться $a_1 = a_n$. Нам также известно, что

   $(q_0, a_2\ldots a_n, a_1\alpha) \overset{*}{\vdash} (q_1, a_n, a_1\alpha)$

По теореме 6.6 символ a_n можно удалить из конца входа, поскольку он не используется. Тогда

$$(q_0, a_2 \ldots a_{n-1}, a_1\alpha) \overset{*}{\vdash} (q_1, \varepsilon, a_1\alpha)$$

Поскольку вход у этой последовательности короче, чем n, можно применить индуктивное предположение и заключить, что $a_2 \ldots a_{n-1}$ имеет вид yy^R для некоторого y. Поскольку $x = a_1 y y^R a_n$, и нам известно, что $a_1 = a_n$, делаем вывод, что x имеет вид ww^R; в частности $w = a_1 y$.

Приведенные рассуждения составляют сердцевину доказательства того, что x допускается только тогда, когда равен ww^R для некоторого w. Таким образом, необходимость доказана. Вместе с достаточностью, доказанной ранее, она гласит, что P допускает в точности цепочки из L_{wwr}. □

6.2.2. Допустимость по пустому магазину

Для каждого МП-автомата $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ определим

$$N(P) = \{w \mid (q_0, w, Z_0) \overset{*}{\vdash} (q, \varepsilon, \varepsilon)\},$$

где q — произвольное состояние. Таким образом, $N(P)$ представляет собой множество входов w, которые P может прочитать, одновременно опустошив свой магазин.[2]

Пример 6.8. МП-автомат P из примера 6.2 никогда не опустошает свой магазин, поэтому $N(P) = \varnothing$. Однако небольшое изменение позволит автомату P допускать L_{wwr} как по пустому магазину, так и по заключительному состоянию. Вместо перехода $\delta(q_1, \varepsilon, Z_0) =$
$\{(q_2, Z_0)\}$ используем $\delta(q_1, \varepsilon, Z_0) = \{(q_2, \varepsilon)\}$. Теперь P выталкивает последний символ из магазина, когда допускает, поэтому $L(P) = N(P) = L_{wwr}$. □

Поскольку множество допускающих состояний не имеет значения, иногда в случаях, когда нас будет интересовать допуск только по пустому магазину, будем записывать МП-автомат в виде шестерки $(Q, \Sigma, \Gamma, \delta, q_0, Z_0)$, опуская седьмой компонент.

6.2.3. От пустого магазина к заключительному состоянию

Покажем, что классы языков, допускаемых МП-автоматами по заключительному состоянию и по пустому магазину, совпадают. В разделе 6.3 будет доказано, что МП-автоматы определяют КС-языки. Наша первая конструкция показывает, как, исходя из МП-автомата P_N, допускающего язык L по пустому магазину, построить МП-автомат P_F, допускающий L по заключительному состоянию.

Теорема 6.9. Если $L = N(P_N)$ для некоторого МП-автомата $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$, то существует такой МП-автомат P_F, у которого $L = L(P_F)$.

---

[2] Символ N в $N(P)$ обозначает “пустой магазин” (“null stack”).

Доказательство. Идея доказательства представлена на рис. 6.4. Используется новый символ X_0, который не должен быть элементом Γ; X_0 является как стартовым символом автомата P_F, так и маркером дна магазина, позволяющим узнать, когда P_N опустошает магазин. Таким образом, если P_F обозревает X_0 на вершине магазина, то он знает, что P_N опустошает свой магазин на этом же входе.

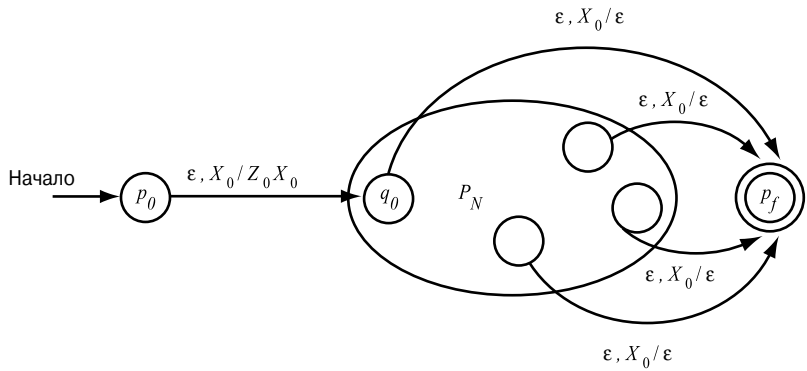


Рис. 6.4. P_F *имитирует* P_N *и допускает, если* P_N *опустошает свой магазин*

Нам нужно также новое начальное состояние p_0, единственной функцией которого является затолкнуть Z_0, стартовый символ автомата P_N, на вершину магазина и перейти в состояние q_0, начальное для P_N. Далее P_F имитирует P_N до тех пор, пока магазин P_N не станет пустым, что P_F определяет по символу X_0 на вершине своего магазина. И наконец, нужно еще одно новое состояние p_f, заключительное в автомате P_F; он переходит в p_f, как только обнаруживает, что P_N опустошает свой магазин.

Описание P_F имеет следующий вид.

$$P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$$

Функция δ_F определяется таким образом.

1. $\delta_F(p_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$. В своем начальном состоянии P_F спонтанно переходит в начальное состояние автомата P_N, заталкивая его стартовый символ Z_0 в магазин.
2. Для всех состояний q из Q, входов a из Σ (или $a = \varepsilon$) и магазинных символов Y из Γ, $\delta_F(q, a, Y)$ содержит все пары из $\delta_N(q, a, Y)$.
3. В дополнение к правилу (2), $\delta_F(q, \varepsilon, X_0)$ содержит (p_f, ε) для каждого состояния q из Q.

Докажем, что $w \in L(P_F)$ тогда и только тогда, когда $w \in N(P_N)$.

(*Достаточность*) Нам дано, что $(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, \varepsilon)$ для некоторого состояния q. По теореме 6.5 можно добавить X_0 на дно магазина и заключить, что $(q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, X_0)$. Поскольку по правилу 2 у P_F есть все переходы P_N, можно утверждать также,

что $(q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, X_0)$. Если эту последовательность переходов собрать вместе с начальным и заключительным переходами в соответствии с правилами 1 и 3, то получим следующее.

$$(p_0, w, X_0) \underset{P_F}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_F}{\vdash}} (q, \varepsilon, X_0) \underset{P_F}{\vdash} (p_f, \varepsilon, \varepsilon) \tag{6.1}$$

Таким образом, P_F допускает w по заключительному состоянию.

(*Необходимость*) Заметим, что дополнительные переходы по правилам 1 и 3 дают весьма ограниченные возможности для допускания w по заключительному состоянию. Правило 3 должно использоваться только на последнем шаге, и его можно использовать, если магазин P_F содержит лишь X_0. Но X_0 не появляется нигде в магазине, кроме его дна. Правило 1 используется только на первом шаге, и оно *должно* использоваться на первом шаге.

Таким образом, любое вычисление в автомате P_F, допускающее w, должно выглядеть как последовательность (6.1). Более того, середина вычисления (все переходы, кроме первого и последнего) должна также быть вычислением в P_N с X_0 под магазином. Это обусловлено тем, что, за исключением первого и последнего шагов, P_F не может использовать переходы, которые не являются переходами P_N, и X_0 не может оказаться на вершине магазина (иначе вычисление заканчивалось бы на следующем шаге). Таким образом, $(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, \varepsilon)$, и $w \in N(P_N)$. □

Пример 6.10. Построим МП-автомат, который обрабатывает последовательности слов `if` и `else` в С-программе, где i обозначает `if`, а e — `else`. Как показано в разделе 5.3.1, в любом префиксе программы количество `else` не может превышать числа `if`, поскольку в противном случае слову `else` нельзя сопоставить предшествующее ему `if`. Итак, магазинный символ Z используется для подсчета разницы между текущими количествами просмотренных i и e. Этот простой МП-автомат с единственным состоянием представлен диаграммой переходов на рис. 6.5.

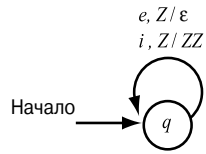


Рис. 6.5. МП-автомат, допускающий ошибки if/else по пустому магазину

Увидев i, автомат заталкивает Z, а e — выталкивает. Поскольку он начинает с одним Z в магазине, в действительности он следует правилу, что если в магазине Z^n, то символов i прочитано на $n - 1$ больше, чем e. В частности, если магазин пуст, то символов e прочитано на один больше, чем i, и входная последовательность недопустима в С-прог-

рамме. Именно такие цепочки наш МП-автомат P_N допускает по пустому магазину. Его формальное определение имеет следующий вид.

$$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$$

Функция δ_N задается таким образом.

1. $\delta_N(q, i, Z) = \{(q, ZZ)\}$. По этому правилу заталкивается Z при i на входе.
2. $\delta_N(q, e, Z) = \{(q, \varepsilon)\}$. Z выталкивается при входе e.

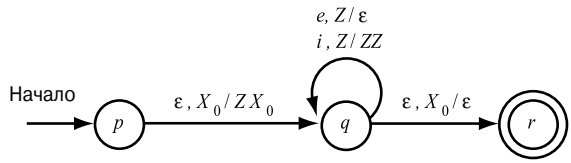


Рис. 6.6. Конструкция МП-автомата, допускающего по заключительному состоянию и построенного на основе автомата P_N (см. рис. 6.5)

Теперь на основе P_N построим МП-автомат P_F, допускающий этот же язык по заключительному состоянию; его диаграмма переходов показана на рис. 6.6.[3] Вводим новые начальное и заключительное состояния p и r, а также используем X_0 в качестве маркера дна магазина. Формально автомат P_F определяется следующим образом.

$$P_F = (\{p, q, r\}, \{i, e\}, \{Z, X_0\}, \delta_F, q, Z)$$

Функция δ_F задается таким образом.

1. $\delta_F(p, \varepsilon, X_0) = \{(q, ZX_0)\}$. По этому правилу P_F начинает имитировать P_N с маркером дна магазина.
2. $\delta_F(q, i, Z) = \{(q, ZZ)\}$. Z заталкивается при входе i, как у P_N.
3. $\delta_F(q, e, Z) = \{(q, \varepsilon)\}$. Z выталкивается при входе e, как у P_N.
4. $\delta_F(q, e, X_0) = \{(r, \varepsilon)\}$. P_F допускает, когда имитируемый P_N опустошает свой магазин.

□

6.2.4. От заключительного состояния к пустому магазину

Теперь пойдем в обратном направлении: исходя из МП-автомата P_F, допускающего язык L по заключительному состоянию, построим другой МП-автомат P_N, который допускает L по пустому магазину. Конструкция проста и представлена на рис. 6.7. Добавляется переход по ε из каждого допускающего состояния автомата P_F в новое состояние p. Находясь в состоянии p, P_N опустошает магазин и ничего не прочитывает на входе.

[3] Не обращайте внимания на то, что тут использованы новые состояния p и r, хотя в конструкции теоремы 6.9 указаны p_0 и p_f. Имена состояний могут быть совершенно произвольными.

Таким образом, как только P_F приходит в допускающее состояние после прочтения w, P_N опустошает свой магазин, также прочитав w.

Во избежание имитации случая, когда P_F случайно опустошает свой магазин без допуска, P_N должен также использовать маркер X_0 на дне магазина. Маркер является его стартовым символом и аналогично конструкции теоремы 6.9 P_N должен начинать работу в новом состоянии p_0, единственная функция которого — затолкнуть стартовый символ автомата P_F в магазин и перейти в начальное состояние P_F. В сжатом виде конструкция представлена на рис. 6.7, а ее формальное описание приводится в следующей теореме.

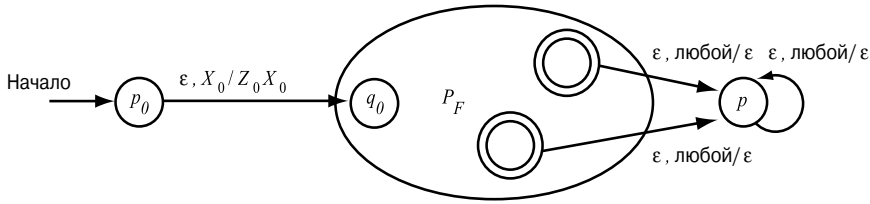


Рис. 6.7. P_N *имитирует* P_F *и опустошает свой магазин тогда и только тогда, когда* P_F *достигает заключительного состояния*

Теорема 6.11. Пусть $L = L(P_F)$ для некоторого МП-автомата $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$. Тогда существует такой МП-автомат P_N, у которого $L = N(P_N)$.

Доказательство. Конструкция соответствует рис. 6.7. Пусть

$$P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0),$$

где δ_N определена следующим образом.

1. $\delta_N(p_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$. Работа начинается с заталкивания стартового символа автомата P_F в магазин и перехода в его начальное состояние.
2. Для всех состояний q из Q, входов a из Σ или $a = \varepsilon$ и магазинных символов Y из Γ $\delta_N(q, a, Y)$ содержит все пары из $\delta_F(q, a, Y)$, т.е. P_N имитирует P_F.
3. Для всех допускающих состояний q из F и магазинных символов Y из $\Gamma \cup \{X_0\}$ $\delta_N(q, \varepsilon, Y)$ содержит (p, ε). По этому правилу, как только P_F допускает, P_N может начать опустошение магазина без чтения входных символов.
4. Для всех магазинных символов Y из $\Gamma \cup \{X_0\}$, $\delta_N(p, \varepsilon, Y) = \{(p, \varepsilon)\}$. Попав в состояние p (только в случае, когда P_F допускает), P_N выталкивает символы из магазина до его опустошения. Входные символы при этом не читаются.

Теперь нужно доказать, что $w \in N(P_N)$ тогда и только тогда, когда $w \in L(P_F)$. Идеи аналогичны теореме 6.9. Достаточность представляет собой непосредственную имитацию, а необходимость требует проверки ограниченного числа действий, которые может совершить МП-автомат P_N.

(*Достаточность*) Пусть $(q_0, w, Z_0) \overset{*}{\underset{P_F}{\vdash}} (q, \varepsilon, \alpha)$ для некоторого допускающего состояния q и цепочки α в магазине. Вспомним, что каждый переход P_F есть и у P_N, и что теорема 6.5 разрешает нам держать X_0 в магазине под символами из Г. Тогда $(q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, \alpha X_0)$. Следовательно, P_N может совершить следующие действия.

$$(p_0, w, X_0) \underset{P_N}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, \alpha X_0) \overset{*}{\underset{P_N}{\vdash}} (p, \varepsilon, \varepsilon)$$

Первый переход соответствует правилу 1 построения P_N, а последняя последовательность переходов — правилам 3 и 4. Таким образом, P_N допускает w по пустому магазину.

(*Необходимость*) Единственный путь, по которому P_N может опустошить свой магазин, состоит в достижении состояния p, так как X_0 находится в магазине и является символом, для которого у P_F переходы не определены. P_N может достичь состояния p только тогда, когда P_F приходит в допускающее состояние. Первым переходом автомата P_N может быть только переход, заданный правилом 1. Таким образом, каждое допускающее вычисление P_N выглядит следующим образом (q — допускающее состояние автомата P_F).

$$(p_0, w, X_0) \underset{P_N}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, \alpha X_0) \overset{*}{\underset{P_N}{\vdash}} (p, \varepsilon, \varepsilon)$$

Кроме того, между МО (q_0, w, Z_0X_0) и $(q, \varepsilon, \alpha X_0)$ все переходы являются переходами автомата P_F. В частности, X_0 никогда не появляется на вершине магазина до достижения МО $(q, \varepsilon, \alpha X_0)$.[4] Таким образом, приходим к выводу, что у P_F есть такое же вычисление, но без X_0 в магазине, т.е. $(q_0, w, Z_0) \overset{*}{\underset{P_F}{\vdash}} (q, \varepsilon, \alpha)$. Итак, P_F допускает w по заключительному состоянию, т.е. $w \in L(P_F)$. □

6.2.5. Упражнения к разделу 6.2

6.2.1. Постройте МП-автоматы, допускающие следующие языки. Можно использовать допускание как по заключительному состоянию, так и по пустому магазину — что удобнее:

а) (*) $\{0^n1^n \mid n \geq 1\}$;

б) множество всех цепочек из 0 и 1, в префиксах которых количество символов 1 не больше количества символов 0;

в) множество всех цепочек из 0 и 1 с одинаковыми количествами символов 0 и 1.

[4] Хотя α может быть ε, и в этом случае P_F опустошает свой магазин и одновременно допускает.

6.2.2. (!) Постройте МП-автоматы, допускающие следующие языки:

а) (*) $\{a^i b^j c^k \mid i = j$ или $j = k\}$. Заметим, что этот язык отличается от языка из упражнения 5.1.1, *б*;

б) множество всех цепочек из 0 и 1, у которых количество символов 0 вдвое больше количества символов 1.

6.2.3. (!!) Постройте МП-автоматы, допускающие следующие языки:

а) $\{a^i b^j c^k \mid i \neq j$ или $j \neq k\}$;

б) множество всех цепочек из символов a и b, которые *не* имеют вида ww, т.е. не являются повторениями никакой цепочки.

6.2.4. Пусть P — МП-автомат, допускающий по пустому магазину язык $L = N(P)$, и пусть $\varepsilon \notin L$. Опишите, как изменить P, чтобы он допускал $L \cup \{\varepsilon\}$ по пустому магазину.

6.2.5. МП-автомат $P = (\{q_0, q_1, q_2, q_3, f\}, \{a, b\}, \{Z_0, A, B\}, \delta, q_0, Z_0, \{f\})$ имеет следующее определение δ.

$\delta(q_0, a, Z_0) = (q_1, AAZ_0)$ $\quad$ $\delta(q_0, b, Z_0) = (q_2, BZ_0)$ $\quad$ $\delta(q_0, \varepsilon, Z_0) = (f, \varepsilon)$

$\delta(q_1, a, A) = (q_1, AAA)$ $\quad$ $\delta(q_1, b, A) = (q_1, \varepsilon)$ $\quad$ $\delta(q_1, \varepsilon, Z_0) = (q_0, Z_0)$

$\delta(q_2, a, B) = (q_3, \varepsilon)$ $\quad$ $\delta(q_2, b, B) = (q_2, BB)$ $\quad$ $\delta(q_2, \varepsilon, Z_0) = (q_0, Z_0)$

$\delta(q_3, \varepsilon, B) = (q_2, \varepsilon)$ $\quad$ $\delta(q_3, \varepsilon, Z_0) = (q_1, AZ_0)$

Фигурные скобки опущены, поскольку каждое из указанных множеств имеет только один выбор перехода.

а) (*) приведите трассу выполнения (последовательность МО), по которой видно, что $bab \in L(P)$;

б) приведите трассу выполнения, показывающую, что $abb \in L(P)$;

в) укажите содержимое магазина после того, как P прочитал b^7a^4 на входе;

г) (!) дайте неформальное описание $L(P)$.

6.2.6. Рассмотрим МП-автомат P из упражнения 6.1.1:

а) преобразуйте P в другой МП-автомат P_1, допускающий по пустому магазину тот же язык, который допускается P по заключительному состоянию, т.е. $N(P_1) = L(P)$;

б) постройте МП-автомат P_2 такой, что $L(P_2) = N(P)$, т.е. P_2 допускает по заключительному состоянию то, что P допускает по пустому магазину.

6.2.7. (!) Покажите, что если P — МП-автомат, то существует МП-автомат P_2, у которого только два магазинных символа и $L(P_2) = L(P)$. *Указание*. Рассмотрите двоичное представление магазинных символов P.

6.2.8. (***!**) МП-автомат называется *ограниченным*, если при любом переходе он может увеличивать высоту магазина не более, чем на один символ. Таким образом, если (p, γ) содержится в функции переходов, то $|\gamma| \leq 2$. Докажите, что если P — МП-автомат, то существует ограниченный МП-автомат P_3, для которого $L(P_3) = L(P)$.

6.3. Эквивалентность МП-автоматов и КС-грамматик

В этом разделе мы покажем, что МП-автоматы определяют КС-языки. План доказательства изображен на рис. 6.8. Цель состоит в том, чтобы доказать равенство следующих классов языков.

1. Класс КС-языков, определяемых КС-грамматиками.
2. Класс языков, допускаемых МП-автоматами по заключительному состоянию.
3. Класс языков, допускаемых МП-автоматами по пустому магазину.

Мы уже показали, что классы 2 и 3 равны. После этого достаточно доказать, что совпадают классы 1 и 2.

Рис. 6.8. Организация конструкций, показывающих эквивалентность трех способов определения КС-языков

6.3.1. От грамматик к МП-автоматам

По данной грамматике G строится МП-автомат, имитирующий ее левые порождения. Любую левовыводимую цепочку, которая не является терминальной, можно записать в виде $xA\alpha$, где A — крайняя слева переменная, x — цепочка терминалов слева от A, α — цепочка терминалов и переменных справа. $A\alpha$ называется *остатком* (tail) этой левовыводимой цепочки. У терминальной левовыводимой цепочки остатком является ε.

Идея построения МП-автомата по грамматике состоит в том, чтобы МП-автомат имитировал последовательность левовыводимых цепочек, используемых в грамматике для порождения данной терминальной цепочки w. Остаток каждой цепочки $A\alpha$ появляется в магазине с переменной A на вершине. При этом x "представлен" прочитанными на входе символами, а суффикс цепочки w после x считается непрочитанным.

Предположим, что МП-автомат находится в конфигурации $(q, y, A\alpha)$, представляющей левовыводимую цепочку $xA\alpha$. Он угадывает продукцию, используемую для расши-

рения A, скажем, $A \to \beta$. Переход автомата состоит в том, что A на вершине магазина заменяется цепочкой β, и достигается МО $(q, y, \beta\alpha)$. Заметим, что у этого МП-автомата есть всего одно состояние, q.

Теперь $(q, y, \beta\alpha)$ может не быть представлением следующей левовыводимой цепочки, поскольку β может иметь терминальный префикс. В действительности, β может вообще не иметь переменных, а у α может быть терминальный префикс. Все терминалы в начале цепочки $\beta\alpha$ нужно удалить до появления следующей переменной на вершине магазина. Эти терминалы сравниваются со следующими входными символами для того, чтобы убедиться, что наши предположения о левом порождении входной цепочки w правильны; в противном случае данная ветвь вычислений МП-автомата отбрасывается.

Если таким способом нам удается угадать левое порождение w, то в конце концов мы дойдем до левовыводимой цепочки w. В этот момент все символы в магазине или расширены (если это переменные), или совпали с входными (если это терминалы). Магазин пуст, и мы допускаем по пустому магазину.

Уточним приведенное неформальное описание. Пусть $G = (V, T, Q, S)$ — КС-грамматика. Построим МП-автомат $P = (\{q\}, T, V \cup T, \delta, q, S)$, который допускает $L(G)$ по пустому магазину. Функция переходов δ определена таким образом:

1. $\delta(q, \varepsilon, A) = \{(q, \beta) \mid A \to \beta$ — продукция $G\}$ для каждой переменной A.
2. $\delta(q, a, a) = \{(q, \varepsilon)\}$ для каждого терминала a.

Пример 6.12. Преобразуем грамматику выражений (см. рис. 5.2) в МП-автомат. Напомним эту грамматику.

$$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$
$$E \to I \mid E * E \mid E + E \mid (E)$$

Множеством входных символов для МП-автомата является $\{a, b, 0, 1, (,), +, *\}$. Эти восемь символов вместе с переменными I и E образуют магазинный алфавит. Функция переходов определена следующим образом:

а) $\delta(q, \varepsilon, I) = \{(q, a), (q, b), (q, Ia), (q, Ib), (q, I0), (q, I1)\}$;

б) $\delta(q, \varepsilon, E) = \{(q, I), (q, E + E), (q, E * E), (q, (E))\}$;

в) $\delta(q, a, a) = \{(q, \varepsilon)\}$; $\delta(q, b, b) = \{(q, \varepsilon)\}$; $\delta(q, 0, 0) = \{(q, \varepsilon)\}$; $\delta(q, 1, 1) = \{(q, \varepsilon)\}$; $\delta(q, (, () = \{(q, \varepsilon)\}$; $\delta(q,),)) = \{(q, \varepsilon)\}$; $\delta(q, +, +) = \{(q, \varepsilon)\}$; $\delta(q, *, *) = \{(q, \varepsilon)\}$.

Заметим, что пункты (а) и (б) появились по правилу 1, а восемь переходов (в) — по правилу 2. Других переходов у МП-автомата нет.

Теорема 6.13. Если МП-автомат P построен по грамматике G в соответствии с описанной выше конструкцией, то $N(P) = L(G)$.

Доказательство. Докажем, что $w \in N(P)$ тогда и только тогда, когда $w \in L(G)$.

(*Достаточность*) Пусть $w \in L(G)$. Тогда w имеет следующее левое порождение.

$$S = \gamma_1 \underset{lm}{\Rightarrow} \gamma_2 \underset{lm}{\Rightarrow} \ldots \underset{lm}{\Rightarrow} \gamma_n = w$$

Покажем индукцией по i, что $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$, где y_i и α_i представляют левовыводимую цепочку γ_i. Точнее, пусть α_i является остатком γ_i, и $\gamma_i = x_i\alpha_i$. Тогда y_i — это такая цепочка, что $x_iy_i = w$, т.е. то, что остается на входе после чтения x_i.

Базис. $\gamma_1 = S$ при $i = 1$. Таким образом, $x_1 = \varepsilon$, и $y_1 = w$. Поскольку $(q, w, S) \overset{*}{\vdash} (q, w, S)$ через 0 переходов, базис доказан.

Индукция. Теперь рассмотрим вторую и последующие левовыводимые цепочки. Предположим, что

$$(q, w, S) \overset{*}{\vdash} (q, y_i, \alpha_i)$$

и докажем, что $(q, w, S) \overset{*}{\vdash} (q, y_{i+1}, \alpha_{i+1})$. Поскольку α_i является остатком, он начинается переменной A. Кроме того, шаг порождения $\gamma_i \underset{lm}{\Rightarrow} \gamma_{i+1}$ включает замену переменной A одним из тел ее продукций, скажем, β. Правило 1 построения P позволяет нам заменить A на вершине магазина цепочкой β, а правило 2 — сравнить затем любые терминалы на вершине магазина со следующими входными символами. В результате достигается МО $(q, y_{i+1}, \alpha_{i+1})$, которое представляет следующую левовыводимую цепочку γ_{i+1}.

Для завершения доказательства заметим, что $\alpha_n = \varepsilon$, так как остаток цепочки γ_n (а она представляет собой w) пуст. Таким образом, $(q, w, S) \overset{*}{\vdash} (q, \varepsilon, \varepsilon)$, т.е. P допускает w по пустому магазину.

(*Необходимость*) Нам нужно доказать нечто более общее, а именно: если P выполняет последовательность переходов, в результате которой переменная A удаляется из вершины магазина, причем магазин под ней не обрабатывается, то из A в грамматике G порождается любая цепочка, прочитанная на входе в этом процессе. Формально:

- если $(q, x, A) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$, то $A \overset{*}{\underset{G}{\Rightarrow}} x$.

Доказательство проведем индукцией по числу переходов, совершенных P.

Базис. Один переход. Единственной возможностью является то, что $A \to \varepsilon$ — продукция грамматики G, и эта продукция использована в правиле типа 1 МП-автоматом P. В этом случае $x = \varepsilon$, и $A \Rightarrow \varepsilon$.

Индукция. Предположим, что P совершает n переходов, где $n > 1$. Первый переход должен быть типа 1, где переменная A на вершине магазина заменяется одним из тел ее продукций. Причина в том, что правило типа 2 может быть использовано, когда на вершине магазина находится терминал. Пусть использована продукция $A \to Y_1Y_2\ldots Y_k$, где каждый Y_i есть либо терминал, либо переменная.

В процессе следующих $n - 1$ переходов P должен прочитать x на входе и вытолкнуть $Y_1, Y_2, \ldots, Y_k$ из магазина по очереди. Цепочку x можно разбить на подцепочки $x_1x_2\ldots x_k$,

где x_1 — порция входа, прочитанная до выталкивания Y_1 из магазина, т.е. когда длина магазина впервые уменьшается до $k-1$. Тогда x_2 является следующей порцией входа, читаемой до выталкивания Y_2, и т.д.

На рис. 6.9 представлены разбиение входа x и соответствующая обработка магазина. Предполагается, что β имеет вид BaC, поэтому x разбивается на три части $x_1x_2x_3$, где $x_2 = a$. Заметим, что вообще, если Y_i — терминал, то x_i также должен быть терминалом.

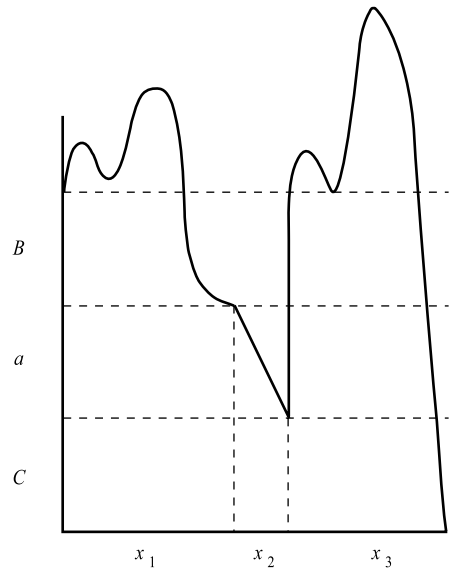


Рис. 6.9. МП-автомат A *прочитывает* x *и удаляет* BaC *из своего магазина*

Формально мы можем заключить, что $(q, x_ix_{i+1}\ldots x_k, Y_i) \overset{*}{\vdash} (q, x_{i+1}\ldots x_k, \varepsilon)$ для всех $i = 1, 2, \ldots, k$. Кроме того, длина ни одной из этих последовательностей переходов не превышает $n-1$, поэтому применимо предположение индукции в случае, если Y_i — переменная. Таким образом, можно заключить, что $Y_i \overset{*}{\Rightarrow} x_i$.

Если Y_i — терминал, то должен совершаться только один переход, в котором проверяется совпадение x_i и Y_i. Опять-таки, можно сделать вывод, что $Y_i \overset{*}{\Rightarrow} x_i$; на этот раз порождение пустое. Теперь у нас есть порождение

$$A \Rightarrow Y_1Y_2\ldots Y_k \overset{*}{\Rightarrow} x_1Y_2\ldots Y_k \overset{*}{\Rightarrow} \ldots \overset{*}{\Rightarrow} x_1x_2\ldots x_k.$$

Таким образом, $A \overset{*}{\Rightarrow} x$.

Для завершения доказательства положим $A = S$ и $x = w$. Поскольку нам дано, что $w \in N(P)$, то $(q, w, S) \overset{*}{\vdash} (q, \varepsilon, \varepsilon)$. По доказанному индукцией имеем $S \overset{*}{\Rightarrow} w$, т.е. $w \in L(G)$. □

6.3.2. От МП-автоматов к грамматикам

Завершим доказательство эквивалентности, показав, что для любого МП-автомата P найдется КС-грамматика G, язык которой совпадает с языком, допускаемым P по пустому магазину. Идея доказательства основана на том, что выталкивание одного символа из магазина вместе с прочтением некоторого входа является основным событием в процессе работы МП-автомата. При выталкивании из магазина МП-автомат может изменять свое состояние, поэтому нам следует учитывать состояние, достигаемое автоматом, когда он полностью освобождает свой магазин.

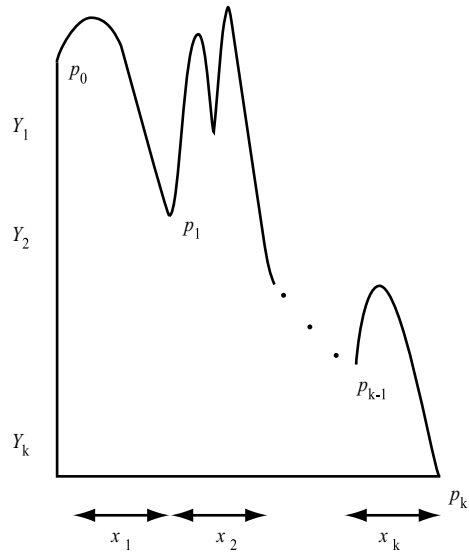


Рис. 6.10. МП-автомат совершает последовательность переходов, в результате которых символы по одному удаляются из магазина

На рис. 6.10 показано, как последовательность символов Y_1, Y_2, …, Y_k удаляется из магазина. До удаления Y_1 прочитывается вход x_1. Подчеркнем, что это "удаление" является окончательным результатом, возможно, многих переходов. Например, первый переход может изменить Y_1 на некоторый другой символ Z, следующий — изменить Z на UV, дальнейшие переходы — вытолкнуть U, а затем V. Окончательный результат заключается в том, что Y_1 изменен на ничто, т.е. вытолкнут, и все прочитанные к этому моменту входные символы образуют x_1.

На рис. 6.10 показано также окончательное изменение состояния. Предполагается, что МП-автомат начинает работу в состоянии p_0 с Y_1 на вершине магазина. После всех переходов, результат которых состоит в удалении Y_1, МП-автомат находится в состоянии p_1. Затем он достигает окончательного удаления Y_2, прочитывая при этом x_2 и приходя, возможно, за много переходов, в состояние p_2. Вычисление продолжается до тех пор, пока каждый из магазинных символов не будет удален.

Наша конструкция эквивалентной грамматики использует переменные, каждая из которых представляет “событие”, состоящее в следующем.

1. Окончательное удаление некоторого символа X из магазина.
2. Изменение состояния от некоторого p (вначале) до q, когда X окончательно заменяется ε в магазине.

Такую переменную обозначим составным символом $[pXq]$. Заметим, что эта последовательность букв является описанием *одной* переменной, а не пятью символами грамматики. Формальное построение дается следующей теоремой.

Теорема 6.14. Пусть $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ — МП-автомат. Тогда существует КС-грамматика G, для которой $L(G) = N(P)$.

Доказательство. Построим $G = (V, \Sigma, R, S)$, где V состоит из следующих переменных.

1. Специальный стартовый символ S.
2. Все символы вида $[pXq]$, где p и q — состояния из Q, а X — магазинный символ из Γ.

   Грамматика G имеет следующие продукции:

   а) продукции $S \to [q_0 Z_0 p]$ для всех состояний p. Интуитивно символ вида $[q_0 Z_0 p]$ предназначен для порождения всех тех цепочек w, которые приводят P к выталкиванию Z_0 из магазина в процессе перехода из состояния q_0 в состояние p. Таким образом, $(q, w, Z_0) \overset{*}{\vdash} (q, \varepsilon, \varepsilon)$. Эти продукции гласят, что стартовый символ S порождает все цепочки w, приводящие P к опустошению магазина после старта в начальной конфигурации;

   б) пусть $\delta(q, a, X)$ содержит пару $(r, Y_1 Y_2 \ldots Y_k)$, где a есть либо символ из Σ, либо ε, а k — некоторое неотрицательное число; при $k = 0$ пара имеет вид (r, ε). Тогда для всех списков состояний $r_1, r_2, \ldots, r_k$ в грамматике G есть продукция $[qXr_k] \to a[rY_1r_1][r_1Y_2r_2]\ldots[r_{k-1}Y_kr_k]$.

   Она гласит, что один из путей выталкивания X и перехода из состояния q в состояние r_k заключается в том, чтобы прочитать a (оно может быть равно ε), затем использовать некоторый вход для выталкивания Y_1 из магазина с переходом из состояния r в состояние r_1, далее прочитать некоторый вход, вытолкнуть Y_2 и перейти из r_1 в r_2, и т.д.

Докажем корректность неформальной интерпретации переменных вида $[qXp]$:

- $[qXp] \overset{*}{\Rightarrow} w$ тогда и только тогда, когда $(q, w, X) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$.

(*Достаточность*) Пусть $(q, w, X) \overset{*}{\underset{P}{\vdash}} (p, \varepsilon, \varepsilon)$. Докажем, что $[qXp] \overset{*}{\Rightarrow} w$, используя индукцию по числу переходов МП-автомата.

Базис. Один шаг. Пара (p, ε) должна быть в $\delta(q, w, X)$, и w есть либо одиночный символ, либо ε. Из построения G следует, что $[qXp] \to w$ является продукцией, поэтому $[qXp] \Rightarrow w$.

Индукция. Предположим, что последовательность $(q, w, X) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$ состоит из n переходов, и $n > 1$. Первый переход должен иметь вид

$$(q, w, X) \vdash (r_0, X, Y_1Y_2\ldots Y_k) \overset{*}{\vdash} (p, \varepsilon, \varepsilon),$$

где $w = aX$ для некоторого a, которое является либо символом из Σ, либо ε. Отсюда следует, что пара $(r_0, Y_1Y_2\ldots Y_k)$ должна быть в $\delta(q, a, X)$. Кроме того, по построению G существует продукция $[qXr_k] \to a[r_0Y_1r_1][r_1Y_2r_2]\ldots[r_{k-1}Y_kr_k]$, у которой $r_k = p$ и $r_1, r_2, \ldots, r_{k-1}$ — некоторые состояния из Q.

На рис. 6.10 показано, что символы $Y_1, Y_2, \ldots, Y_k$ удаляются из магазина по очереди, и для $i = 1, 2, \ldots, k-1$ можно выбрать состояние p_i, в котором находится МП-автомат при удалении Y_i. Пусть $X = w_1w_2\ldots w_k$, где w_i — входная цепочка, которая прочитывается до удаления Y_i из магазина. Тогда $(r_{i-1}, w_i, Y_i) \overset{*}{\vdash} (r_i, \varepsilon, \varepsilon)$.

Поскольку ни одна из указанных последовательностей переходов не содержит более, чем n переходов, к ним можно применить предположение индукции. Приходим к выводу, что $[r_{i-1}Y_ir_i] \overset{*}{\Rightarrow} w_i$. Соберем эти порождения вместе.

$$[qXr_k] \Rightarrow a[r_0Y_1r_1][r_1Y_2r_2]\ldots[r_{k-1}Y_kr_k] \overset{*}{\Rightarrow}$$
$$aw_1[r_1Y_2r_2]\ldots[r_{k-1}Y_kr_k] \overset{*}{\Rightarrow}$$
$$aw_1w_2[r_2Y_3r_3]\ldots[r_{k-1}Y_kr_k] \overset{*}{\Rightarrow}$$
$$\ldots$$
$$aw_1w_2\ldots w_k = w$$

Здесь $r_k = p$.

(*Необходимость*) Доказательство проводится индукцией по числу шагов в порождении.

Базис. Один шаг. Тогда $[qXp] \to w$ должно быть продукцией. Единственная возможность для существования этой продукции — если в P есть переход, в котором X выталкивается, а состояние q меняется на p. Таким образом, пара (p, ε) должна быть в $\delta(q, a, X)$, и $a = w$. Но тогда $(q, w, X) \vdash (p, \varepsilon, \varepsilon)$.

Индукция. Предположим, что $[qXp] \overset{*}{\Rightarrow} w$ за n шагов, где $n > 1$. Рассмотрим подробно первую выводимую цепочку, которая должна выглядеть следующим образом.

$$[qXr_k] \Rightarrow a[r_0Y_1r_1][r_1Y_2r_2]\ldots[r_{k-1}Y_kr_k] \overset{*}{\Rightarrow} w$$

Здесь $r_k = p$. Эта продукция должна быть в грамматике, так как $(r_0, Y_1Y_2\ldots Y_k)$ есть в $\delta(q, a, X)$.

Цепочку w можно разбить на $w = aw_1w_2\ldots w_k$ так, что $[r_{i-1}Y_ir_i] \overset{*}{\Rightarrow} w_i$ для всех $i = 1, 2, \ldots, k-1$. По предположению индукции для всех этих i верно следующее утверждение.

$$(r_{i-1}, w_i, Y_i) \overset{*}{\vdash} (r_i, \varepsilon, \varepsilon)$$

Используя теорему 6.5 для того, чтобы поместить нужные цепочки вокруг w_i на входе и под Y_i в магазине, получим

$$(r_{i-1}, w_iw_{i+1}\ldots w_k, Y_iY_{i+1}\ldots Y_k) \overset{*}{\vdash} (r_i, w_{i+1}\ldots w_k, Y_{i+1}\ldots Y_k).$$

Соберем все эти последовательности вместе и получим следующее порождение.

$$(q, aw_1w_2\ldots w_k, X) \vdash (r_0, w_1w_2\ldots w_k, Y_1Y_2\ldots Y_k) \overset{*}{\vdash}$$
$$(r_1, w_2w_3\ldots w_k, Y_2Y_3\ldots Y_k) \overset{*}{\vdash} (r_2, w_3\ldots w_k, Y_3\ldots Y_k) \overset{*}{\vdash} \ldots \overset{*}{\vdash} (r_k, \varepsilon, \varepsilon)$$

Поскольку $r_k = p$, мы доказали, что $(q, w, X) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$.

Завершим доказательство. $S \overset{*}{\Rightarrow} w$ тогда и только тогда, когда $[q_0Zp_0] \overset{*}{\Rightarrow} w$ для некоторого p в соответствии с построенными правилами для стартового символа S. Выше уже доказано, что $[q_0Zp_0] \overset{*}{\Rightarrow} w$ тогда и только тогда, когда $(q, w, Z_0) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$, т.е. когда P допускает w по пустому магазину. Таким образом, $L(G) = N(P)$. □

Пример 6.15. Преобразуем в грамматику МП-автомат $P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$ из примера 6.10. Напомним, что P_N допускает все цепочки, которые нарушают правило, что каждое e (else) должно соответствовать некоторому предшествующему i (if). Так как P_N имеет только одно состояние и один магазинный символ, грамматика строится просто. В ней есть лишь следующие две переменные:

а) S — стартовый символ, который есть в каждой грамматике, построенной методом теоремы 6.14;

б) $[qZq]$ — единственная тройка, которую можно собрать из состояний и магазинных символов автомата P_N.

Грамматика G имеет следующие продукции.

1. Единственной продукцией для S является $S \to [qZq]$. Но если бы у МП-автомата было n состояний, то было бы и n продукций такого вида, поскольку последним может быть любое из n состояний. Первое состояние должно быть начальным, а магазинный символ — стартовым, как в нашей продукции выше.
2. Из того факта, что $\delta_N(q, i, Z)$ содержит (q, ZZ), получаем продукцию $[qZq] \to i[qZq][qZq]$. В этом простом примере есть только одна такая продукция. Но если бы у автомата было n состояний, то одно такое правило порождало бы n^2 продукций, поскольку как промежуточным состоянием в теле, так и последним состоянием в голове и теле могли быть любые два состояния. Таким образом, если бы r и p

были двумя произвольными состояниями, то создавалась бы продукция $[qZp] \to i[qZr][rZp]$.

3. Из того, что $\delta_N(q, e, Z)$ содержит (q, ε), получаем продукцию $[qZq] \to e$. Заметим, что в этом случае список магазинных символов, которыми заменяется Z, пуст, поэтому единственным символом в теле является входной символ, приводящий к этому переходу.

Можно для удобства заменить тройку $[qZq]$ каким-либо простым символом, например A. В таком случае грамматика состоит из следующих продукций.

$$S \to A$$
$$A \to iAA \mid e$$

В действительности можно заметить, что S и A порождают одни и те же цепочки, поэтому их можно обозначить одинаково и записать грамматику в окончательном виде.

$$G = (\{S\}, \{i, e\}, \{S \to iSS \mid e\}, S)$$

□

6.3.3. Упражнения к разделу 6.3

6.3.1. (*) Преобразуйте грамматику

$$S \to 0S1 \mid A$$
$$A \to 1A0 \mid S \mid \varepsilon$$

в МП-автомат, допускающий тот же язык по пустому магазину.

6.3.2. Преобразуйте грамматику

$$S \to aAA$$
$$A \to aS \mid bS \mid a$$

в МП-автомат, допускающий тот же язык по пустому магазину.

6.3.3. (*) Преобразуйте МП-автомат $P = (\{p, q\}, \{0, 1\}, \{X, Z_0\}, \delta, q, Z_0)$ в КС-грамматику, где δ задана следующим образом.

1. $\delta(q, 1, Z_0) = \{(q, XZ_0)\}$.
2. $\delta(q, 1, X) = \{(q, XX)\}$.
3. $\delta(q, 0, X) = \{(p, X)\}$.
4. $\delta(q, \varepsilon, X) = \{(q, \varepsilon)\}$.
5. $\delta(p, 1, X) = \{(p, \varepsilon)\}$.
6. $\delta(p, 0, Z_0) = \{(q, Z_0)\}$.

6.3.4. Преобразуйте МП-автомат из упражнения 6.1.1 в КС-грамматику.

6.3.5. Ниже приведены КС-языки. Постройте для каждого из них МП-автомат, допускающий этот язык по пустому магазину. При желании можно сначала построить КС-грамматику для этого языка, а затем преобразовать ее в МП-автомат.

а) $\{a^n b^m c^{2(n+m)} \mid n \geq 0, m \geq 0\}$;

б) $\{a^i b^j c^k \mid i = 2j$ или $j = 2k\}$;

в) $\{0^n 1^m \mid n \leq m \leq 2n\}$.

6.3.6. (***!**) Докажите, что если P — МП-автомат, то существует МП-автомат P_1 с одним состоянием, для которого $N(P_1) = N(P)$.

6.3.7. (**!**) Пусть у нас есть МП-автомат с s состояниями, t магазинными символами, в правилах которого длина цепочки, замещающей символ в магазине, не превышает u. Дайте как можно более точную верхнюю оценку числа переменных КС-грамматики, которая строится по этому МП-автомату с помощью метода из раздела 6.3.2.

6.4. Детерминированные автоматы с магазинной памятью

Хотя МП-автоматы по определению недетерминированы, их детерминированный случай чрезвычайно важен. В частности, синтаксические анализаторы в целом ведут себя как детерминированные МП-автоматы, поэтому класс языков, допускаемых этими автоматами, углубляет понимание конструкций, пригодных для языков программирования. В этом разделе определяются детерминированные МП-автоматы и частично исследуются работы, которые им под силу и на которые они не способны.

6.4.1. Определение детерминированного МП-автомата

Интуитивно МП-автомат является детерминированным, если в любой ситуации у него нет возможности выборов перехода. Эти выборы имеют два вида. Если $\delta(q, a, X)$ содержит более одной пары, то МП-автомат безусловно не является детерминированным, поскольку можно выбирать из этих двух пар. Однако если $\delta(q, a, X)$ всегда одноэлементно, все равно остается возможность выбора между чтением входного символа и совершением ε-перехода. Таким образом, МП-автомат $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ определяется как *детерминированный* (ДМП-автомат), если выполнены следующие условия.

1. $\delta(q, a, X)$ имеет не более одного элемента для каждого q из Q, a из Σ или $a = \varepsilon$ и X из Γ.
2. Если $\delta(q, a, X)$ непусто для некоторого a из Σ, то $\delta(q, \varepsilon, X)$ должно быть пустым.

Пример 6.16. Оказывается, КС-язык L_{wwr} из примера 6.2 не имеет ДМП-автомата. Однако путем помещения "центрального маркера" c в середину слов получается язык $L_{wcwr} = \{wcw^R \mid w \in (\mathbf{0} + \mathbf{1})^*\}$, распознаваемый некоторым ДМП-автоматом.

Стратегией этого ДМП-автомата является заталкивание символов 0 и 1 в магазин до появления на входе маркера c. Затем автомат переходит в другое состояние, в котором сравнивает входные и магазинные символы и выталкивает магазинные в случае их сов-

падения. Находя несовпадение, он останавливается без допускания (“умирает”); его вход не может иметь вид wcw^R. Если путем удаления магазинных символов он достигает стартового символа, отмечающего дно магазина, то он допускает свой вход.

По своей идее этот автомат очень похож на МП-автомат, изображенный на рис. 6.2. Однако тот МП-автомат был недетерминированным, поскольку в состоянии q_0 всегда имел возможность выбора между заталкиванием очередного входного символа в магазин и переходом в состояние q_1 без чтения входа, т.е. он должен был угадывать, достигнута ли середина. ДМП-автомат для L_{wcwr} изображен в виде диаграммы переходов на рис. 6.11.

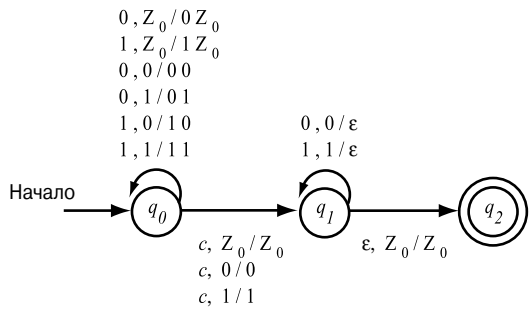


Рис. 6.11. ДМП-автомат, допускающий L_{wcwr}

Очевидно, этот МП-автомат детерминирован. У него никогда нет выбора перехода в одном и том же состоянии и при использовании одних и тех же входных и магазинных символов. Что же касается выбора между использованием входного символа или ε, то единственным ε-переходом, который он совершает, является переход из q_1 в q_2 с Z_0 на вершине магазина. Однако в состоянии q_1 с Z_0 на вершине других переходов нет. □

6.4.2. Регулярные языки и детерминированные МП-автоматы

ДМП-автоматы допускают класс языков, который находится между регулярными и КС-языками. Вначале докажем, что языки ДМП-автоматов включают в себя все регулярные.

Теорема 6.17. Если L — регулярный язык, то $L = L(P)$ для некоторого ДМП-автомата P.

Доказательство. По существу, ДМП-автомат может имитировать детерминированный конечный автомат. МП-автомат содержит некоторый символ Z_0 в магазине, так как он должен иметь магазин, но в действительности он игнорирует магазин, используя только состояние. Формально, пусть $A = (Q, \Sigma, \delta_A, q_0, F)$ — конечный автомат. Построим $P = (Q, \Sigma, \{Z_0\}, \delta_P, q_0, Z_0, F)$, определив $\delta_P(q, a, Z_0) = \{(p, Z_0)\}$ для всех состояний p и q из Q, при которых $\delta_A(q, a) = p$.

Мы утверждаем, что $(q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (p, \varepsilon, Z_0)$ тогда и только тогда, когда $\hat{\delta}(q_0, w) = p$, т.е. P имитирует A, используя его состояние. Доказательства в обе стороны просты и

проводятся индукцией по $|w|$, поэтому оставляются для завершения читателю. Поскольку как A, так и P допускают, достигнув какого-либо состояния из F, приходим к выводу, что их языки равны. □

Если мы хотим, чтобы ДМП-автомат допускал по пустому магазину, то обнаруживаем, что возможности по распознаванию языков существенно ограничены. Говорят, что язык L имеет *префиксное свойство*, или *свойство префиксности*, если в L нет двух различных цепочек x и y, где x является префиксом y.

Пример 6.18. Язык L_{wcwr} из примера 6.16 имеет префиксное свойство, т.е. в нем не может быть двух разных цепочек wcw^R и xcx^R, одна из которых является префиксом другой. Чтобы убедиться в этом, предположим, что wcw^R — префикс xcx^R, и $w \neq x$. Тогда w должна быть короче, чем x. Следовательно, c в w приходится на позицию, в которой x имеет 0 или 1, а это противоречит предположению, что wcw^R — префикс xcx^R.

С другой стороны, есть очень простые языки, не имеющие префиксного свойства. Рассмотрим $\{0\}^*$, т.е. множество всех цепочек из символов 0. Очевидно, в этом языке есть пары цепочек, одна из которых является префиксом другой, так что этот язык не обладает префиксным свойством. В действительности, из *любых* двух цепочек одна является префиксом другой, хотя это условие и сильнее, чем то, которое нужно для отрицания префиксного свойства. □

Заметим, что язык $\{0\}^*$ регулярен. Таким образом, неверно, что каждый регулярный язык есть $N(P)$ для некоторого ДМП-автомата P. Оставляем в качестве упражнения следующее утверждение.

Теорема 6.19. Язык L есть $N(P)$ для некоторого ДМП-автомата P тогда и только тогда, когда L имеет префиксное свойство и L есть $L(P')$ для некоторого ДМП-автомата P'. □

6.4.3. Детерминированные МП-автоматы и КС-языки

Мы уже видели, что ДМП-автоматы могут допускать языки вроде L_{wcwr}, которые не являются регулярными. Для того чтобы убедиться в его нерегулярности, предположим, что это не так, и используем лемму о накачке. Пусть n — константа из леммы. Рассмотрим цепочку $w = 0^n c 0^n$ из L_{wcwr}. Если ее "накачивать", изменяется длина первой группы символов 0, и получаются цепочки из L_{wcwr}, у которых "центральный маркер" расположен не в центре. Так как эти цепочки не принадлежат L_{wcwr}, получаем противоречие и делаем вывод, что L_{wcwr} нерегулярен.

С другой стороны, существуют КС-языки, вроде L_{wwr}, которые не могут допускаться по заключительному состоянию никаким ДМП-автоматом. Формальное доказательство весьма сложно, но интуитивно прозрачно. Если P — ДМП-автомат, допускающий L_{wwr}, то при чтении последовательности символов 0 он должен записать их в магазин или сделать что-нибудь равносильное для подсчета их количества. Например, записывать один X для каждых 00 на входе и использовать состояние для запоминания четности или нечетности числа символов 0.

Предположим, P прочитал n символов 0 и затем видит на входе 110^n. Он должен проверить, что после 11 находятся n символов 0, и для этого он должен опустошить свой магазин.[5] Теперь он прочитал 0^n110^n. Если далее он видит идентичную цепочку, он должен допускать, поскольку весь вход имеет вид ww^R, где $w = 0^n110^n$. Однако если P видит 0^m110^m, где $m \neq n$, он должен *не допускать*. Поскольку его магазин пуст, он не может запомнить, каким было произвольное целое n, и не способен допустить L_{wwr}. Подведем итог.

- Языки, допускаемые ДМП-автоматами по заключительному состоянию, включают регулярные языки как собственное подмножество, но сами образуют собственное подмножество КС-языков.

6.4.4. Детерминированные МП-автоматы и неоднозначные грамматики

Мощность ДМП-автоматов можно уточнить, заметив, что все языки, допускаемые ими, имеют однозначные грамматики. К сожалению, класс языков ДМП-автоматов не совпадает с подмножеством КС-языков, не являющихся существенно неоднозначными. Например, L_{wwr} имеет однозначную грамматику $S \rightarrow 0S0 \mid 1S1 \mid \varepsilon$, хотя и не является ДМП-автоматным языком. Следующая теорема уточняет заключительное утверждение из подраздела 6.4.3.

Теорема 6.20. Если $L = N(P)$ для некоторого ДМП-автомата P, то L имеет однозначную КС-грамматику.

Доказательство. Утверждаем, что конструкция теоремы 6.14 порождает однозначную КС-грамматику G, когда МП-автомат, к которому она применяется, детерминирован. Вначале вспомним (см. теорему 5.29), что для однозначности грамматики G достаточно показать, что она имеет уникальные левые порождения.

Предположим, P допускает w по пустому магазину. Тогда он делает это с помощью одной-единственной последовательности переходов, поскольку он детерминирован и не может работать после опустошения магазина. Зная эту последовательность переходов, мы можем однозначно определить выбор каждой продукции в левом порождении w в G. Правило автомата P, на основании которого применяется продукция, всегда одно. Но правило, скажем, $\delta(q, a, X) = \{(r, Y_1Y_2\ldots Y_k)\}$, может порождать много продукций грамматики G, с различными состояниями в позициях, отражающих состояния P после удаления каждого из Y_1, Y_2, …, Y_k. Однако, поскольку P детерминирован, осуществляется только одна из этих последовательностей переходов, поэтому только одна из этих продукций в действительности ведет к порождению w. □

Мы можем, однако, доказать больше: даже языки, допускаемые ДМП-автоматами по заключительному состоянию, имеют однозначные грамматики. Поскольку мы знаем

[5] Это интуитивное утверждение требует сложного формального доказательства; возможен ли другой способ для P сравнить два равных блока символов 0?

лишь, как строятся грамматики, исходя из МП-автоматов, допускающих по пустому магазину, нам придется изменять язык, чтобы он обладал префиксным свойством, а затем модифицировать грамматику, чтобы она порождала исходный язык. Обеспечим это с помощью "концевого маркера".

Теорема 6.21. Если $L = L(P)$ для некоторого ДМП-автомата P, то L имеет однозначную КС-грамматику.

Доказательство. Пусть \$ будет "концевым маркером", отсутствующим в цепочках языка L, и пусть $L' = L\$$. Таким образом, цепочки языка L' представляют собой цепочки из L, к которым дописан символ \$. Тогда L' имеет префиксное свойство, и по теореме 6.19 $L' = N(P')$ для некоторого ДМП-автомата P'.[6] По теореме 6.20 существует однозначная грамматика G', порождающая язык $N(P')$, т.е. L'.

Теперь по грамматике G' построим G, для которой $L(G) = L$. Для этого нужно лишь избавиться от маркера \$ в цепочках. Будем рассматривать \$ как переменную грамматики G и введем продукцию $\$ \to \varepsilon$; остальные продукции G и G' одинаковы. Поскольку $L(G') = L'$, получаем, что $L(G) = L$.

Утверждаем, что G однозначна. Действительно, левые порождения в G совпадают с левыми порождениями в G', за исключением последнего шага в G — изменения \$ на ε. Таким образом, если бы терминальная цепочка w имела два левых порождения в G, то $w\$$ имела бы два порождения в G'. Поскольку G' однозначна, G также однозначна. □

6.4.5. Упражнения к разделу 6.4

6.4.1. Для каждого из следующих МП-автоматов укажите, является ли он детерминированным. Либо докажите, что он соответствует определению ДМП-автомата, либо укажите правила, нарушающие его:

а) МП-автомат из примера 6.2;

б) (*) МП-автомат из упражнения 6.1.1;

в) МП-автомат из упражнения 6.3.3.

6.4.2. Постройте ДМП-автомат для каждого из следующих языков:

а) $\{0^n1^m \mid n \le m\}$;

б) $\{0^n1^m \mid n \ge m\}$;

в) $\{0^n1^m0^n \mid n$ и m произвольны$\}$.

---

[6] Доказательство теоремы 6.19 возникает в упражнении 6.4.3, но построение P' по P несложно. Добавим новое состояние q, в которое переходит P', если P перешел в заключительное состояние и следующим входным символом является \$. Находясь в состоянии q, P' выталкивает все символы из магазина. Кроме того, для P' нужен дополнительный маркер дна магазина, чтобы избежать случайного опустошения магазина при имитации P.

6.4.3. Теорему 6.19 можно доказать с помощью следующих трех шагов:

а) докажите, что если $L = N(P)$ для некоторого ДМП-автомата P, то L обладает префиксным свойством;

б) (!) докажите, что если $L = N(P)$ для некоторого ДМП-автомата P, то существует ДМП-автомат P', для которого $L = L(P')$;

в) (*!) докажите, что если L обладает префиксным свойством и $L = L(P')$ для некоторого ДМП-автомата P', то существует ДМП-автомат P, для которого $L = N(P)$.

6.4.4. (!!) Докажите, что язык

$L = \{0^n1^n \mid n \geq 1\} \bigcup \{0^n1^{2n} \mid n \geq 1\}$

является КС-языком, не допускаемым ни одним ДМП-автоматом. *Указание*. Докажите, что должны существовать две цепочки вида 0^n1^n для различных значений n, скажем, n_1 и n_2, чтение которых приводит гипотетический ДМП-автомат для языка L к одному и тому же МО. Интуитивно ДМП-автомат должен удалить из своего магазина почти все, что туда было помещено при чтении символов 0, для того, чтобы проверить, что далее читается равное количество символов 1. Таким образом, ДМП-автомат не может решить, следует ли допускать вход, следующий после чтения n_1 или n_2 символов 1.

Резюме

- *Магазинные автоматы*. МП-автомат — это недетерминированный конечный автомат с магазином, который может использоваться для запоминания цепочек произвольной длины. Магазин может читаться и изменяться только со стороны его вершины.
- *Переходы магазинных автоматов*. МП-автомат выбирает следующий переход на основе своего текущего состояния, следующего входного символа и символа на вершине магазина. Он может также выбрать переход, не зависящий от входного символа и без его чтения. Будучи недетерминированным, МП-автомат может иметь некоторое конечное число выборов перехода; каждый из них состоит из нового состояния и цепочки магазинных символов, заменяющей символ на вершине магазина.
- *Допускание магазинными автоматами*. Существуют два способа, которыми МП-автомат может сигнализировать допуск. Один заключается в достижении заключительного состояния, другой — в опустошении магазина. Эти способы эквивалентны в том смысле, что любой язык, допускаемый по одному способу, допускается и по другому (некоторым другим МП-автоматом).

- *Мгновенные описания* (*МО*), или *конфигурации*. Для описания "текущего положения" МП-автомата используется МО, образуемое состоянием, оставшимся входом и содержимым магазина. Отношение переходов $\vdash$ между МО представляет отдельные переходы МП-автомата.
- *Магазинные автоматы и грамматики*. Класс языков, допускаемых МП-автоматами как по заключительному состоянию, так и по пустому магазину, совпадает с классом КС-языков.
- *Детерминированные магазинные автоматы*. МП-автомат является детерминированным, если у него никогда нет выбора перехода для данных состояния, входного символа (включая ε) и магазинного символа. У него также нет выбора между переходом с использованием входного символа и переходом без его использования.
- *Допускание детерминированными магазинными автоматами*. Два способа допускания — по заключительному состоянию и по пустому магазину — неэквивалентны для детерминированных МП-автоматов. Точнее, языки, допускаемые по пустому магазину, — это те, и только те, языки, которые допускаются по заключительному состоянию и обладают префиксным свойством (ни одна цепочка языка не является префиксом другой).
- *Языки, допускаемые ДМП-автоматами*. Все регулярные языки допускаются (по заключительному состоянию) ДМП-автоматами, и существуют нерегулярные языки, допускаемые ДМП-автоматами. Языки ДМП-автоматов являются контекстно-свободными, причем для них существуют однозначные грамматики. Таким образом, языки ДМП-автоматов лежат строго между регулярными и контекстно-свободными языками.

Литература

Идея магазинного автомата была высказана независимо в работах Эттингера [4] и Шютценберже [5]. Установление эквивалентности магазинных автоматов и контекстно-свободных грамматик также явилось результатом независимых исследований; оно было представлено Хомским в техническом отчете MIT за 1961 г., но впервые было опубликовано в [1].

Детерминированный МП-автомат впервые был предложен Фишером [2] и Шютценберже [5]. Позднее он приобрел большое значение как модель синтаксического анализатора. Примечательно, что Кнут предложил в [3] "LR(k)-грамматики", подкласс КС-грамматик, порождающих в точности языки ДМП-автоматов. LR(k)-грамматики, в свою очередь, образуют базис для YACC, инструмента для создания анализаторов, обсуждаемого в разделе 5.3.2.

1. J. Evey, "Application of pushdown store machines," *Proc. Fall Joint Computer Conference* (1963), AFIPS Press, Montvale, NJ, pp. 215–227.

2. P. C. Fischer, “On computabililty by certain classes of restricted Turing machines,” *Proc. Fourth Annl. Symposium on Switching Circuit Theory and Logical Design* (1963), pp. 23–32.
3. D. E. Knuth, “On the translation of languages from left to right,” *Information and Control* **8**:6 (1965), pp. 607–639. (Кнут Д. О переводе (трансляции) языков слева направо. — Сб. “Языки и автоматы”. — М.: Мир, 1975. — С. 9–42.)
4. A. G. Oettinger, “Automatic syntactic analysis and pushdown store,” *Proc. Symposia on Applied Math.* **12** (1961), American Mathematical Society, Providence, RI.
5. M. P. Schutzenberger, “On context-free languages and pushdown automata,” *Information and Control* **6**:3 (1963), pp. 246–264.

ГЛАВА 7

Свойства контекстно-свободных языков

Наше изучение контекстно-свободных языков завершается знакомством с некоторыми из их свойств. Вначале определяются ограничения на структуру продукций КС-грамматик и доказывается, что всякий КС-язык имеет грамматику специального вида. Этот факт облегчает доказательство утверждений о КС-языках.

Затем доказывается "лемма о накачке" для КС-языков. Эта теорема аналогична теореме 4.1 для регулярных языков, но может быть использована для доказательства того, что некоторые языки не являются контекстно-свободными. Далее рассматриваются свойства, изученные в главе 4 для регулярных языков, — свойства замкнутости и разрешимости. Показывается, что некоторые, но не все, свойства замкнутости регулярных языков сохраняются и у КС-языков. Часть задач, связанных с КС-языками, разрешается с помощью обобщения проверок, построенных для регулярных языков, но есть и ряд вопросов о КС-языках, на которые нельзя дать ответ.

7.1. Нормальные формы контекстно-свободных грамматик

Цель этого раздела — показать, что каждый КС-язык (без ε) порождается грамматикой, все продукции которой имеют форму $A \to BC$ или $A \to a$, где A, B и C — переменные, a — терминал. Эта форма называется *нормальной формой Хомского*. Для ее получения нужно несколько предварительных преобразований, имеющих самостоятельное значение.

1. Удалить *бесполезные символы*, т.е. переменные или терминалы, которые не встречаются в порождениях терминальных цепочек из стартового символа.
2. Удалить *ε-продукции*, т.е. продукции вида $A \to \varepsilon$ для некоторой переменной A.
3. Удалить *цепные продукции* вида $A \to B$ с переменными A и B.

7.1.1. Удаление бесполезных символов

Символ X называется *полезным* в грамматике $G = (V, T, P, S)$, если существует некоторое порождение вида $S \overset{*}{\Rightarrow} \alpha X \beta \overset{*}{\Rightarrow} w$, где $w \in T^*$. Отметим, что X может быть как переменной, так и терминалом, а выводимая цепочка $\alpha X \beta$ — первой или последней в по-

рождении. Если символ X не является полезным, то называется *бесполезным*. Очевидно, что исключение бесполезных символов из грамматики не изменяет порождаемого языка, поэтому все бесполезные символы можно обнаружить и удалить.

Наш подход к удалению бесполезных символов начинается с определения двух свойств, присущих полезным символам.

1. Символ X называется *порождающим*, если $X \overset{*}{\Rightarrow} w$ для некоторой терминальной цепочки w. Заметим, что каждый терминал является порождающим, поскольку w может быть этим терминалом, порождаемым за 0 шагов.
2. Символ X называется *достижимым*, если существует порождение $S \overset{*}{\Rightarrow} \alpha X \beta$ для некоторых α и β.

Естественно, что полезный символ является одновременно и порождающим, и достижимым. Если сначала удалить из грамматики непорождающие символы, а затем недостижимые, то, как будет доказано, останутся только полезные.

Пример 7.1. Рассмотрим следующую грамматику.

$$S \to AB \mid a$$
$$A \to b$$

Все символы, кроме B, являются порождающими; a и b порождают самих себя, S порождает a и A порождает b. Если удалить B, то придется удалить и продукцию $S \to AB$, что приводит к следующей грамматике.

$$S \to a$$
$$A \to b$$

Теперь нетрудно обнаружить, что из S достижимы только a и S. Удаление A и b оставляет лишь продукцию $S \to a$. Она образует грамматику с языком $\{a\}$, как и у исходной грамматики.

Заметим, что если начать с проверки достижимости, то все символы грамматики

$$S \to AB \mid a$$
$$A \to b$$

оказываются достижимыми. Если затем удалить B как непорождающий символ, то останется грамматика с бесполезными символами A и b. □

Теорема 7.2. Пусть $G = (V, T, P, S)$ — КС-грамматика, и $L(G) \neq \varnothing$, т.е. G порождает хотя бы одну цепочку. Пусть $G_1 = (V_1, T_1, P_1, S)$ — грамматика, полученная с помощью следующих двух шагов.

1. Вначале удаляются непорождающие символы и все продукции, содержащие один или несколько таких символов. Пусть $G_2 = (V_2, T_2, P_2, S)$ — полученная в результате грамматика. Заметим, что S должен быть порождающим, так как по предположению $L(G)$ содержит хотя бы одну цепочку, поэтому S не удаляется.

2. Затем удаляются все символы, недостижимые в G_2.

Тогда G_1 не имеет бесполезных символов, и $L(G_1) = L(G)$.

Доказательство. Пусть X — оставшийся символ, т.е. $X \in T_1 \cup V_1$. Нам известно, что $X \overset{*}{\underset{G}{\Rightarrow}} w$ для некоторой цепочки w из T^*. Кроме того, каждый символ, использованный в порождении w из X, также является порождающим. Таким образом, $X \overset{*}{\underset{G_2}{\Rightarrow}} w$.

Поскольку X не был удален на втором шаге, нам также известно, что существуют такие α и β, для которых $S \overset{*}{\underset{G_2}{\Rightarrow}} \alpha X \beta$. Кроме того, каждый символ, использованный в этом порождении, достижим, поэтому $S \overset{*}{\underset{G_1}{\Rightarrow}} \alpha X \beta$.

Нам известно, что каждый символ в цепочке $\alpha X \beta$ достижим, и что все эти символы принадлежат $T_2 \cup V_2$, поэтому каждый из них является порождающим в G_2. Порождение некоторой терминальной цепочки, скажем, $\alpha X \beta \overset{*}{\underset{G_2}{\Rightarrow}} xwy$, содержит только символы, достижимые из S, поскольку они достижимы из символов в цепочке $\alpha X \beta$. Таким образом, это порождение есть также порождение в G_1, т.е.

$$S \overset{*}{\underset{G_1}{\Rightarrow}} \alpha X \beta \overset{*}{\underset{G_1}{\Rightarrow}} xwy.$$

Итак, X полезен в G_1. Ввиду произвольности X в G_1 можно заключить, что G_1 не имеет бесполезных символов.

Нам осталось доказать, что $L(G_1) = L(G)$. Как обычно, покажем взаимное включение этих языков.

Включение $L(G_1) \subseteq L(G)$ очевидно, поскольку при построении G_1 из G символы и продукции только удаляются.

Докажем, что $L(G) \subseteq L(G_1)$. Если $w \in L(G)$, то $S \overset{*}{\underset{G}{\Rightarrow}} w$. Каждый символ в этом порождении, очевидно, является как достижимым, так и порождающим, поэтому порождение в G_1 также его содержит. Таким образом, $S \overset{*}{\underset{G_1}{\Rightarrow}} w$, и $w \in L(G_1)$. □

7.1.2. Вычисление порождающих и достижимых символов

Рассмотрим, каким образом вычисляются множества порождающих и достижимых символов грамматики. В алгоритме, используемом в обеих задачах, делается максимум возможного, чтобы обнаружить символы этих двух типов. Докажем, что если правильное индуктивное построение указанных множеств не позволяет обнаружить, что символ является порождающим или достижимым, то он не является символом соответствующего типа.

Базис. Каждый символ из T, очевидно, является порождающим; он порождает сам себя.

Индукция. Предположим, есть продукция $A \to \alpha$, и известно, что каждый символ в α является порождающим. Тогда A — порождающий. Заметим, что это правило включает и случай, когда $\alpha = \varepsilon$; все переменные, имеющие ε в качестве тела продукции, являются порождающими.

Пример 7.3. Рассмотрим грамматику из примера 7.1. Согласно базису a и b — порождающие. По индукции используем продукции $A \to b$ и $S \to a$, чтобы заключить, что A и S — также порождающие. На этом индукция заканчивается. Продукцию $S \to AB$ использовать нельзя, поскольку не установлено, что B — порождающий. Таким образом, множеством порождающих символов является $\{a, b, A, S\}$. □

Теорема 7.4. Вышеприведенный алгоритм находит все порождающие символы грамматики G и только их.

Доказательство. В одну сторону, а именно, что каждый добавленный символ на самом деле является порождающим, доказать легко с помощью индукции по порядку, в котором символы добавляются к множеству порождающих символов. Это оставляется в качестве упражнения.

Для доказательства в другую сторону предположим, что X — порождающий, и пусть $X \overset{*}{\underset{G}{\Rightarrow}} w$. Докажем индукцией по длине порождения, что X будет обнаружен как порождающий.

Базис. Нуль шагов. Тогда X — терминал и находится как порождающий согласно базису алгоритма.

Индукция. Если порождение имеет n шагов, где $n > 0$, то X — переменная. Пусть порождение имеет вид $X \Rightarrow \alpha \overset{*}{\underset{G}{\Rightarrow}} w$, т.е. первой использована продукция $X \to \alpha$. Из каждого символа цепочки α выводится некоторая терминальная цепочка, образующая часть w, и это порождение имеет менее, чем n шагов. По предположению индукции каждый символ цепочки α находится как порождающий. Индуктивная часть алгоритма позволяет с помощью продукции $X \to \alpha$ заключить, что X — порождающий. □

Теперь рассмотрим индуктивный алгоритм, с помощью которого находится множество достижимых символов грамматики $G = (V, T, P, S)$. Можно доказать, что если символ не добавляется к множеству достижимых символов путем максимально возможного обнаружения, то он не является достижимым.

Базис. Очевидно, что S действительно достижим.

Индукция. Пусть обнаружено, что некоторая переменная A достижима. Тогда для всех продукций с головой A все символы тел этих продукций также достижимы.

Пример 7.5. Снова начнем с грамматики из примера 7.1. Согласно базису S достижим. Поскольку S имеет тела продукций AB и a, символы A, B и a также достижимы. У B продукций нет, но A имеет $A \to b$. Делаем вывод, что b достижим. К множеству достижимых символов $\{S, A, B, a, b\}$ добавить больше нечего. □

Теорема 7.6. Вышеприведенный алгоритм находит все достижимые символы грамматики G, и только их.

Доказательство. Это доказательство представляет собой еще одну пару простых индукций в духе теоремы 7.4 и оставляется в качестве упражнения.

7.1.3. Удаление ε-продукций

Покажем теперь, что ε-продукции, хотя и удобны в задачах построения грамматик, не являются существенными. Конечно же, без продукции с телом ε невозможно породить пустую цепочку как элемент языка. Таким образом, в действительности доказывается, что если L задается КС-грамматикой, то $L - \{\varepsilon\}$ имеет КС-грамматику без ε-продукций.

Начнем с обнаружения "ε-порождающих" переменных. Переменная A называется *ε-порождающей*, если $A \stackrel{*}{\Rightarrow} \varepsilon$. Если A — ε-порождающая, то где бы в продукциях она ни встречалась, например в $B \to CAD$, из нее можно (но не обязательно) вывести ε. Продукция с такой переменной имеет еще одну версию без этой переменной в теле ($B \to CD$). Эта версия соответствует тому, что ε-порождающая переменная использована для вывода ε. Используя версию $B \to CAD$, мы не разрешаем из A выводить ε. Это не создает проблем, так как далее мы просто удалим все продукции с телом ε, предохраняя каждую переменную от порождения ε.

Пусть $G = (V, T, P, S)$ — КС-грамматика. Все ε-порождающие символы G можно найти с помощью следующего алгоритма. Далее будет показано, что других ε-порождающих символов, кроме найденных алгоритмом, нет.

Базис. Если $A \to \varepsilon$ — продукция в G, то A — ε-порождающая.

Индукция. Если в G есть продукция $B \to C_1C_2\ldots C_k$, где каждый символ C_i является ε-порождающим, то B — ε-порождающая. Отметим, что для того, чтобы C_i был ε-порождающим, он должен быть переменной, поэтому нам нужно рассматривать продукции, тела которых содержат только переменные.

Теорема 7.7. В любой грамматике ε-порождающими являются только переменные, найденные вышеприведенным алгоритмом.

Доказательство. Переформулируем утверждение в виде "A является ε-порождающей тогда и только тогда, когда алгоритм идентифицирует A как ε-порождающую". Для достаточности нетрудно показать индукцией по порядку, в котором обнаруживаются ε-порождающие символы, что каждый такой символ действительно порождает ε. Для необходимости используем индукцию по длине кратчайшего порождения $A \stackrel{*}{\Rightarrow} \varepsilon$.

Базис. Один шаг. Тогда $A \to \varepsilon$ должно быть продукцией, и A обнаруживается согласно базису алгоритма.

Индукция. Пусть $A \stackrel{*}{\Rightarrow} \varepsilon$ за n шагов, где $n > 1$. Первый шаг должен иметь вид $A \Rightarrow C_1C_2\ldots C_k \stackrel{*}{\Rightarrow} \varepsilon$, где каждый символ C_i порождает ε за число шагов, которое

меньше n. По предположению индукции каждый символ C_i обнаруживается как ε-порождающий. Тогда с помощью индуктивного шага A также обнаруживается как ε-порождающая на основе продукции $A \to C_1C_2\ldots C_k$. □

Пример 7.8. Рассмотрим следующую грамматику.

$S \to AB$

$A \to aAA \mid \varepsilon$

$B \to bBB \mid \varepsilon$

Сначала найдем ε-порождающие символы. A и B непосредственно ε-порождающие, так как имеют продукции с ε в качестве тела. Тогда и S ε-порождающий, поскольку тело продукции $S \to AB$ состоит только из ε-порождающих символов. Таким образом, все три переменные являются ε-порождающими.

Построим теперь продукции грамматики G_1. Сначала рассмотрим $S \to AB$. Все символы тела являются ε-порождающими, поэтому есть 4 способа выбрать присутствие или отсутствие A и B. Однако нам нельзя удалять все символы одновременно, поэтому получаем следующие три продукции.

$S \to AB \mid A \mid B$

Далее рассмотрим продукцию $A \to aAA$. Вторую и третью позиции занимают ε-порождающие символы, поэтому снова есть 4 варианта их присутствия или отсутствия. Все они допустимы, поскольку в любом из них остается терминал a. Два из них совпадают, поэтому в грамматике G_1 будут следующие три продукции.

$A \to aAA \mid aA \mid a$

Аналогично, продукция для B приводит к следующим продукциям в G_1.

$B \to bBB \mid bB \mid b$

Обе ε-продукции из G не вносят в G_1 ничего. Таким образом, следующие продукции образуют G_1.

$S \to AB \mid A \mid B$

$A \to aAA \mid aA \mid a$

$B \to bBB \mid bB \mid b$

□

Завершим наше изучение удаления ε-продукций доказательством, что описанная конструкция не изменяет язык, за исключением того, что цепочки ε в нем больше нет, если она была в языке грамматики G. Поскольку конструкция, очевидно, удаляет ε-продукции, мы будем иметь полное доказательство утверждения о том, что для любой КС-грамматики G найдется такая КС-грамматика G_1 без ε-продукций, для которой

$L(G_1) = L(G) - \{\varepsilon\}$.

Теорема 7.9. Если грамматика G_1 построена по грамматике G с помощью описанной выше конструкции удаления ε-продукций, то $L(G_1) = L(G) - \{\varepsilon\}$.

Доказательство. Нужно доказать, что если $w \neq \varepsilon$, то $w \in L(G_1)$ тогда и только тогда, когда $w \in L(G)$. Как часто случается, проще доказать более общее утверждение. В данном случае будем говорить о терминальных цепочках, порождаемых каждой переменной, несмотря на то, что нас интересуют лишь порождаемые стартовым символом S. Таким образом, докажем следующее утверждение.

- $A \underset{G_1}{\overset{*}{\Rightarrow}} w$ тогда и только тогда, когда $A \underset{G}{\overset{*}{\Rightarrow}} w$ и $w \neq \varepsilon$.

В обе стороны доказательство проводится индукцией по длине порождения.

(*Необходимость*) Пусть $A \underset{G_1}{\overset{*}{\Rightarrow}} w$. Несомненно, $w \neq \varepsilon$, поскольку G_1 не имеет ε-продукций. Докажем индукцией по длине порождения, что $A \underset{G}{\overset{*}{\Rightarrow}} w$.

Базис. Один шаг. В этом случае в G_1 есть продукция $A \to w$. Согласно конструкции G_1 в G есть продукция $A \to \alpha$, причем α — это w, символы которой, возможно, перемежаются ε-порождающими переменными. Тогда в G есть порождение $A \underset{G}{\Rightarrow} \alpha \underset{G}{\overset{*}{\Rightarrow}} w$, где на шагах после первого, если они есть, из всех переменных в цепочке α выводится ε.

Индукция. Пусть в порождении n шагов, $n > 1$. Тогда оно имеет вид $A \underset{G_1}{\Rightarrow} X_1X_2\ldots X_k \underset{G_1}{\overset{*}{\Rightarrow}} w$. Первая использованная продукция должна быть построена по продукции $A \to Y_1Y_2\ldots Y_m$, где цепочка $Y_1Y_2\ldots Y_m$ совпадает с цепочкой $X_1X_2\ldots X_k$, символы которой, возможно, перемежаются дополнительными ε-порождающими переменными. Цепочку w можно разбить на $w_1w_2\ldots w_k$, где $X_i \underset{G_1}{\overset{*}{\Rightarrow}} w_i$ для $i = 1, 2, \ldots, k$. Если X_i есть терминал, то $w_i = X_i$, а если переменная, то порождение $X_i \underset{G_1}{\overset{*}{\Rightarrow}} w_i$ содержит менее n шагов. По предположению индукции $X_i \underset{G}{\overset{*}{\Rightarrow}} w_i$.

Теперь построим соответствующее порождение в G.

$$A \underset{G}{\Rightarrow} Y_1Y_2\ldots Y_m \underset{G}{\overset{*}{\Rightarrow}} X_1X_2\ldots X_k \underset{G}{\overset{*}{\Rightarrow}} w_1w_2\ldots w_k = w$$

На первом шаге применяется продукция $A \to Y_1Y_2\ldots Y_m$, которая, как мы знаем, есть в G. Следующая группа шагов представляет порождение ε из тех Y_j, которые не являются ни одним из X_i. Последняя группа шагов представляет порождения w_i из X_i, которые существуют по предположению индукции.

(*Достаточность*) Пусть $A \underset{G}{\overset{*}{\Rightarrow}} w$ и $w \neq \varepsilon$. Докажем индукцией по длине n порождения, что $A \underset{G_1}{\overset{*}{\Rightarrow}} w$.

Базис. Один шаг. $A \to w$ является продукцией в G. Поскольку $w \neq \varepsilon$, эта же продукция есть и в G_1, поэтому $A \underset{G_1}{\overset{*}{\Rightarrow}} w$.

Индукция. Пусть в порождении n шагов, $n > 1$. Тогда оно имеет вид $A \underset{G}{\Rightarrow} Y_1Y_2\ldots Y_m \underset{G}{\overset{*}{\Rightarrow}} w$. Цепочку w можно разбить на $w_1w_2\ldots w_k$ так, что $Y_i \underset{G}{\overset{*}{\Rightarrow}} w_i$ для $i = 1, 2, \ldots, m$. Пусть $X_1, X_2, \ldots, X_k$ будут теми из Y_j (в порядке записи), для которых $w_i \neq \varepsilon$. $k \geq 1$, поскольку $w \neq \varepsilon$. Таким образом, $A \to X_1X_2\ldots X_k$ является продукцией в G_1.

Утверждаем, что $X_1X_2\ldots X_k \underset{G}{\overset{*}{\Rightarrow}} w$, поскольку только Y_j, которых нет среди $X_1, X_2, \ldots, X_k$, использованы для порождения ε и не вносят ничего в порождение w. Так как каждое из порождений $Y_j \underset{G}{\overset{*}{\Rightarrow}} w_j$ содержит менее n шагов, к ним можно применить предположение индукции и заключить, что если $w_j \neq \varepsilon$, то $Y_j \underset{G_1}{\overset{*}{\Rightarrow}} w_j$. Таким образом, $A \underset{G_1}{\Rightarrow} X_1X_2\ldots X_k \underset{G_1}{\overset{*}{\Rightarrow}} w$.

Завершим доказательство. Нам известно, что $w \in L(G_1)$ тогда и только тогда, когда $S \underset{G_1}{\overset{*}{\Rightarrow}} w$. Полагая, что $A = S$ в описанных выше рассуждениях, получаем, что $w \in L(G_1)$ тогда и только тогда, когда $S \underset{G}{\overset{*}{\Rightarrow}} w$ и $w \neq \varepsilon$. Таким образом, $w \in L(G_1)$ тогда и только тогда, когда $w \in L(G_1)$ и $w \neq \varepsilon$. □

7.1.4. Удаление цепных продукций

Цепная продукция — это продукция вида $A \to B$, где и A, и B являются переменными. Эти продукции могут быть полезными: в примере 5.27 мы видели, как использование цепных продукций $E \to T$ и $T \to F$ позволило построить следующую однозначную грамматику для простых арифметических выражений.

$$\begin{aligned} I &\to a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F &\to I \mid (E) \\ T &\to F \mid T * F \\ E &\to T \mid E + T \end{aligned}$$

Вместе с тем, цепные продукции могут усложнять некоторые доказательства и создавать излишние шаги в порождениях, которые по техническим соображениям там совсем не нужны. Например, в продукции $E \to T$ переменную T можно расширить обоими возможными способами, заменив эту продукцию двумя: $E \to F \mid T * F$. Это изменение все еще не избавляет от цепных продукций из-за появления $E \to F$. Дальнейшая замена F дает $E \to I \mid (E) \mid T * F$, однако при этом остается $E \to I$. Но если в этой продукции заменить I всеми шестью возможными способами, то получим следующие продукции.

$E \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \mid (E) \mid T * F$

Как видно, цепных продукций для E нет. Заметим, что продукция $E \to a$ *не является* цепной, единственный символ в ее теле — терминал, а не переменная.

Представленная выше техника расширения цепных продукций до их исчезновения работает довольно часто. Однако она терпит неудачу, если в грамматике есть цикл из цепных продукций, вроде $A \to B$, $B \to C$ и $C \to A$. Техника, гарантирующая результат, включает первоначальное нахождение всех пар переменных A и B, для которых $A \overset{*}{\Rightarrow} B$ получается с использованием последовательности лишь цепных продукций. Заметим, что $A \overset{*}{\Rightarrow} B$ возможно и без использования цепных продукций, например, с помощью продукций $A \to BC$ и $C \to \varepsilon$.

Определив все подобные пары, любую последовательность шагов порождения, в которой $A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \ldots \Rightarrow B_n \Rightarrow \alpha$ с нецепной продукцией $B_n \longrightarrow \alpha$, можно заменить продукцией $A \longrightarrow \alpha$. Однако вначале рассмотрим индуктивное построение пар (A, B), для которых $A \overset{*}{\Rightarrow} B$ получается с использованием лишь цепных продукций. Назовем такую пару *цепной парой* (unit pair).

Базис. (A, A) является цепной парой для любой переменной, т.е. $A \overset{*}{\Rightarrow} A$ за нуль шагов.

Индукция. Предположим, что пара (A, B) определена как цепная, и $B \to C$ — продукция с переменной C. Тогда (A, C) — цепная пара.

Пример 7.10. Рассмотрим грамматику выражений из примера 5.27, воспроизведенную выше. Базис дает цепные пары (E, E), (T, T), (F, F) и (I, I). На индуктивном шаге можно сделать следующие порождения пар.

1. (E, E) и продукция $E \to T$ дают пару (E, T).
2. (E, T) и продукция $T \to F$ — пару (E, F).
3. (E, F) и $F \to I$ дают пару (E, I).
4. (T, T) и $T \to F$ — пару (E, I).
5. (T, F) и $F \to I$ — пару (T, I).
6. (F, F) и $F \to I$ — пару (F, I).

Больше пар, которые можно было бы вывести, нет. На самом деле эти десять пар представляют все порождения, использующие только цепные продукции. □

Способ построения пар теперь очевиден. Нетрудно доказать, что предложенный алгоритм обеспечивает порождение всех нужных пар. Зная эти пары, можно удалить цепные продукции из грамматики и показать эквивалентность исходной и полученной грамматик.

Теорема 7.11. Приведенный выше алгоритм находит все цепные пары грамматики G, и только их.

Доказательство. С помощью индукции по порядку обнаружения пар нетрудно показать, что если пара (A, B) обнаружена, то $A \overset{*}{\underset{G}{\Rightarrow}} B$ получается с использованием лишь цепных продукций. Это оставляется в качестве упражнения.

Предположим, что $A \overset{*}{\underset{G}{\Rightarrow}} B$ получено с использованием лишь цепных продукций. Покажем с помощью индукции по длине порождения, что пара (A, B) будет обнаружена.

Базис. Нуль шагов. Тогда $A = B$, и пара (A, B) обнаруживается согласно базису.

Индукция. Предположим, что $A \overset{*}{\underset{G}{\Rightarrow}} B$ получено за n шагов, $n > 0$, и на каждом из них применялась цепная продукция. Тогда порождение имеет вид $A \overset{*}{\underset{G}{\Rightarrow}} C \Rightarrow B$. Порождение $A \overset{*}{\underset{G}{\Rightarrow}} C$ состоит из $n - 1$ шагов, и по предположению индукции пара (A, C) обнаруживается. Наконец, используем индуктивную часть алгоритма, чтобы по паре (A, C) и продукции $C \to B$ обеспечить обнаружение пары (A, B). □

Для удаления цепных продукций по КС-грамматике $G = (V, T, P, S)$ построим КС-грамматику $G_1 = (V_1, T, P_1, S)$ следующим образом.

1. Найдем все цепные пары грамматики G.
2. Для каждой пары (A, B) добавим к P_1 все продукции $A \to \alpha$, где $B \to \alpha$ — нецепная продукция из P. Заметим, что при $A = B$ все нецепные продукции для B из P просто добавляются к P_1.

Пример 7.12. Продолжим пример 7.10, где был выполнен шаг 1 описанного построения для грамматики выражений из примера 5.27. На рис. 7.1 представлен шаг 2 алгоритма, строящий новое множество продукций. При этом первый член пары становится головой продукций, а в качестве их тел используются все тела нецепных продукций для второго члена.

На заключительном шаге из грамматики (см. рис. 7.1) удаляются все цепные продукции. В итоге получается следующая грамматика без цепных продукций, которая порождает то же самое множество выражений, что и грамматика, изображенная на рис. 5.19.

$$E \to E + T \mid T * F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$
$$T \to T * F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$
$$F \to (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$
$$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

□

Теорема 7.13. Если грамматика G_1 построена по грамматике G с помощью алгоритма удаления цепных продукций, описанного выше, то $L(G_1) = L(G)$.

Пара	Продукции
(E, E)	$E \to E + T$
(E, T)	$E \to T * F$
(E, F)	$E \to (E)$
(E, I)	$E \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(T, T)	$T \to T * F$
(T, F)	$T \to (E)$
(T, I)	$T \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(F, F)	$F \to (E)$
(F, I)	$F \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(I, I)	$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

Рис. 7.1. Грамматика, построенная на шаге 2 алгоритма удаления цепных продукций

Доказательство. Докажем, что $w \in L(G_1)$ тогда и только тогда, когда $w \in L(G)$.

(*Достаточность*) Предположим, $S \overset{*}{\underset{G_1}{\Rightarrow}} w$. Поскольку каждая продукция грамматики G_1 эквивалентна последовательности из нуля или нескольких цепных продукций G, за которой следует нецепная продукция G, из $\alpha \underset{G_1}{\Rightarrow} \beta$ следует $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$. Таким образом, каждый шаг порождения в G_1 может быть заменен одним или несколькими шагами в G. Собрав эти последовательности шагов вместе, получим, что $S \overset{*}{\underset{G}{\Rightarrow}} w$.

(*Необходимость*) Предположим, что $w \in L(G)$. Тогда в соответствии с эквивалентностями из раздела 5.2 цепочка w имеет левое порождение, т.е. $S \overset{*}{\underset{lm}{\Rightarrow}} w$. Где бы в левом порождении ни использовалась цепная продукция, переменная ее тела становится крайней слева в выводимой цепочке и сразу же заменяется. Таким образом, левое порождение в G можно разбить на последовательность “шагов”, в которых нуль или несколько цепных продукций сопровождаются нецепной. Заметим, что любая нецепная продукция, перед которой нет цепных, сама по себе образует такой “шаг”. Но по построению грамматики G_1 каждый из этих шагов может быть выполнен одной ее продукцией. Таким образом, $S \overset{*}{\underset{G_1}{\Rightarrow}} w$. □

Подведем итог различным упрощениям грамматик, описанным выше. Нам желательно преобразовывать любую КС-грамматику в эквивалентную, которая не имеет беспо-

лезных символов, ε-продукций и цепных продукций. При этом немаловажен порядок применения преобразований. Безопасным является следующий.

1. Удалить ε-продукции.
2. Удалить цепные продукции.
3. Удалить бесполезные символы.

Заметим, что подобно тому, как в разделе 7.1.1 результат удаления бесполезных символов зависел от порядка соответствующих шагов, данные три шага должны быть упорядочены именно так, иначе в грамматике могут остаться удаляемые элементы.

Теорема 7.14. Если G — КС-грамматика, порождающая язык, в котором есть хотя бы одна непустая цепочка, то существует другая грамматика G_1, не имеющая бесполезных символов, ε-продукций и цепных продукций, у которой $L(G_1) = L(G) - \{\varepsilon\}$.

Доказательство. Начнем с удаления ε-продукций методом, описанным в разделе 7.1.3. Если затем удалить цепные продукции (см. раздел 7.1.4), то ε-продукции не появятся, поскольку каждое из тел новых продукций совпадает с некоторым телом одной из старых. Наконец, удалим бесполезные символы методом раздела 7.1.1. Поскольку это преобразование только удаляет продукции и символы, не вводя новых, то получаемая грамматика будет по-прежнему свободна от цепных и ε-продукций. □

7.1.5. Нормальная форма Хомского

Завершим изучение грамматических упрощений, показав, что для каждого непустого КС-языка, не включающего ε, существует грамматика G, все продукции которой имеют одну из следующих двух форм.

1. $A \to BC$, где A, B и C — переменные.
2. $A \to a$, где A — переменная, a — терминал.

Кроме того, G не имеет бесполезных символов. Такая форма грамматик называется *нормальной формой Хомского*, или *НФХ*, а грамматики в такой форме — *НФХ-грамматиками*.

Для приведения грамматики к НФХ начнем с ее формы, удовлетворяющей теореме 7.14, т.е. грамматика свободна от бесполезных символов, цепных и ε-продукций. Каждая продукция такой грамматики либо имеет вид $A \to a$, допустимый НФХ, либо имеет тело длиной не менее 2. Нужно выполнить следующие преобразования:

а) устроить так, чтобы все тела длины 2 и более состояли только из переменных;

б) разбить тела длины 3 и более на группу продукций, тело каждой из которых состоит из двух переменных.

Конструкция для a следующая. Для каждого терминала a, встречающегося в продукции длины 2 и более, создаем новую переменную, скажем, A. Эта переменная имеет единственную продукцию $A \to a$. Используем переменную A вместо a везде в телах про-

дукций длины 2 и более. Теперь в теле каждой продукции либо одиночный терминал, либо как минимум две переменные и нет терминалов.

Для шага *б* нужно разбить каждую продукцию вида $A \to B_1B_2\ldots B_k$, где $k \geq 3$, на группу продукций с двумя переменными в каждом теле. Введем $k-2$ новых переменных C_1, C_2, …, C_{k-2} и заменим исходную продукцию на $k-1$ следующих продукций.

$$A \to B_1C_1,\ C_1 \to B_2C_2,\ \ldots,\ C_{k-3} \to B_{k-2}C_{k-2},\ C_{k-2} \to B_{k-1}B_k$$

Пример 7.15. Приведем грамматику из примера 7.12 к НФХ. Для части *а* заметим, что у грамматики есть восемь терминалов, *a*, *b*, 0, 1, +, *, (и), каждый из которых встречается в теле, не являющемся одиночным терминалом. Таким образом, нужно ввести восемь новых переменных, соответствующих этим терминалам, и восемь следующих продукций, где переменная заменяется “своим” терминалом.

$$A \to a \quad B \to b \quad Z \to 0 \quad O \to 1$$

$$P \to + \quad M \to * \quad L \to (\quad R \to)$$

Введя эти продукции и заменив каждый терминал в теле, не являющемся одиночным терминалом, соответствующей переменной, получим грамматику, изображенную на рис. 7.2.

$E \to EPT \mid TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$
$T \to TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$
$F \to LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$
$I \to a \mid b \mid IA \mid IB \mid IZ \mid IO$
$A \to \boldsymbol{a}$
$B \to \boldsymbol{b}$
$Z \to 0$
$O \to 1$
$P \to +$
$M \to *$
$L \to ($
$R \to)$

Рис. 7.2. Преобразование тел к одиночным терминалам или нескольким переменным

Теперь все продукции находятся в НФХ, за исключением тех, тела которых имеют длину 3: *EPT*, *TMF*, *LER*. Некоторые из этих тел встречаются в нескольких продукциях, но каждое тело преобразуется один раз.

Для тела *EPT* вводится новая переменная C_1, и продукция $E \to EPT$ меняется на $E \to EC_1$ и $C_1 \to PT$. Для тела *TMF* вводится переменная C_2. Две продукции с этим телом, $E \to TMF$ и $T \to TMF$, меняются на $E \to TC_2$, $T \to TC_2$ и $C_2 \to MF$. Для *LER* вводится C_3 и три продукции с этим телом, $E \to LER$, $T \to LER$ и $F \to LER$, меняются на $E \to LC_3$, $T \to LC_3$, $F \to LC_3$ и $C_3 \to ER$. Окончательная НФХ-грамматика показана на рис. 7.3.

$E \to EC_1 \mid TC_2 \mid LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$

$T \to TC_2 \mid LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$

$F \to LC_3 \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$

$I \to a \mid b \mid IA \mid IB \mid IZ \mid IO$

$A \to a$

$B \to b$

$Z \to 0$

$O \to 1$

$P \to +$

$M \to *$

$L \to ($

$R \to)$

$C_1 \to PT$

$C_2 \to MF$

$C_3 \to ER$

Рис. 7.3. Преобразование тел к одиночным терминалам или двум переменным

Теорема 7.16. Если G — КС-грамматика, язык которой содержит хотя бы одну непустую цепочку, то существует НФХ-грамматика G_1, причем $L(G_1) = L(G) - \{\varepsilon\}$.

Доказательство. По теореме 7.14 можно найти КС-грамматику G_2, для которой $L(G_2) = L(G) - \{\varepsilon\}$, причем G_2 свободна от бесполезных символов, цепных и ε-продукций. Конструкция, преобразующая G_2 в НФХ-грамматику G_1, изменяет продукции таким образом, что каждая продукция из G_2 может быть проимитирована одной или несколькими продукциями из G_1. Наоборот, каждая из введенных в G_1 переменных соответствует лишь одной продукции, поэтому эти переменные можно использовать только надлежащим образом. Более строго, докажем, что $w \in L(G_2)$ тогда и только тогда, когда $w \in L(G_1)$.

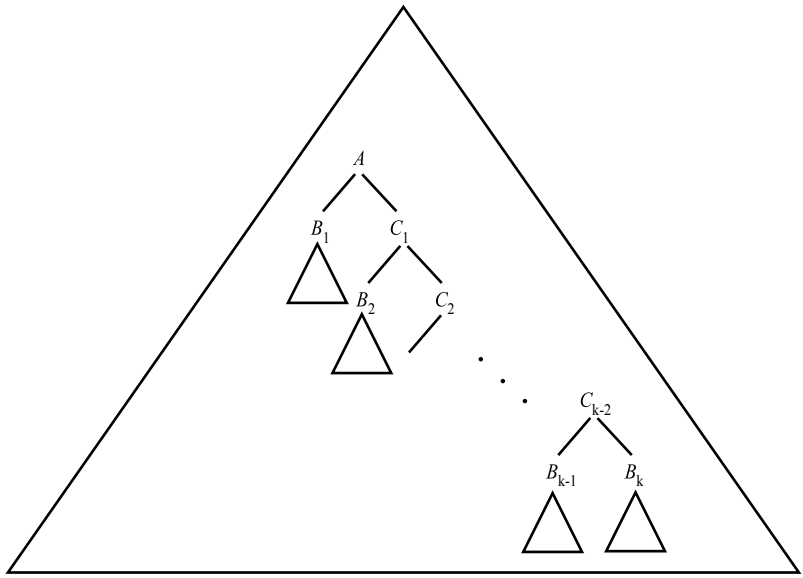
(*Необходимость*) Если w порождается в G_2, то легко заменить каждую использованную продукцию, скажем, $A \to X_1X_2\ldots X_k$, последовательностью продукций из G_1. Таким образом, один шаг порождения в G_2 превращается в один или несколько шагов в порождении w, использующем продукции G_1. Во-первых, если X_i — терминал, то G_1 имеет соответствующую переменную B_i и продукцию $B_i \to X_i$. Во-вторых, если $k > 2$, то G_1 имеет продукции $A \to B_1C_1$, $C_1 \to B_2C_2$ и так далее, где B_i есть либо переменная, введенная для терминала X_i, либо сам X_i, если это переменная. Эти продукции имитируют в G_1 один шаг порождения в G_2, использующий продукцию $A \to X_1X_2\ldots X_k$. Делаем вывод, что в G_1 существует порождение w, поэтому $w \in L(G_1)$.

(*Достаточность*) Предположим, что $w \in L(G_1)$. Тогда в G_1 существует дерево разбора с отметкой корня S и кроной w. Преобразуем это дерево в дерево разбора в G_2, также имеющее отметку корня S и крону w.

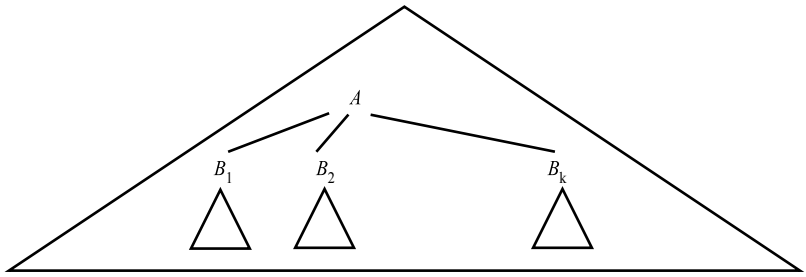
Сначала “сделаем откат” части *б* построения НФХ. Предположим, существует узел с отметкой A и сыновьями, отмеченными B_1 и C_1, где C_1 — одна из переменных, введенных в части *б*. Тогда данная часть дерева разбора должна выглядеть, как на рис. 7.4, *а*. Поскольку каждая из этих введенных переменных соответствует лишь одной продукции, существует только один способ для их возникновения, и все эти переменные, предназначенные для обработки продукции $A \to B_1B_2\ldots B_k$, должны появляться вместе (см. рис. 7.4, *а*).

Каждый такой куст узлов в дереве разбора можно заменить продукцией, которую они представляют. Преобразование дерева разбора показано на рис. 7.4, *б*.

Полученное дерево все еще не обязательно является деревом разбора в G_2. Причина в том, что на шаге *а* построения НФХ были введены другие переменные, порождающие



а)



б)

Рис. 7.4. Дерево разбора должно использовать введенные переменные специальным образом

одиночные терминалы. Однако их можно обнаружить в полученном дереве разбора и заменить узел, отмеченный такой переменной A, и его единственного сына, отмеченного a, одиночным узлом с отметкой a. Теперь каждый внутренний узел дерева разбора образует продукцию G_2, и мы приходим к выводу, что $w \in L(G_2)$. □

Нормальная форма Грейбах

Существует еще одна интересная нормальная форма для грамматик, которая не будет обоснована. Любой непустой язык, не включающий ε, есть $L(G)$ для некоторой грамматики G с продукциями вида $A \to a\alpha$, где a — терминал, а α — цепочка из нуля или нескольких переменных. Преобразование грамматики к этой форме является сложным, даже если задачу упростить, скажем, начав с НФХ-грамматики. Иначе говоря, первая переменная каждой продукции расширяется до тех пор, пока не будет получен терминал. Однако на этом пути возможны циклы, в которых терминал не достигается, и этот процесс необходимо "замкнуть". Для того чтобы породить все последовательности переменных, которые могут появиться на пути к порождению этого терминала, создается продукция с терминалом в качестве первого символа тела и переменными вслед за ним.

Эта форма, называемая *нормальной формой Грейбах*, по имени Шейлы Грейбах (Sheila Greibach), которая первой дала способ построения таких грамматик, имеет несколько интересных следствий. Поскольку каждое использование продукции вводит ровно один терминал в выводимую цепочку, цепочка длины n порождается в точности за n шагов. Кроме того, если применить конструкцию из теоремы 6.13 для построения МП-автомата по грамматике в нормальной форме Грейбах, то получается МП-автомат без ε-переходов, показывающий, что такие переходы в МП-автомате всегда можно удалить.

7.1.6. Упражнения к разделу 7.1

7.1.1. (*) Найдите грамматику, не содержащую бесполезных символов и эквивалентную следующей грамматике.

$S \to AB \mid CA$

$A \to a$

$B \to BC \mid AB$

$C \to aB \mid b$

7.1.2. Рассмотрите следующую грамматику:

$S \to ASB \mid \varepsilon$

$A \to aAS \mid a$

$B \to SbS \mid A \mid bb$

а) удалите ε-продукции;

б) удалите цепные продукции;

в) есть ли здесь бесполезные символы? Если да, удалите их;

г) приведите грамматику к нормальной форме Хомского.

7.1.3. Выполните упражнение 7.1.2 со следующей грамматикой.

$S \to 0A0 \mid 1B1 \mid BB$

$A \to C$

$B \to S \mid A$

$C \to S \mid \varepsilon$

7.1.4. Выполните упражнение 7.1.2 со следующей грамматикой.

$S \to AAA \mid B$

$A \to aA \mid B$

$B \to \varepsilon$

7.1.5. Выполните упражнение 7.1.2 со следующей грамматикой.

$S \to aAa \mid bBb \mid \varepsilon$

$A \to C \mid a$

$B \to C \mid b$

$C \to CDE \mid \varepsilon$

$D \to A \mid B \mid ab$

7.1.6. Постройте НФХ-грамматику для множества цепочек сбалансированных скобок. Можно начать с грамматики, находящейся в НФХ.

7.1.7. (!!) Пусть G — КС-грамматика с p продукциями и длины тел продукций не превосходят n. Докажите, что если $A \overset{*}{\underset{G}{\Rightarrow}} \varepsilon$, то найдется порождение ε из A, в котором не более, чем $(n^p - 1)/(n - 1)$ шагов. Как близко можно на самом деле подойти к этой границе?

7.1.8. (!) Предположим, что нам дана грамматика G с n продукциями, ни одна из которых не является ε-продукцией, и мы преобразуем ее в НФХ:

а) докажите, что НФХ-грамматика имеет не более, чем $O(n^2)$ продукций;

б) докажите, что у НФХ-грамматики число продукций может быть прямо пропорциональным n^2. *Указание*. Рассмотрите конструкцию удаления цепных продукций.

7.1.9. Дайте индуктивные доказательства, необходимые для завершения следующих теорем:

а) часть теоремы 7.4, в которой доказывалось, что обнаруживаемые символы действительно являются порождающими;

б) теорема 7.6 в обе стороны, где доказывалась корректность алгоритма из раздела 7.1.2 для обнаружения достижимых символов;

в) часть теоремы 7.11, где доказывалось, что все обнаруженные пары действительно являются цепными парами.

7.1.10. (***!**) Можно ли для каждого КС-языка найти грамматику, все продукции которой имеют вид или $A \to BCD$ (тело состоит из трех переменных), или $A \to a$ (тело образовано одиночным терминалом)? Приведите либо доказательство, либо контрпример.

7.1.11. В этом упражнении показывается, что для каждого КС-языка L, содержащего хотя бы одну непустую цепочку, найдется КС-грамматика в нормальной форме Грейбах, порождающая $L - \{\varepsilon\}$. Напомним, что тела продукций грамматики в нормальной форме Грейбах (НФГ) начинаются терминалом. В построении используется ряд лемм и конструкций.

1. Пусть КС-грамматика G имеет продукцию $A \to \alpha B \beta$, и всеми продукциями для B являются $B \to \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$. Заменив $A \to \alpha B \beta$ всеми продукциями, у которых вместо B подставлены тела всех B-продукций, получим $A \to \alpha\gamma_1\beta \mid \alpha\gamma_2\beta \mid \ldots \mid \alpha\gamma_n\beta$. Построенная грамматика порождает тот же язык, что и G.

   Далее будем предполагать, что грамматика G для L находится в нормальной форме Хомского, а ее переменные обозначены $A_1, A_2, \ldots, A_k$.

2. (***!**) Докажите, что путем повторных применений преобразования из части 1 грамматику G можно превратить в эквивалентную грамматику, у которой тело каждой продукции для A_i начинается или терминалом, или A_j для некоторого $j \geq i$. В обоих случаях все символы после первого в теле продукции — переменные.

3. (**!**) Пусть G_1 — грамматика, полученная из G путем выполнения шага 2. Пусть A_i — произвольная переменная и $A_i \to A_i\alpha_1 \mid \ldots \mid A_i\alpha_m$ — все A_i-продукции, тело которых начинается с A_i. Пусть $A_i \to \beta_1 \mid \ldots \mid \beta_p$ — все остальные A_i-продукции. Заметим, что каждое β_i начинается либо терминалом, либо переменной с индексом больше i. Введем новую переменную B_i и заменим первую группу из m продукций следующими:

   $A_i \to \beta_1 \mid \ldots \mid \beta_p$
   $B_i \to \alpha_1 B_i \mid \alpha_1 \mid \ldots \mid \alpha_m B_i \mid \alpha_m$

   Докажите, что полученная грамматика порождает тот же язык, что и G или G_1.

4. (***!**) Пусть после шага 3 получена грамматика G_2. Отметим, что тела всех A_i-продукций начинаются или терминалом, или A_j с $j > i$. Кроме того, тела B_i-продукций начинаются или терминалом, или некоторым A_j. Докажите, что G_2 имеет эквивалентную грамматику в НФГ. *Указание*. Сначала преобразуй-

те должным образом продукции для A_k, затем для A_{k-1} и так далее до A_1, используя часть 1. Затем снова с помощью части 1 преобразуйте продукции для B_i в любом порядке.

7.1.12. Используйте построения из упражнения 7.1.11 для преобразования в НФГ следующей грамматики.

$S \to AA \mid 0$
$A \to SS \mid 1$

7.2. Лемма о накачке для контекстно-свободных языков

В этом разделе развивается инструмент доказательства, что некоторые языки не являются контекстно-свободными. Теорема, называемая “леммой о накачке для контекстно-свободных языков”[1], гласит, что в любой достаточно длинной цепочке КС-языка можно найти две близлежащие короткие подцепочки (одна из них может быть пустой) и совместно их “накачивать”. Таким образом, обе подцепочки можно повторить i раз для любого целого i, и полученная цепочка также будет принадлежать языку.

Эта теорема отличается от аналогичной для регулярных языков (теорема 4.1), гласящей, что всегда можно найти одну короткую цепочку для ее накачки. Разница видна, если рассмотреть язык типа $L = \{0^n1^n \mid n \geq 1\}$. Можно показать, что он нерегулярен, если зафиксировать n и накачать подцепочку из нулей, получив цепочку, в которой символов 0 больше, чем 1. Однако лемма о накачке для КС-языков утверждает, что можно найти две короткие цепочки, поэтому нам пришлось бы использовать для накачки цепочку из нулей и цепочку из единиц, порождая таким образом только цепочки из L. Этот результат нас устраивает, так как L — КС-язык, а для построения цепочек, не принадлежащих языку L, лемма о накачке для КС-языков использоваться и не должна.

7.2.1. Размер деревьев разбора

Первый шаг на пути к лемме о накачке для КС-языков состоит в том, чтобы рассмотреть вид и размер деревьев разбора. Одно из применений НФХ — преобразовывать деревья разбора в бинарные (двоичные). Такие деревья имеют ряд удобных свойств, и одно из них используется здесь.

Теорема 7.17. Пусть дано дерево разбора, соответствующее НФХ-грамматике $G = (V, T, P, S)$, и пусть кроной дерева является терминальная цепочка w. Если n — наибольшая длина пути (от корня к листьям), то $|w| \leq 2^{n-1}$.

Доказательство. Докажем с помощью простой индукции по n.

---

[1] Напомним, что в русскоязычной литературе был принят термин “лемма о разрастании”, но “накачка”, на наш взгляд, точнее отражает суть происходящего. — *Прим. ред.*

Базис. $n = 1$. Напомним, что длина пути есть число ребер, т.е. на 1 меньше числа узлов. Таким образом, дерево с максимальной длиной пути 1 состоит из корня и листа, отмеченного терминалом. Цепочка w является этим терминалом, и $|w| = 1$. Поскольку $2^{n-1} = 2^0 = 1$, базис доказан.

Индукция. Предположим, самый длинный путь имеет длину n, и $n > 1$. Корень дерева использует продукцию, которая должна иметь вид $A \to BC$, поскольку $n > 1$, т.е. нельзя начать дерево, использовав продукцию с терминалом. Ни один из путей в поддеревьях с корнями в B и C не может иметь длину больше, чем $n - 1$, так как в этих путях исключено ребро от корня к сыну, отмеченному B или C. Таким образом, по предположению индукции эти два поддерева имеют кроны длины не более 2^{n-2}. Крона всего дерева представляет собой конкатенацию этих двух крон, поэтому имеет длину не более $2^{n-2} + 2^{n-2} = 2^{n-2}$. Шаг индукции доказан. □

7.2.2. Утверждение леммы о накачке

Лемма о накачке для КС-языков подобна лемме о накачке для регулярных языков, но каждая цепочка z КС-языка L разбивается на пять частей, и совместно накачиваются вторая и четвертая из них.

Теорема 7.18. (Лемма о накачке для КС-языков.) Пусть L — КС-язык. Тогда существует такая константа n, что если z — произвольная цепочка из L, длина которой не меньше n, то можно записать $z = uvwxy$, причем выполняются следующие условия.

1. $|vwx| \le n$. Таким образом, средняя часть не слишком длинная.
2. $vx \neq \varepsilon$. Поскольку v и x — подцепочки, которые должны "накачиваться", это условие гласит, что хотя бы одна из них непуста.
3. $uv^iwx^iy \in L$ для всех $i \geq 0$. Две цепочки, v и x, могут быть "накачаны" произвольное число раз, включая 0, и полученная при этом цепочка также будет принадлежать L.

Доказательство. Вначале для L найдем грамматику G в нормальной форме Хомского. Технически это невозможно, если L есть КС-язык $\varnothing$ или $\{\varepsilon\}$. Однако при $L = \varnothing$ утверждение теоремы, которое говорит о цепочке z, не может быть нарушено, поскольку такой цепочки z нет в $\varnothing$. НФХ-грамматика G в действительности порождает $L - \{\varepsilon\}$, но это также не имеет значения, поскольку выбирается $n > 0$, и z никак не сможет быть ε.

Итак, пусть НФХ-грамматика $G = (V, T, P, S)$ имеет m переменных и порождает язык $L(G) = L - \{\varepsilon\}$. Выберем $n = 2^m$. Предположим, что z из L имеет длину не менее n. По теореме 7.17 любое дерево разбора, наибольшая длина путей в котором не превышает m, должно иметь крону длиной не более $2^{m-1} = n/2$. Такое дерево разбора не может иметь крону z, так как z для этого слишком длинная. Таким образом, любое дерево разбора с кроной z имеет путь длиной не менее $m + 1$.

На рис. 7.5 представлен самый длинный путь в дереве для z, где k не менее m, и путь имеет длину $k + 1$. Поскольку $k \geq m$, на этом пути встречается не менее $m + 1$ переменных A_0, A_1, …, A_k. Но V содержит всего m различных переменных, поэтому хотя бы две из $m + 1$ последних переменных на пути (т.е. от A_{k-m} до A_k включительно) должны совпадать. Пусть $A_i = A_j$, где $k - m \leq i \leq j \leq k$.

Тогда дерево можно разделить так, как показано на рис. 7.6. Цепочка w является кроной поддерева с корнем A_j. Цепочки v и x — это цепочки соответственно слева и справа от w в кроне большего поддерева с корнем A_i. Заметим, что цепных продукций нет, поэтому v и x не могут одновременно быть ε, хотя одна из них и может. Наконец, u и y образуют части z, лежащие соответственно слева и справа от поддерева с корнем A_i.

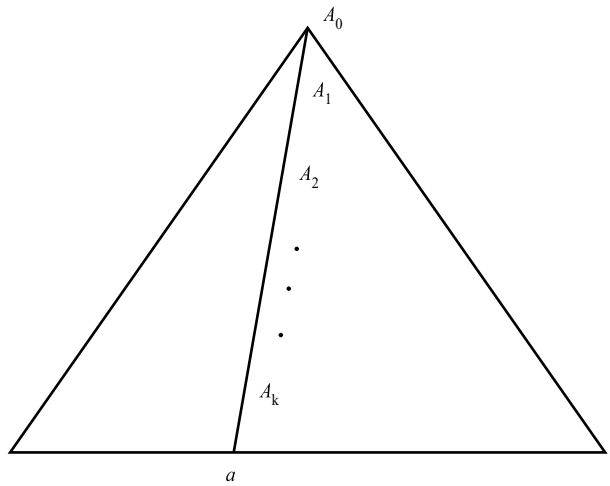


Рис. 7.5. Каждая достаточно длинная цепочка в L *должна иметь длинный путь в своем дереве разбора*

Если $A_i = A_j = A$, то по исходному дереву можно построить новое дерево разбора, как показано на рис. 7.7, *а*. Сначала можно заменить поддерево с корнем A_i, имеющее крону vwx, поддеревом с корнем A_j, у которого крона w. Это допустимо, поскольку корни обоих поддеревьев отмечены одной и той же переменной A. Полученное дерево представлено на рис. 7.7, *б*. Оно имеет крону и соответствует случаю $i = 0$ в шаблоне цепочек uv^iwx^iy.

Еще одна возможность представлена на рис. 7.7, *в*. Там поддерево с корнем A_j заменено поддеревом с корнем A_i. Допустимость этой замены также обусловлена тем, что отметки корней совпадают. Кроной этого дерева является uv^2wx^2y. Если бы мы затем заменили поддерево с кроной w (см. рис. 7.7, *в*) бо́льшим поддеревом с кроной vwx, то получили бы дерево с кроной uv^3wx^3y и так далее для любого показателя i. Итак, в G существуют деревья разбора для всех цепочек вида uv^iwx^iy, и лемма о накачке почти доказана.

Осталось условие 1, гласящее, что $|vwx| \leq n$. Мы выбирали A_i как можно ближе к кроне дерева, поэтому $k - i \leq m$. Таким образом, самый длинный путь в поддереве с корнем

A_i имеет длину не более $m + 1$. По теореме 7.17 поддерево с корнем A_i имеет крону, длина которой не больше, чем $2^m = n$. □

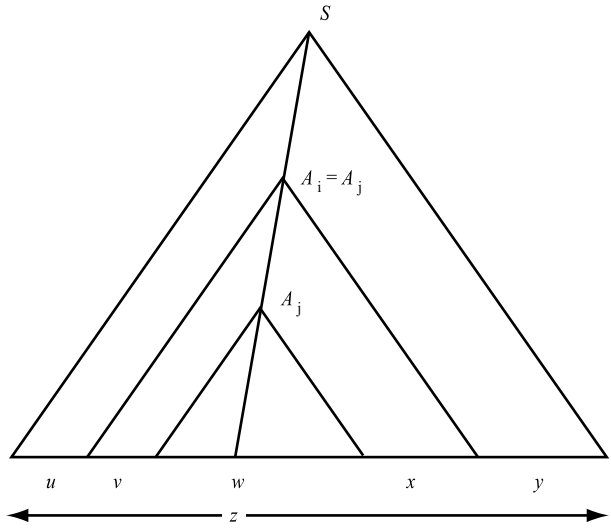


Рис. 7.6. Разделение цепочки z *для накачивания*

7.2.3. Приложения леммы о накачке к КС-языкам

Отметим, что лемма о накачке для КС-языков, как и для регулярных языков, используется в виде "игры с противником" следующим образом.

1. Мы выбираем язык L, желая доказать, что он не контекстно-свободный.
2. Наш "противник" выбирает заранее неизвестное нам n, поэтому мы должны рассчитывать на любое возможное значение.
3. Мы выбираем z и при этом можем использовать n как параметр.
4. Противник разбивает z на $uvwxy$, соблюдая ограничения $|vwx| \le n$ и $vx \neq \varepsilon$.
5. Мы "выигрываем", если можем, выбирая i и показывая, что uv^iwx^iy не принадлежит языку L.

Рассмотрим несколько примеров языков, о которых с помощью леммы о накачке можно доказать, что они не контекстно-свободные. Первый пример показывает, что хотя цепочки контекстно-свободных языков могут иметь по две соответствующие друг другу группы символов, но три такие группы уже невозможны.

Пример 7.19. Пусть $L = \{0^n1^n2^n \mid n \geq 1\}$, т.е. L состоит из цепочек вида $\mathbf{0}^+\mathbf{1}^+\mathbf{2}^+$ с одинаковыми количествами каждого из символов, например, 012, 001122 и т.д. Предполо-

жим, что L контекстно-свободный. Тогда существует целое n из леммы о накачке.[2] Выберем $z = 0^n1^n2^n$.

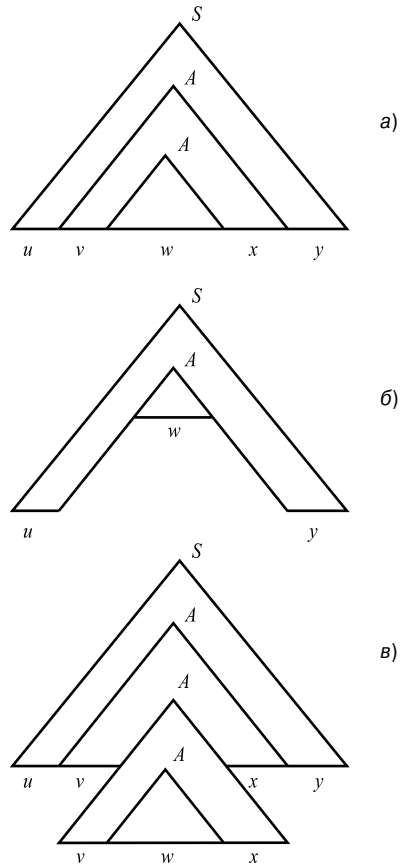


Рис. 7.7. Накачивание цепочек v *и* x *0 раз и 2 раза*

Предположим, что “противник” разбивает z как $z = uvwxy$, где $|vwx| \le n$ и v и x не равны ε одновременно. Тогда нам известно, что vwx не может включать одновременно нули и двойки, поскольку последний нуль и первая двойка разделены $n + 1$ позициями. Докажем, что L содержит некоторую цепочку, которая не может быть в L, получив тем самым противоречие к предположению, что L контекстно-свободный. Возможны следующие случаи.

1. vwx не имеет двоек, т.е. vx состоит только из нулей и единиц и содержит хотя бы один из этих символов. Тогда цепочка uwy, которая по лемме о накачке должна быть

[2] Напомним, что это n есть константа, обеспеченная леммой о накачке и не имеющая ничего общего с локальной переменной n, использованной в определении языка L.

в L, имеет n двоек, но меньше, чем n нулей или единиц. Следовательно, она не принадлежит L, и в этом случае L не контекстно-свободен.

2. vwx не имеет нулей. Аналогично, uwy имеет n нулей, но меньше двоек или единиц, поэтому не принадлежит L.

В любом случае приходим к выводу, что L содержит цепочку, которая не может ему принадлежать. Это противоречие позволяет заключить, что наше предположение ложно, т.е. L *не является* КС-языком. □

Еще одно свойство КС-языков состоит в том, что в их цепочках не может быть двух соответствующих друг другу перемежающихся пар равных количеств символов. Эта идея уточняется следующим примером.

Пример 7.20. Пусть $L = \{0^i1^j2^i3^j \mid i \geq 1 \text{ и } j \geq 1\}$. Если он контекстно-свободен, то пусть n — константа для L, и выберем $z = 0^n1^n2^n3^n$. Можно записать $z = uvwxy$, соблюдая обычные ограничения $|vwx| \leq n$ и $vx \neq \varepsilon$. Тогда vwx или состоит из символов одного вида, или захватывает символы двух различных соседних видов.

Если vwx состоит из символов одного вида, то uwy имеет по n символов трех различных видов и меньше, чем n символов четвертого вида. Таким образом, uwy не может быть в L. Если vwx захватывает символы двух различных соседних видов, скажем, единицы и двойки, то в uwy их не хватает. Если не хватает единиц, то, поскольку там есть n троек, эта цепочка не может быть в L. Если же не хватает двоек, то uwy также не может быть в L, поскольку содержит n нулей. Получаем противоречие к предположению о том, что L — КС-язык, и приходим к выводу, что он таковым не является. □

В заключительном примере покажем, что в цепочках КС-языков не может быть двух одинаковых цепочек произвольной длины, если они выбираются в алфавите, состоящем более чем из одного символа. Следствием этого замечания, между прочим, является то, что КС-грамматики не являются подходящим механизмом для описания определенных “семантических” ограничений в языках программирования, например, что идентификатор должен быть объявлен до его использования. На практике для запоминания объявленных идентификаторов используется другой механизм, “таблица символов”, и никто не пытается строить синтаксический анализатор, который проверял бы соблюдение принципа “определение до использования”.

Пример 7.21. Пусть $L = \{ww \mid w \in \{0, 1\}^*\}$, т.е. L состоит из повторяющихся цепочек, например, ε, 0101, 00100010 или 110110. Если он контекстно-свободный, то пусть n — константа из леммы о накачке для L. Рассмотрим цепочку $z = 0^n1^n0^n1^n$. Очевидно, $z \in L$.

Следуя шаблону предыдущих примеров, можно записать $z = uvwxy$, причем $|vwx| \leq n$ и $vx \neq \varepsilon$. Покажем, что uwy не принадлежит L, тем самым доказав от противного, что L не может быть КС-языком.

Заметим сразу, что, поскольку $|vwx| \leq n$, то $|uwy| \geq 3n$. Таким образом, если uwy является повторением цепочки, скажем, tt, то t имеет длину не менее $3n/2$. Возможны несколько вариантов в зависимости от расположения vwx в пределах z.

1. Предположим, vwx находится в пределах первых n нулей. Для определенности пусть vx состоит из k нулей, где $k > 0$. Тогда uwy начинается с $0^{n-k}1^n$. Поскольку $|uwy| = 4n - k$ и по предположению $uwy = tt$, то $|t| = 2n - k/2$. Таким образом, t не заканчивается в первом блоке из единиц, т.е. заканчивается символом 0. Но uwy заканчивается единицей, поэтому не может равняться tt.
2. Предположим, vwx захватывает первый блок нулей и первый блок единиц. Возможно, vx состоит только из нулей, если $x = \varepsilon$. Тогда uwy не может быть вида tt по той же причине, что и в случае 1. Если же vx содержит хотя бы одну единицу, то t, длина которой не менее $3n/2$, должна заканчиваться на 1^n, поскольку uwy заканчивается на 1^n. Однако из n единиц состоит только последний блок в uwy, поэтому t не может повторяться в uwy.
3. Если vwx содержится в первом блоке единиц, то uwy не может быть в L по тем же причинам, что и во второй части случая 2.
4. Предположим, vwx захватывает первый блок единиц и второй блок нулей. Если vx не имеет нулей, то все получается так же, как если бы vwx содержалась в первом блоке единиц. Если vx содержит хотя бы один нуль, то uwy начинается блоком из n нулей, как и t, если $uwy = tt$. Однако в uwy второго блока из n нулей для t нет, поэтому uwy не может быть в L.
5. В остальных случаях, когда vwx находится во второй части z, аргументы симметричны по отношению к случаям, когда vwx содержится в первой части z.

Итак, в любом случае uwy не принадлежит L, и мы приходим к выводу, что L не контекстно-свободный.

7.2.4. Упражнения к разделу 7.2

7.2.1. Используйте лемму о накачке для КС-языков, чтобы показать, что каждый из следующих языков не контекстно-свободный.

а) (*) $\{a^i b^j c^k \mid i < j < k\}$;

б) $\{a^n b^n c^i \mid i \leq n\}$;

в) $\{0^p \mid p$ — простое$\}$. *Указание*. Используйте те же идеи, что и в примере 4.3, где доказывалась нерегулярность этого языка;

г) (*!) $\{0^i 1^j \mid j = i^2\}$;

д) (!) $\{a^n b^n c^i \mid n \leq i \leq 2n\}$;

е) (!) $\{ww^Rw \mid w$ — цепочка из нулей и единиц$\}$, т.е. множество цепочек, состоящих из цепочки w, за которой записаны ее обращение и она же еще раз, например 001100001.

7.2.2. (!) Когда мы пытаемся применить лемму о накачке к КС-языку, "выигрывает противник", и нам не удается завершить доказательство. Покажите, что является ошибочным, когда в качестве L выбирается один из следующих языков:

а) $\{00, 11\}$;

б) (*) $\{0^n1^n \mid n \geq 1\}$;

в) (*) множество палиндромов в алфавите $\{0, 1\}$.

7.2.3. (!) Существует более сильная версия леммы о накачке для КС-языков, известная как *лемма Огдена*. Она отличается от доказанной леммы о накачке тем, что позволяет нам сосредоточиться на любых n "выделенных" позициях цепочки z и гарантирует, что накачиваемые цепочки содержат от 1 до n выделенных позиций. Преимущество этого свойства в том, что язык может иметь цепочки, состоящие из двух частей, одна из которых может быть накачана без создания цепочек, не принадлежащих языку, тогда как вторая при накачке *обязательно* порождает цепочки вне языка. Если мы не можем утверждать, что накачка имеет место во второй части, то мы не можем завершить доказательство того, что язык не контекстно-свободный. Формальное утверждение леммы Огдена состоит в следующем. Для любого КС-языка L существует такая константа n, что если z — произвольная цепочка из L длиной не менее n, в которой выделено не менее n *различных* позиций, то z можно записать в виде $uvwxy$, причем выполняются следующие условия.

1. vwx имеет не более n выделенных позиций.
2. vx имеет хотя бы одну выделенную позицию.
3. $uv^iwx^iy \in L$ для всех $i \geq 0$.

Докажите лемму Огдена. *Указание*. Доказательство на самом деле весьма похоже на доказательство леммы о накачке (теорема 7.18), если мы представим себе, что в том доказательстве невыделенные позиции цепочки z отсутствуют, когда выбирается длинный путь в дереве разбора для z.

7.2.4. (*) Используйте лемму Огдена (упражнение 7.2.3) для упрощения доказательства того, что $L = \{ww \mid w \in \{0, 1\}^*\}$ — не КС-язык (см. пример 7.21). *Указание*. В выбранной цепочке z сделайте выделенной только одну группу из n последовательных символов.

7.2.5. Используйте лемму Огдена (упражнение 7.2.3) для доказательства того, что следующие языки не являются контекстно-свободными:

а) (!) $\{0^i1^j0^k \mid j = \max(i, k)\}$;

б) (!!) $\{a^n b^n c^i \mid i \neq n\}$. *Указание*. Рассмотрите цепочку $z = a^n b^n c^{n!}$, где n — константа из леммы Огдена.

7.3. Свойства замкнутости контекстно-свободных языков

Рассмотрим некоторые операции над контекстно-свободными языками, которые гарантированно порождают КС-языки. Многие из этих свойств замкнутости соответствуют теоремам для регулярных языков из раздела 4.2. Однако есть и отличия.

Вначале введем операцию подстановки, по которой каждый символ в цепочках из одного языка заменяется целым языком. Эта операция обобщает гомоморфизм, рассмотренный в разделе 4.2.3, и является полезной в доказательстве свойств замкнутости КС-языков, относительно некоторых других операций, например, регулярных (объединение, конкатенация и замыкание). Покажем, что КС-языки замкнуты относительно гомоморфизма и обратного гомоморфизма. В отличие от регулярных языков, КС-языки не замкнуты относительно пересечения и разности. Однако пересечение или разность КС-языка и регулярного языка всегда является КС-языком.

7.3.1. Подстановки

Пусть Σ — алфавит. Предположим, для каждого символа a из Σ выбран язык L_a. Выбранные языки могут быть в любых алфавитах, не обязательно Σ и не обязательно одинаковых. Выбор языков определяет функцию s (*подстановка*, substitution) на Σ, и L_a обозначается как $s(a)$ для каждого символа a.

Если $w = a_1 a_2 \ldots a_n$ — цепочка из Σ^*, то $s(w)$ представляет собой язык всех цепочек $x_1 x_2 \ldots x_n$, у которых для $i = 1, 2, \ldots, n$ цепочка x_i принадлежит языку $s(a_i)$. Иными словами, $s(w)$ является конкатенацией языков $s(a_1)s(a_2)\ldots s(a_n)$. Определение s можно распространить на языки: $s(L)$ — это объединение $s(w)$ по всем цепочкам w из L.

Пример 7.22. Пусть $s(0) = \{a^n b^n \mid n \geq 1\}$ и $s(1) = \{aa, bb\}$, т.е. s — подстановка на алфавите $\Sigma = \{0, 1\}$. Язык $s(0)$ представляет собой множество цепочек с одним или несколькими символами a, за которыми следует такое же количество b, а $s(1)$ — конечный язык, состоящий из двух цепочек aa и bb.

Пусть $w = 01$. Тогда $s(w)$ есть конкатенация языков $s(0)s(1)$. Точнее, $s(w)$ состоит из всех цепочек вида $a^n b^n aa$ и $a^n b^{n+2}$, где $n \geq 1$.

Теперь предположим, что $L = L(\mathbf{0}^*)$, т.е. множество всех цепочек из нулей. Тогда $s(L) = (s(0))^*$. Этот язык представляет собой множество всех цепочек вида

$$a^{n_1} b^{n_1} a^{n_2} b^{n_2} \ldots a^{n_k} b^{n_k}$$

для некоторого $k \geq 0$ и произвольной последовательности положительных целых $n_1, n_2, \ldots, n_k$. Он включает, например, цепочки ε, *aabbaaabbb* и *abaabbabab*. □

Теорема 7.23. Если L — КС-язык в алфавите Σ, а s — подстановка на Σ, при которой $s(a)$ является КС-языком для каждого a из Σ, то $s(L)$ также является КС-языком.

Доказательство. Основная идея состоит в том, что мы можем взять КС-грамматику для L и заменить каждый терминал a стартовым символом грамматики для языка $s(a)$. В результате получим единственную грамматику, порождающую $s(L)$. Однако тут нужно быть аккуратным.

Более формально, пусть грамматика $G = (V, \Sigma, P, S)$ задает язык L, а грамматика $G_a = (V_a, T_a, P_a, S_a)$ — язык, подставляемый вместо каждого a из Σ. Поскольку для переменных можно выбирать любые имена, обеспечим, чтобы множества имен переменных V и V_a (для всех a) не пересекались. Цель такого выбора имен — гарантировать, что при сборе продукций разных грамматик в одно множество невозможно случайно смешать продукции двух грамматик, и, таким образом, получить порождения, невозможные в данных грамматиках.

Построим новую грамматику $G' = (V', T', P', S)$ для $s(L)$ следующим образом.

- V' представляет собой объединение V и всех V_a по a из Σ.
- T' является объединением T_a по a из Σ.
- P' состоит из следующих продукций.

1. Все продукции каждого из P_a для a из Σ.
2. Все продукции P, но с изменением везде в их телах каждого терминала a на S_a.

Таким образом, все деревья разбора в грамматике G' начинаются как деревья разбора в G, но вместо того, чтобы порождать крону в Σ^*, они содержат границу, на которой все узлы отмечены переменными S_a вместо a из Σ. Каждый такой узел является корнем дерева в G_a, крона которого представляет собой терминальную цепочку из $s(a)$ (рис. 7.8).

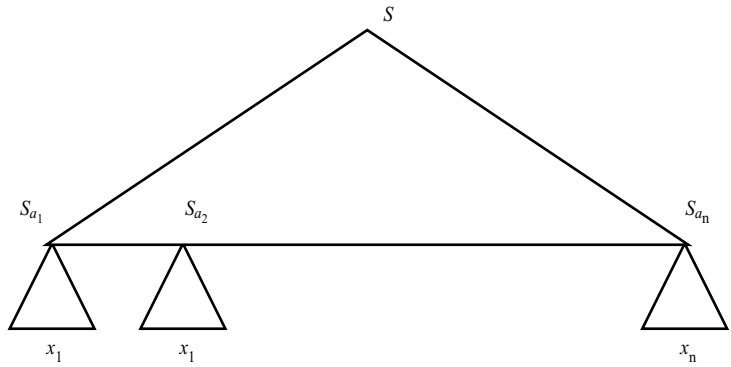


Рис. 7.8. Дерево разбора в G′ *начинается деревом разбора в* G *и заканчивается многими деревьями, соответствующими грамматикам* G_a

Докажем, что описанная конструкция правильна в том смысле, что G' порождает язык $s(L)$. Формально будет доказано следующее утверждение.

- Цепочка w принадлежит $L(G')$ тогда и только тогда, когда w принадлежит $s(L)$.

(*Достаточность*) Пусть w принадлежит $s(L)$. Тогда существует цепочка $x = a_1a_2\ldots a_n$ в L и такие цепочки x_i в $s(a_i)$ при $i = 1, 2, \ldots, n$, для которых $w = x_1x_2\ldots x_n$. Тогда часть G', образованная продукциями из G с подстановками S_a вместо каждого a, порождает цепочку, которая выглядит, как x, но с S_a вместо каждого a, т.е. цепочку $S_{a1}S_{a2}\ldots S_{an}$. Эта часть порождения w представлена верхним треугольником на рис. 7.8.

Продукции каждой G_a являются также продукциями G', поэтому порождение x_i из S_{a_i} есть также порождение в G'. Деревья разбора для этих порождений представлены на рис. 7.8 нижними треугольниками. Поскольку крона этого дерева разбора в G' есть $x_1x_2\ldots x_n = w$, приходим к выводу, что w принадлежит $L(G')$.

(*Необходимость*) Предположим, что w принадлежит $L(G')$. Утверждаем, что дерево разбора для w должно выглядеть, как дерево на рис. 7.8. Причина в том, что переменные каждой из грамматик G и G_a для a из Σ попарно различны. Таким образом, верхушка дерева, начинающаяся переменной S, должна использовать только продукции G до тех пор, пока не будет порожден некоторый символ S_a, а под этим символом могут использоваться только продукции грамматики G_a. В результате, если w имеет дерево разбора T, можно выделить цепочку $a_1a_2\ldots a_n$ в $L(G)$ и цепочки x_i в языках $s(a_i)$, для которых верно следующее.

1. $w = x_1x_2\ldots x_n$.
2. Цепочка $S_{a1}S_{a2}\ldots S_{an}$ является кроной дерева, образованного из T удалением некоторых поддеревьев (см. рис. 7.8).

Но цепочка $x_1x_2\ldots x_n$ принадлежит $s(L)$, поскольку образована подстановкой цепочек x_i вместо каждого из a_i. Таким образом, делаем вывод, что w принадлежит $s(L)$. □

7.3.2. Приложения теоремы о подстановке

С использованием теоремы 7.23 можно обосновать несколько свойств замкнутости, хорошо знакомых нам по регулярным языкам. Перечислим их в следующей теореме.

Теорема 7.24. Контекстно-свободные языки замкнуты относительно следующих операций.

1. Объединение.
2. Конкатенация.
3. Замыкание (*) и транзитивное замыкание ($^+$).
4. Гомоморфизм.

Доказательство. Каждое утверждение требует лишь определения соответствующей подстановки. Каждое из следующих доказательств использует подстановку контекстно-свободных языков в другие, в результате чего по теореме 7.23 порождаются КС-языки.

1. *Объединение*. Пусть L_1 и L_2 — КС-языки. Тогда $L_1 \cup L_2$ является языком $s(L)$, где L — язык $\{1, 2\}$, а s — подстановка, определяемая как $s(1) = L_1$ и $s(2) = L_2$.

2. *Конкатенация*. Пусть L_1 и L_2 — КС-языки. Тогда L_1L_2 представляет собой язык $s(L)$, где L — язык $\{12\}$, а s — такая же подстановка, как и в п. 1.
3. *Замыкание и транзитивное замыкание*. Если L_1 — КС-язык, L — язык $\{1\}^*$, а s — подстановка $s(1) = L_1$, то $L_1^* = s(L)$. Аналогично, если в качестве L взять язык $\{1\}^+$, то $L_1^+ = s(L)$.
4. Пусть L — КС-язык над алфавитом Σ, и h — гомоморфизм на Σ. Пусть s — подстановка, заменяющая каждый символ a из Σ языком, состоящим из единственной цепочки $h(a)$, т.е. $s(a) = \{h(a)\}$ для всех a из Σ. Тогда $h(L) = s(L)$. □

7.3.3. Обращение

КС-языки замкнуты также относительно обращения. Для доказательства этого факта использовать теорему о подстановках нельзя, но существует простая конструкция на основе грамматик.

Теорема 7.25. Если L — КС-язык, то и L^R — КС-язык.

Доказательство. Пусть L — $L(G)$ для некоторой КС-грамматики $G = (V, T, P, S)$. Построим $G^R = (V, T, P^R, S)$, где продукции P^R представляют собой "обращения" продукций из P. Таким образом, если $A \to \alpha$ — продукция G, то $A \to \alpha^R$ — продукция G^R. С помощью индукции по длине порождений в G и G^R нетрудно показать, что $L(G^R) = L^R$. По сути, все выводимые в G^R цепочки являются обращениями цепочек, выводимых в G, и наоборот. Формальное доказательство оставляется в качестве упражнения. □

7.3.4. Пересечение с регулярным языком

КС-языки не замкнуты по пересечению. Это доказывает следующий простой пример.

Пример 7.26. В примере 7.19 было выяснено, что язык

$$L = \{0^n1^n2^n \mid n \geq 1\}$$

не является контекстно-свободным. Однако следующие два — контекстно-свободные.

$$L = \{0^n1^n2^i \mid n \geq 1, i \geq 1\}$$
$$L = \{0^i1^n2^n \mid n \geq 1, i \geq 1\}$$

Первый из них порождается следующей грамматикой.

$$S \to AB$$
$$A \to 0A1 \mid 01$$
$$B \to 2B \mid 2$$

В этой грамматике A порождает все цепочки вида 0^n1^n, а B — все последовательности двоек. Аналогична и грамматика для второго языка.

$$S \to AB$$
$$A \to 0A \mid 0$$
$$B \to 1B2 \mid 12$$

Здесь A порождает все последовательности нулей, а B — цепочки вида 1^n2^n.

Однако $L = L_1 \bigcap L_2$. Чтобы в этом убедиться, заметим, что L_1 требует равных количеств нулей и единиц в цепочках, тогда как L_2 — равных количеств единиц и двоек. Поэтому цепочка из пересечения этих языков должна иметь поровну каждого из символов и, следовательно, принадлежать L.

Если бы КС-языки были замкнуты по пересечению, то мы могли бы доказать ложное утверждение о том, что L — контекстно-свободный язык. Полученное противоречие позволяет заключить, что КС-языки не замкнуты по пересечению. □

Вместе с тем, есть более слабое утверждение о пересечении. КС-языки замкнуты относительно операции “пересечение с регулярным языком”. Формальное утверждение и его доказательство представлены в следующей теореме.

Теорема 7.27. Если L — КС-язык, а R — регулярный язык, то $L \bigcap R$ является КС-языком.

Доказательство. Нам понадобится представление КС-языков с помощью МП-автоматов, а также конечноавтоматное представление регулярных языков. Данное доказательство обобщает доказательство теоремы 4.8, где для получения пересечения регулярных языков “параллельно запускались” два конечных автомата. Здесь конечный автомат запускается параллельно с МП-автоматом, и в результате получается еще один МП-автомат (рис. 7.9).

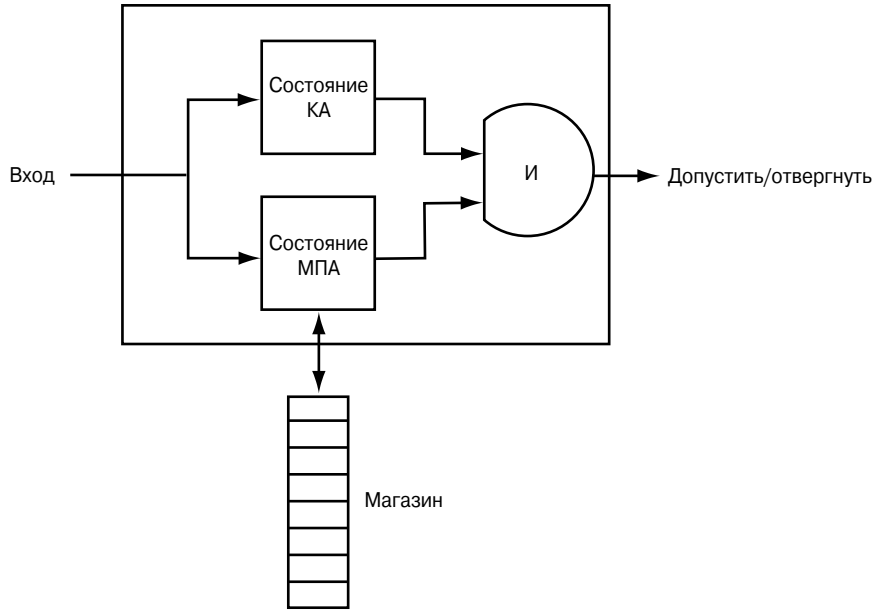


Рис. 7.9. Для создания нового МП-автомата конечный автомат и МП-автомат запускаются параллельно

Формально, пусть

$$P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P) \text{ —}$$

МП-автомат, допускающий L по заключительному состоянию, и пусть

$$A = (Q_A, \Sigma, \delta_A, q_A, F_A) \text{ —}$$

ДКА для R. Построим МП-автомат

$$P' = (Q_P \times Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A),$$

где $\delta((q, p), a, X)$ определяется как множество всех пар $((r, s), \gamma)$, где $s = \hat{\delta}_A(p, a)$ и пара (r, γ) принадлежит $\delta_P(q, A, X)$.

Таким образом, для каждого перехода МП-автомата P мы можем совершить такой же переход в МП-автомате P', дополнительно отслеживая состояние ДКА A во втором компоненте состояния P'. Отметим, что a может быть символом из Σ или ε. В первом случае $\hat{\delta}_A(p, a) = \delta_A(p, a)$, но если $a = \varepsilon$, то $\hat{\delta}_A(p, a) = p$, т.е. A не меняет состояние, когда P совершает ε-переход.

С помощью простой индукции по числу переходов, совершаемых МП-автоматами, нетрудно доказать, что $(q_P, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \gamma)$ тогда и только тогда, когда $((q_P, q_A), w, Z_0) \overset{*}{\underset{P'}{\vdash}} ((q, p), \varepsilon, \gamma)$, где $p = \hat{\delta}_A(p, w)$. Эти индукции оставляются в качестве упражнения. Но (q, p) является допускающим состоянием P' тогда и только тогда, когда q — допускающее состояние P и p — допускающее состояние A. Отсюда заключаем, что P' допускает w тогда и только тогда, когда его допускают P и A вместе, т.е. w принадлежит $L \cap R$. □

Пример 7.28. На рис. 6.6 был определен МП-автомат F, допускающий по заключительному состоянию множество цепочек, которые состоят из i и e. Такие цепочки представляют минимальные нарушения правил, определяющих, в каком порядке слова `if` и `else` могут встречаться в С-программах. Назовем этот язык L. МП-автомат F был определен так:

$$P_F = (\{p, q, r\}, \{i, e\}, \{Z, X_0\}, \delta_F, p, X_0, \{r\}),$$

где δ_F состоит из следующих правил.

1. $\delta_F(p, \varepsilon, X_0) = \{(q, ZX_0)\}$.
2. $\delta_F(q, i, Z) = \{(q, ZZ)\}$.
3. $\delta_F(q, e, Z) = \{(q, \varepsilon)\}$.
4. $\delta_F(q, e, X_0) = \{(r, \varepsilon)\}$.

Теперь определим конечный автомат

$$A = (\{s, t\}, \{i, e\}, \delta_A, s, \{s, t\}),$$

допускающий цепочки языка $\mathbf{i}^*\mathbf{e}^*$, т.е. все цепочки, в которых символы e следуют за i. Назовем этот язык R. Функция переходов δ_A определяется следующими правилами:

а) $\delta_A(s, i) = s$;

б) $\delta_A(s, e) = t$;

в) $\delta_A(t, e) = t$.

Строго говоря, A не является ДКА, как предполагается в теореме 7.27, поскольку в нем отсутствует дьявольское состояние для случая, когда вход i получен в состоянии t. Однако такая же конструкция работает даже для НКА, так как МП-автомат, который строится, может быть недетерминированным. В данном же случае МП-автомат на самом деле детерминирован, хотя и "умирает" на некоторых входных последовательностях.

Построим следующий МП-автомат.

$$P = (\{p, q, r\} \times \{s, t\}, \{i, e\}, \{Z, X_0\}, \delta, (p, s), X_0, \{r\} \times \{s, t\})$$

Переходы δ перечислены ниже и проиндексированы номерами правил для МП-автомата F (числа от 1 до 4) и правил для ДКА A (буквы a, $б$, $в$). Если МП-автомат F совершает ε-переход, правило для A не используется. Отметим, что правила для автомата P строятся "ленивым" способом: правила для состояния строятся только тогда, когда обнаружено, что оно достигается из начального состояния P.

1. $\delta((p, s), \varepsilon, X_0) = \{((q, s), ZX_0)\}$.

2, а. $\delta((q, s), i, Z) = \{((q, s), ZZ)\}$.

3, б. $\delta((q, s), e, Z) = \{((q, t), \varepsilon)\}$.

4. $\delta((q, s), \varepsilon, X_0) = \{(r, s), \varepsilon)\}$. Отметим, что это правило никогда не будет применяться, поскольку невозможно вытолкнуть символ из магазина без e на входе, но как только P видит e, вторым компонентом его состояния становится t.

3, в. $\delta((q, t), e, Z) = \{((q, t), \varepsilon)\}$.

4. $\delta((q, t), \varepsilon, X_0) = \{((r, t), \varepsilon)\}$.

Язык $L \cap R$ представляет собой множество цепочек с некоторым количеством символов i, за которыми записаны символы e (на один больше), т.е. $\{i^n e^{n+1} \mid n \geq 0\}$. Как видим, такие блоки символов i с блоками символов e нарушают правила записи слов `if` и `else` в языке С. Этот язык, очевидно, является КС-языком, порождаемым грамматикой с продукциями $S \to iSe \mid e$.

Заметим, что МП-автомат P допускает язык $L \cap R$. После помещения Z в магазин он заносит новые символы Z в магазин при чтении символов i, оставаясь в состоянии (q, s). Как только на входе появляется e, он переходит в состояние (q, t) и начинает выталкивание из магазина. Он умирает, если видит i до того, как X_0 оказывается на вершине магазина. В последнем же случае он спонтанно переходит в состояние (r, t) и допускает. □

Поскольку КС-языки не замкнуты по пересечению, но замкнуты по пересечению с регулярным языком, то становятся понятными и свойства замкнутости относительно операций разности и дополнения. Перечислим эти свойства в следующей теореме.

Теорема 7.29. Пусть L_1, L_2 и L обозначают КС-языки, а R — регулярный язык. Справедливы следующие утверждения.

1. $L - R$ является контекстно-свободным языком.
2. $\overline{L}$ может не быть КС-языком.
3. $L_1 - L_2$ может не быть КС-языком.

Доказательство. Для п. 1 заметим, что $L - R = L \cap \overline{R}$. Если R регулярно, то по теореме 4.5 регулярно и $\overline{R}$. Тогда по теореме 7.27 $L - R$ — КС-язык.

Для п. 2 предположим, что если L является КС-языком, то $\overline{L}$ — КС-язык. Но поскольку $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$, и КС-языки замкнуты по объединению, получаем, что они замкнуты и по пересечению. Однако это невозможно (см. пример 7.26).

Наконец, докажем п. 3. Очевидно, Σ^* является КС-языком для любого алфавита Σ; нетрудно построить КС-грамматику или МП-автомат для этого регулярного языка. Таким образом, если бы язык $L_1 - L_2$ был контекстно-свободным для любых КС-языков L_1 и L_2, то и $\Sigma^* - L$ должен быть КС-языком, если L — КС-язык. Однако $\Sigma^* - L = \overline{L}$ при соответствующем алфавите Σ. Полученное противоречие к утверждению 2 доказывает, что $L_1 - L_2$ не обязательно является КС-языком. □

7.3.5. Обратный гомоморфизм

Вспомним операцию “обратного гомоморфизма” из раздела 4.2.4. Если h — гомоморфизм, а L — произвольный язык, то $h^{-1}(L)$ представляет собой множество таких цепочек w, для которых $h(w) \in L$. Доказательство того, что регулярные языки замкнуты относительно обратного гомоморфизма, представлено на рис. 4.6. Там показано, как строится конечный автомат, обрабатывающий свои входные символы a путем применения к ним гомоморфизма h и имитации другого конечного автомата на цепочках $h(a)$.

Замкнутость относительно обратного гомоморфизма можно доказать подобным путем, используя МП-автоматы вместо конечных автоматов. Однако при использовании МП-автоматов возникает проблема, которой не было с конечными автоматами. Действие конечного автомата при обработке входной цепочки заключается в изменении состояния, и это выглядит так же, как переход по одиночному входному символу.

Однако для МП-автоматов последовательность переходов может быть не похожа на переход по одному входному символу. В частности, за n переходов МП-автомат может вытолкнуть n символов из своего магазина, тогда как при одном переходе выталкивается только один. Таким образом, конструкция МП-автоматов, аналогичная представленной на рис. 4.6, будет существенно сложнее; она изображена эскизно на рис. 7.10. Дополни-

тельная ключевая идея заключается в том, что, когда прочитан вход a, цепочка $h(a)$ помещается в "буфер". Символы $h(a)$ используются по одному и подаются на вход имитируемому МП-автомату. Когда буфер опустошается, основной МП-автомат читает свой следующий входной символ и применяет гомоморфизм к нему. Эта конструкция уточняется в следующей теореме.

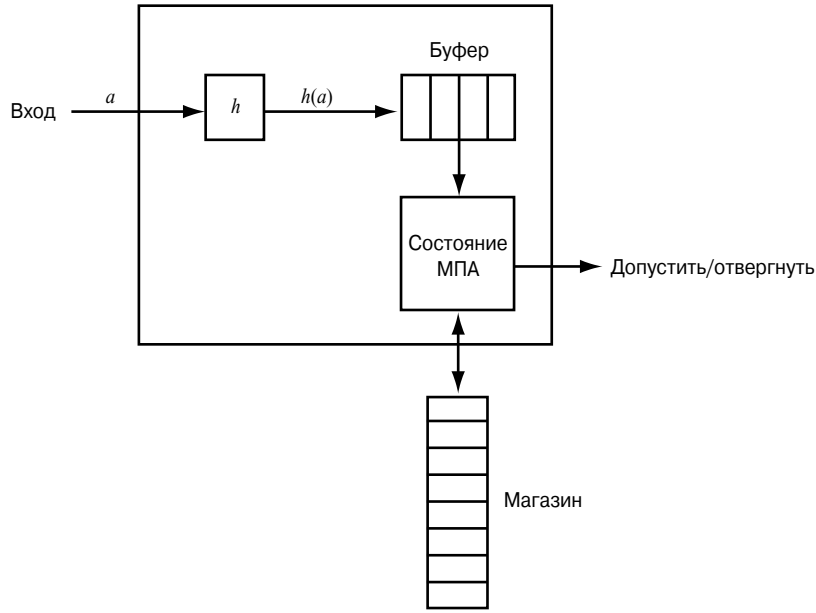


Рис. 7.10. Построение МП-автомата, допускающего обратный гомоморфизм того, что допускает данный МП-автомат

Теорема 7.30. Пусть L — КС-язык, h — гомоморфизм. Тогда $h^{-1}(L)$ является КС-языком.

Доказательство. Пусть h применяется к символам из алфавита Σ и порождает цепочки из T^*. Предположим также, что L — язык в алфавите T, допускаемый по заключительному состоянию МП-автоматом $P = (Q, T, \Gamma, \delta, q_0, Z_0, F)$. Построим следующий новый МП-автомат P':

$$P' = (Q', \Sigma, \Gamma, \delta', (q_0, \varepsilon), Z_0, F \times \{\varepsilon\}) \tag{7.1}$$

Обозначения в определении P' имеют следующий смысл.

1. Q' есть множество пар (q, x), где q — состояние из Q, а x — суффикс (не обязательно собственный) некоторой цепочки $h(a)$ для символа a из Σ. Таким образом, первый компонент состояния P' является состоянием P, а второй — компонентом буфера. Предполагается, что буфер периодически загружается цепочкой $h(a)$, а затем сокращается с головы по мере чтения его символов, которые подаются на вход имитируемому МП-автомату P.

2. δ' определяется следующими правилами:

   а) $\delta'((q, \varepsilon), a, X) = \{((q, h(a)), X)\}$ для всех символов a из Σ, всех состояний q из Q и магазинных символов X из Γ. Отметим, что здесь a не может быть ε. Когда буфер пуст, P' может прочитать свой следующий входной символ a и поместить $h(a)$ в буфер;

   б) если $\delta(q, b, X)$ содержит (p, γ), где $b \in T$ или $b = \varepsilon$, то $\delta'((q, bx), \varepsilon, X)$ содержит $((p, x), \gamma)$. Таким образом, P' всегда имеет возможность имитации перехода P, используя голову буфера. Если $b \in T$, то буфер должен быть непустым, но если $b = \varepsilon$, то буфер может быть пустым.

3. Отметим, что в соответствии с определением (7.1) начальным состоянием P' является (q_0, ε), т.е. P' стартует в начальном состоянии P с пустым буфером.

4. Аналогично, допускающими состояниями P' являются такие состояния (q, ε), у которых q — допускающее состояние P.

Следующее утверждение характеризует взаимосвязь P' и P.

- $(q_0, h(w), Z_0) \overset{*}{\underset{P}{\vdash}} (p, \varepsilon, \gamma)$ тогда и только тогда, когда $((q_0, \varepsilon), w, Z_0) \overset{*}{\underset{P'}{\vdash}} ((p, \varepsilon), \varepsilon, \gamma)$.

Доказательство в обоих направлениях проводится индукцией по числу переходов, совершаемых автоматами. При доказательстве достаточности заметим, что если буфер P' не пуст, то он не может читать свой следующий входной символ, а должен имитировать P до тех пор, пока буфер не опустошится (хотя когда буфер пуст, он все еще может имитировать P). Детали оставляются в качестве упражнения.

Приняв указанную взаимосвязь P и P', заметим, что вследствие способа, которым определены допускающие состояния P', P допускает $h(w)$ тогда и только тогда, когда P' допускает w. Таким образом, $L(P') = h^{-1}(L(P))$. □

7.3.6. Упражнения к разделу 7.3

7.3.1. Докажите, что КС-языки замкнуты относительно следующих операций:

а) (*) *Init*, определенная в упражнении 4.2.6, *в*. *Указание*. Начните с НФХ-грамматики для языка L;

б) (*!) операция L/a, определенная в упражнении 4.2.2. *Указание*. Начните с НФХ-грамматики для языка L;

в) (!!) *Cycle*, определенная в упражнении 4.2.11. *Указание*. Используйте конструкцию, основанную на МП-автомате.

7.3.2. Рассмотрим следующие два языка:

$L_1 = \{a^n b^{2n} c^m \mid n, m \geq 0\}$
$L_2 = \{a^n b^m c^{2m} \mid n, m \geq 0\}$

а) покажите, что каждый из них является контекстно-свободным, построив для них КС-грамматики;

б) (!) укажите, является ли $L_1 \cap L_2$ КС-языком. Ответ обоснуйте.

7.3.3. (!!) Покажите, что КС-языки *не замкнуты* относительно следующих операций:

а) (*) *Min*, определенная в упражнении 4.2.6, *а*;

б) *Max*, определенная в упражнении 4.2.6, *б*;

в) *Half*, определенная в упражнении 4.2.8;

г) *Alt*, определенная в упражнении 4.2.7.

7.3.4. *Shuffle* (*Перемешивание*) двух цепочек w и x является множеством всех цепочек, которые можно получить путем произвольного чередования позиций w и x. Точнее, *shuffle*(w, x) есть множество цепочек z, обладающих следующими свойствами.

1. Каждая позиция z может быть назначена w или x, но не обеим сразу.
2. Позиции z, назначенные w, при чтении слева направо образуют w.
3. Позиции z, назначенные x, при чтении слева направо образуют x.

Например, если $w = 01$ и $x = 110$, то *shuffle*(01, 110) есть множество цепочек {01110, 01101, 10110, 10101, 11010, 11001}. Для иллюстрации рассмотрим цепочку 10110. Ее вторая и пятая позиции назначены 01, а первая, третья и четвертая — 110. Цепочка 01110 может быть построена тремя способами: позиция 1 и одна из 2, 3, 4 назначается 01, а оставшиеся три в каждом случае — 110.

Перемешивание языков, *shuffle*(L_1, L_2), определяется как объединение *shuffle*(w, x) по всем парам цепочек, w из L_1 и x из L_2:

а) постройте *shuffle*(00, 111);

б) (*) укажите, что представляет собой *shuffle*(L_1, L_2), если $L_1 = L(\mathbf{0}^*)$ и $L_2 = \{0^n1^n \mid n \geq 0\}$;

в) (*!) докажите, что если L_1 и L_2 — регулярные языки, то и *shuffle*(L_1, L_2) регулярен. *Указание*. Начните с конечных автоматов для L_1 и L_2;

г) (!) докажите, что если L является КС-языком, а R — регулярным языком, то *shuffle*(L, R) — КС-язык. *Указание*. Начните с МП-автомата для L и ДКА для R;

д) (!!) приведите контрпример, показывающий, что если L_1 и L_2 — КС-языки, то *shuffle*(L_1, L_2) может не быть КС-языком.

7.3.5. (*!!) Цепочка y называется *перестановкой* цепочки x, если символы y можно переупорядочить и получить x. Например, перестановками цепочки $x = 011$ являются 110, 101 и 011. Если L — язык, то *perm*(L) — это множество цепочек, являющихся перестановками цепочек из L. Например, если $L = \{0^n1^n \mid n \geq 0\}$, то *perm*($L$) представляет собой множество цепочек, в которых поровну символов 0 и 1:

а) приведите пример регулярного языка L над алфавитом $\{0, 1\}$, для которого $perm(L)$ нерегулярен. Ответ обоснуйте. *Указание*. Попытайтесь найти регулярный язык, перестановками цепочек которого являются все цепочки с одинаковыми количествами 0 и 1;

б) приведите пример регулярного языка L в алфавите $\{0, 1, 2\}$, для которого $perm(L)$ не является КС-языком;

в) докажите, что для каждого регулярного языка L в двухсимвольном алфавите $perm(L)$ является КС-языком.

7.3.6. Приведите формальное доказательство теоремы 7.25 о том, что КС-языки замкнуты относительно обращения.

7.3.7. Дополните доказательство теоремы 7.27, показав, что

$(q_P, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \gamma)$

тогда и только тогда, когда $((q_P, q_A), w, Z_0) \overset{*}{\underset{P'}{\vdash}} ((q, p), \varepsilon, \gamma)$ и $p = \hat{\delta}(p_A, w)$.

7.4. Свойства разрешимости КС-языков

Теперь рассмотрим, на какие вопросы о контекстно-свободных языках можно дать ответ. По аналогии с разделом 4.3, где речь шла о свойствах разрешимости регулярных языков, все начинается с представления КС-языка — с помощью грамматики или МП-автомата. Поскольку из раздела 6.3 нам известно о взаимных преобразованиях грамматик и МП-автоматов, можно предполагать, что доступны оба представления, и в каждом конкретном случае будем использовать более удобное.

Мы обнаружим, что разрешимых вопросов, связанных с КС-языками, совсем немного. Основное, что можно сделать, — это проверить, пуст ли язык, и принадлежит ли данная цепочка языку. Этот раздел завершается кратким обсуждением проблем, которые являются, как будет показано в главе 9, “неразрешимыми”, т.е. не имеющими алгоритма разрешения. Начнем этот раздел с некоторых замечаний о сложности преобразований между грамматиками и МП-автоматами, задающими язык. Эти расчеты важны в любом вопросе об эффективности разрешения свойств КС-языков по данному их представлению.

7.4.1. Сложность взаимных преобразований КС-грамматик и МП-автоматов

Прежде чем приступать к алгоритмам разрешения вопросов о КС-языках, рассмотрим сложность преобразования одного представления в другое. Время выполнения преобразования является составной частью стоимости алгоритма разрешения в тех случаях, когда алгоритм построен для одной формы представления, а язык дан в другой.

В дальнейшем n будет обозначать длину представления МП-автомата или КС-грамматики. Использование этого параметра в качестве представления размера грамматики или автомата является “грубым”, в том смысле, что некоторые алгоритмы имеют время выполнения, которое описывается точнее в терминах других параметров, например, число переменных в грамматике или сумма длин магазинных цепочек, встречающихся в функции переходов МП-автомата.

Однако мера общей длины достаточна для решения наиболее важных вопросов: является ли алгоритм линейным относительно длины входа (т.е. требует ли он времени, чуть большего, чем нужно для чтения входа), экспоненциальным (т.е. преобразование выполнимо только для примеров малого размера) или нелинейным полиномиальным (т.е. алгоритм можно выполнить даже для больших примеров, но время будет значительным).

Следующие преобразования линейны, как мы увидим далее, относительно размера входных данных.

1. Преобразование КС-грамматики в МП-автомат по алгоритму из теоремы 6.13.
2. Преобразование МП-автомата, допускающего по заключительному состоянию, в МП-автомат, допускающий по пустому магазину, с помощью конструкции из теоремы 6.11.
3. Преобразование МП-автомата, допускающего по пустому магазину, в МП-автомат, допускающий по заключительному состоянию, с использованием конструкции из теоремы 6.9.

С другой стороны, время преобразования МП-автомата в грамматику (теорема 6.14) существенно больше. Заметим, что n, общая длина входа, гарантированно является верхней границей числа состояний или магазинных символов, поэтому переменных вида $[pXq]$, построенных для грамматики, может быть не более n^3. Однако время выполнения преобразования может быть экспоненциальным, если у МП-автомата есть переход, помещающий большое число символов в магазин. Отметим, что одно правило может поместить в магазин почти n символов.

Если мы вспомним построение продукций грамматики по правилу вида “$\delta(q, a, X)$ содержит $(r_0, Y_1Y_2\ldots Y_k)$”, то заметим, что оно порождает набор продукций вида $[qXr_k] \rightarrow a[r_0Y_1r_1][r_1Y_2r_2]\ldots[r_{k-1}Y_kr_k]$ для всех последовательностей состояний $r_1, r_2, \ldots, r_k$. Поскольку k может быть близко к n, общее число продукций возрастает как n^n. Такое построение невозможно довести до конца для МП-автомата разумного размера, даже если он имеет всего одну цепочку, записываемую в магазин.

К счастью, этого наихудшего случая всегда можно избежать. Как предлагалось в упражнении 6.2.8, помещение длинной цепочки в магазин можно разбить на последовательность из не более, чем n шагов, на каждом из которых в магазин помещается всего один символ. Таким образом, если $\delta(q, a, X)$ содержит $(r_0, Y_1Y_2\ldots Y_k)$, можно ввести новые

состояния $p_2, p_3, \ldots, p_{k-1}$. Затем изменим $(r_0, Y_1Y_2\ldots Y_k)$ в $\delta(q, a, X)$ на $(p_{k-1}, Y_{k-1}Y_k)$ и введем новые переходы

$$\delta(p_{k-1}, \varepsilon, Y_{k-1}) = \{(p_{k-2}, Y_{k-2}Y_{k-1})\},\ \delta(p_{k-2}, \varepsilon, Y_{k-2}) = \{(p_{k-3}, Y_{k-3}Y_{k-2})\}$$

и так далее до $\delta(p_2, \varepsilon, Y_2) = \{(r_0, Y_1Y_2)\}$.

Теперь в любом переходе не более двух магазинных символов. При этом добавлено не более n новых состояний, и общая длина всех правил перехода δ выросла не более, чем в константу раз, т.е. осталась $O(n)$. Существует $O(n)$ правил перехода, и каждое порождает $O(n^2)$ продукций, поскольку в продукциях, порожденных каждым правилом, должны быть выбраны всего два состояния. Таким образом, грамматика имеет длину $O(n^3)$ и может быть построена за кубическое время. Проведенный неформальный анализ резюмируется в следующей теореме.

Теорема 7.31. Существует алгоритм сложности $O(n^3)$, который по МП-автомату P строит КС-грамматику длиной не более $O(n^3)$. Эта грамматика порождает язык, допускаемый P по пустому магазину. В дополнение, можно построить грамматику, которая порождает язык, допускаемый P по заключительному состоянию. □

7.4.2. Временная сложность преобразования к нормальной форме Хомского

Алгоритмы могут зависеть от первичного преобразования в нормальную форму Хомского, поэтому посмотрим на время выполнения различных алгоритмов, использованных для приведения произвольной грамматики к НФХ. Большинство шагов сохраняют, с точностью до константного сомножителя, длину описания грамматики, т.е. по грамматике длиной n они строят другую длиной $O(n)$. “Хорошие” (с точки зрения затрат времени) преобразования перечислены в следующем списке.

1. С использованием подходящего алгоритма (см. раздел 7.4.3) определение достижимых и порождающих символов грамматики может быть выполнено за линейное время, $O(n)$. Удаление получившихся бесполезных символов требует $O(n)$ времени и не увеличивает размер грамматики.
2. Построение цепных пар и удаление цепных продукций, как в разделе 7.1.4, требует времени $O(n^2)$, и получаемая грамматика имеет размер $O(n^2)$.
3. Замена терминалов переменными в телах продукций, как в разделе 7.1.5 (нормальная форма Хомского), требует времени $O(n)$ и приводит к грамматике длиной $O(n)$.
4. Разделение тел продукций длины 3 и более на тела длины 2 (раздел 7.1.5) также требует времени $O(n)$ и приводит к грамматике длиной $O(n)$.

“Плохой” является конструкция из раздела 7.1.3, где удаляются ε-продукции. По телу продукции длиной k можно построить $2^k - 1$ продукций новой грамматики. Поскольку k

может быть пропорционально n, эта часть построения может занимать $O(2^n)$ времени и приводить к грамматике длиной $O(2^n)$.

Во избежание этого экспоненциального взрыва достаточно ограничить длины тел продукций. К каждому телу продукции можно применить прием из раздела 7.1.5, но только если в теле нет терминалов. Таким образом, в качестве предварительного шага перед удалением ε-продукций рекомендуется разделить все продукции с длинными телами на последовательность продукций с телами длины 2. Этот шаг требует времени $O(n)$ и увеличивает грамматику только линейно. Конструкция из раздела 7.1.3 для удаления ε-продукций будет работать с телами длиной не более 2 так, что время выполнения будет $O(n)$ и полученная грамматика будет длиной $O(n)$.

С такой модификацией общего построения НФХ единственным нелинейным шагом будет удаление цепных продукций. Поскольку этот шаг требует $O(n^2)$ времени, можно заключить следующее.

Теорема 7.32. По грамматике G длиной n можно найти грамматику в нормальной форме Хомского, эквивалентную G, за время $O(n^2)$; полученная грамматика будет иметь длину $O(n^2)$. □

7.4.3. Проверка пустоты КС-языков

Алгоритм проверки пустоты КС-языка L нам уже знаком. Чтобы определить, является ли стартовый символ S данной грамматики G для языка L порождающим, можно использовать алгоритм из раздела 7.1.2. L пуст тогда и только тогда, когда S не является порождающим.

Поскольку эта проверка весьма важна, рассмотрим детальнее, сколько времени требуется для поиска всех порождающих символов грамматики G. Пусть G имеет длину n. Тогда у нее может быть порядка n переменных, и каждый проход индуктивного обнаружения порождающих переменных может занимать $O(n)$ времени для проверки всех продукций G. Если на каждом проходе обнаруживается только одна новая порождающая переменная, то может понадобиться $O(n)$ проходов. Таким образом, простая реализация проверки на порождающие символы требует $O(n^2)$ времени, т.е. является квадратичной.

Однако существует более аккуратный алгоритм, который заранее устанавливает структуру данных для того, чтобы обнаружить порождающие символы всего за $O(n)$ времени. Структура данных (рис. 7.11) начинает с массива, индексированного переменными, как показано слева, который говорит, установлено ли, что переменная является порождающей. Массив на рис. 7.11 показывает, что переменная B уже обнаружена как порождающая, но о переменной A это еще неизвестно. В конце алгоритма каждая отметка “?” превращается в “нет”, поскольку каждая переменная, не обнаруженная алгоритмом как порождающая, на самом деле является непорождающей.

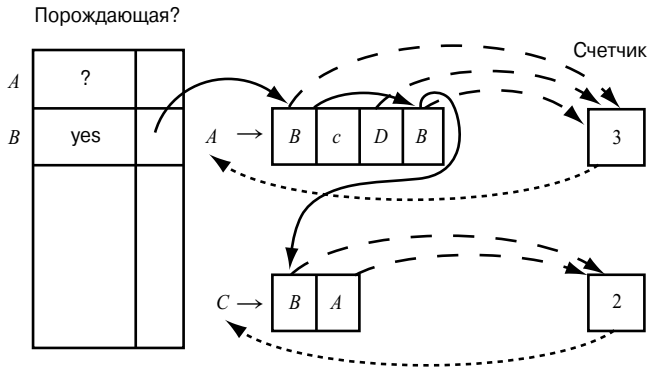


Рис. 7.11. Структуры данных для линейной проверки пустоты

Для продукций предварительно устанавливается несколько видов полезных ссылок. Во-первых, для каждой переменной заводится список всех возможных позиций, в которых эта переменная встречается. Например, список для переменной B представлен сплошными линиями. Во-вторых, для каждой продукции ведется счетчик числа позиций, содержащих переменные, способность которых породить терминальную цепочку еще не учтена. Пунктирные линии представляют связи, ведущие от продукций к их счетчикам. Счетчики, показанные на рис. 7.11, предполагают, что ни одна из переменных в телах продукций еще не учитывалась, хотя уже и установлено, что B — порождающая.

Предположим, мы уже обнаружили, что B — порождающая. Мы спускаемся по списку позиций в телах, содержащих B. Для каждой такой позиции счетчик ее продукции уменьшаем на 1; позиций, которые нужны для заключения, что переменная в голове продукции тоже порождающая, остается на одну меньше.

Если счетчик достигает 0, то понятно, что переменная в голове продукции является порождающей. Связь, представленная точечными линиями, приводит к переменной, и эту переменную можно поместить в очередь переменных, о которых еще неизвестно, порождают ли они (переменная B уже исследована). Эта очередь не показана.

Обоснуем, что этот алгоритм требует $O(n)$ времени. Важными являются следующие утверждения.

- Поскольку грамматика размера n имеет не более n переменных, создание и инициализация массива требует времени $O(n)$.
- Есть не более n продукций, и их общая длина не превосходит n, поэтому инициализация связей и счетчиков, представленных на рис. 7.11, может быть выполнена за время $O(n)$.
- Когда обнаруживается, что счетчик продукции получил значение 0, т.е. все позиции в ее теле являются порождающими, вся проделанная работа может быть разделена на следующие два вида.

1. Работа, выполненная для продукции: обнаружение, что счетчик обнулен, поиск переменной, скажем, A, в голове продукции, проверка, установлено ли, что эта переменная является порождающей, и помещение ее в очередь, если это не так. Все эти шаги требуют $O(1)$ времени для каждой продукции, поэтому вся такая работа в целом требует $O(n)$ времени.
2. Работа, выполненная при посещении позиций в телах продукций, имеющих переменную A в голове. Эта работа пропорциональна числу позиций с переменной A. Следовательно, совокупная работа, выполненная со всеми порождающими переменными, пропорциональна сумме длин тел продукций, а это $O(n)$.

Отсюда делаем вывод, что общая работа, выполненная этим алгоритмом, есть $O(n)$.

Другие способы использования линейной проверки пустоты

Структура данных и счетчики, примененные в разделе 7.4.3 для проверки, является ли переменная порождающей, могут использоваться для обеспечения линейности времени некоторых других проверок из раздела 7.1. Назовем два важных примера.

1. Какие символы достижимы?
2. Какие символы являются ε-порождающими?

7.4.4. Проверка принадлежности КС-языку

Проблема принадлежности цепочки w КС-языку L также разрешима. Есть несколько неэффективных способов такой проверки; они требуют времени, экспоненциального относительно $|w|$, в предположении, что язык L представлен заданной грамматикой или МП-автоматом, и размер представления считается константой, не зависящей от w. Например, начнем с преобразования какого-либо данного нам представления в НФХ-грамматику. Поскольку дерево разбора в такой грамматике является бинарным, при длине n слова w в дереве будет ровно $2n - 1$ узлов, отмеченных переменными. Несложное индуктивное доказательство этого факта оставляется в качестве упражнения. Количество возможных деревьев и разметок их узлов, таким образом, "всего лишь" экспоненциально относительно n, поэтому в принципе их можно перечислить, и проверить, имеет ли какое-нибудь из деревьев крону w.

Существует гораздо более эффективный метод, основанный на идее "динамического программирования" и известный также, как "алгоритм заполнения таблицы" или "табуляция". Данный алгоритм известен как *CYK-алгоритм*[3] (*алгоритм Кока-Янгера-Касами*). Он начинает с НФХ-грамматики $G = (V, T, P, S)$ для языка L. На вход алгоритма подается цепочка $w = a_1a_2\ldots a_n$ из T^*. За время $O(n^3)$ алгоритм строит таблицу, которая говорит, принадлежит ли w языку L. Отметим, что при вычислении этого времени сама

[3] Он назван по фамилиям трех авторов (J. Cocke, D. Younger и T. Kasami), независимо пришедших к одной и той же, по сути, идее.

по себе грамматика рассматривается фиксированной, и ее размер вносит лишь константный сомножитель в оценку времени, измеряемого в терминах длины цепочки, проверяемой на принадлежность L.

В CYK-алгоритме строится треугольная таблица (рис. 7.12). Горизонтальная ось соответствует позициям цепочки $w = a_1a_2\ldots a_n$, имеющей в нашем примере длину 5. Содержимое клетки, или вход таблицы X_{ij}, есть множество таких переменных A, для которых $A \overset{*}{\Rightarrow} a_ia_{i+1}\ldots a_j$. Заметим, в частности, что нас интересует, принадлежит ли S множеству X_{1n}, поскольку это то же самое, что $S \overset{*}{\Rightarrow} w$, т.е. $w \in L$.

Таблица заполняется построчно снизу вверх. Отметим, что каждая строка соответствует определенной длине подцепочек; нижняя — подцепочкам длины 1, вторая снизу — подцепочкам длины 2 и так далее до верхней строки, соответствующей одной подцепочке длиной n, т.е. w. Ниже обсуждается метод, с помощью которого вычисление одного входа требует времени $O(n)$. Поскольку всего входов $n(n + 1)/2$, весь процесс построения таблицы занимает $O(n^3)$ времени. Алгоритм вычисления X_{ij} таков.

X_{15}				
X_{14}	X_{25}			
X_{13}	X_{24}	X_{35}		
X_{12}	X_{23}	X_{34}	X_{45}	
X_{11}	X_{22}	X_{33}	X_{44}	X_{55}
a_1	a_2	a_3	a_4	a_5

Рис. 7.12. Таблица, построенная алгоритмом Кока-Янгера-Касами

Базис. Вычисляем первую строку следующим образом. Поскольку цепочка, которая начинается и заканчивается в позиции i, представляет собой просто терминал a_i, а грамматика находится в НФХ, единственный способ породить a_i заключается в использовании продукции вида $A \to a_i$ грамматики G. Итак, X_{ii} является множеством переменных A, для которых $A \to a_i$ — продукция G.

Индукция. Пусть нужно вычислить X_{ij} в $(j - i + 1)$-й строке, и все множества X в нижних строках уже вычислены, т.е. известны для всех подцепочек, более коротких, чем $a_ia_{i+1}\ldots a_j$, и в частности, для всех собственных префиксов и суффиксов этой цепочки. Можно предполагать, что $j - i > 0$, поскольку случай $j = i$ рассмотрен в базисе. Поэтому любое порождение $A \overset{*}{\Rightarrow} a_ia_{i+1}\ldots a_j$ должно начинаться шагом $A \Longrightarrow BC$. Тогда B порождает некоторый префикс строки $a_ia_{i+1}\ldots a_j$, скажем, $B \overset{*}{\Rightarrow} a_ia_{i+1}\ldots a_k$ для некоторого $k < j$. Кроме того, C порождает остаток $a_ia_{i+1}\ldots a_j$, т.е. $C \overset{*}{\Rightarrow} a_{k+1}a_{k+2}\ldots a_j$.

Приходим к выводу, что для того, чтобы A попало в X_{ij}, нужно найти переменные B и C и целое k, при которых справедливы следующие условия.

1. $i \le k < j$.
2. B принадлежит X_{ik}.
3. C принадлежит $X_{k+1,\,j}$.
4. $A \to BC$ — продукция в G.

Поиск таких переменных A требует обработки не более n пар вычисленных ранее множеств: $(X_{ii}, X_{i+1,\,j})$, $(X_{i,\,i+1}, X_{i+2,\,j})$ и т.д. до $(X_{i,\,j-1}, X_{jj})$. Таким образом, мы поднимаемся по колонке, расположенной под X_{ij}, и одновременно спускаемся по диагонали (рис. 7.13).

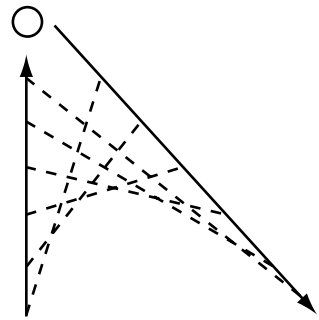


Рис. 7.13. Вычисление X_{ij} *требует совместной обработки столбца под* X_{ij} *и диагонали справа от него*

Теорема 7.33. Вышеописанный алгоритм корректно вычисляет X_{ij} для всех i и j. Таким образом, $w \in L(G)$ тогда и только тогда, когда $S \in X_{1n}$. Кроме того, время выполнения алгоритма есть $O(n^3)$.

Доказательство. Представив базис и индуктивную часть алгоритма, мы объяснили, почему алгоритм находит корректные множества переменных. Рассмотрим время выполнения. Заметим, что нужно вычислить $O(n^2)$ элементов таблицы, и каждое вычисление вовлекает сравнение и вычисление не более, чем n пар элементов. Важно помнить, что, хотя в каждом множестве X_{ij} может быть много переменных, грамматика G зафиксирована и число ее переменных не зависит от n — длины проверяемой цепочки w. Таким образом, время сравнения элементов X_{ik} и $X_{k+1,\,j}$ и поиска переменных, входящих в X_{ij}, есть $O(1)$. Поскольку для каждого X_{ij} возможно не более n таких пар, общее время составляет $O(n^3)$. □

Пример 7.34. Рассмотрим следующие продукции грамматики G в НФХ.

$S \to AB \mid BC$
$A \to BA \mid a$
$B \to CC \mid b$
$C \to AB \mid a$

Проверим, принадлежит ли цепочка *baaba* языку $L(G)$. На рис. 7.14 показана таблица, заполненная для этой строки.

Для построения первой (нижней) строки используется базисное правило. Нужно рассмотреть лишь переменные с телом продукции *a* (это *A* и *C*) и телом *b* (это *B*). Таким образом, $X_{11} = X_{44} = \{B\}$, $X_{22} = X_{33} = X_{55} = \{A, C\}$.

Во второй строке показаны значения X_{12}, X_{23}, X_{34} и X_{45}. Рассмотрим, например, как вычисляется X_{12}. Цепочку *ba*, занимающую позиции 1 и 2, можно разбить на непустые подцепочки единственным способом. Первая должна занимать позицию 1, вторая — позицию 2. Для того чтобы переменная порождала *ba*, она должна иметь продукцию с телом, первая переменная которого принадлежит $X_{11} = \{B\}$ (т.е. порождает *b*), а вторая — $X_{22} = \{A, C\}$ (т.е. порождает *a*). Таким телом может быть только *BA* или *BC*. Просмотрев грамматику, находим, что с такими телами там есть только продукции $A \to BA$ и $S \to BC$. Таким образом, две головы, *A* и *S*, образуют X_{12}.

{*S*, *A*, *C*}				
-	{*S*, *A*, *C*}			
-	{*B*}	{*B*}		
{*S*, *A*}	{*B*}	{*S*, *C*}	{*S*, *A*}	
{*B*}	{*A*, *C*}	{*A*, *C*}	{*B*}	{*A*, *C*}
b	*a*	*a*	*b*	*a*

Рис. 7.14. Таблица для цепочки baaba*, построенная алгоритмом Кока-Янгера-Касами*

В качестве более сложного примера рассмотрим вычисление X_{24}. Цепочку *aab* в позициях с 2 по 4 можно разбить, заканчивая первую подцепочку в 2 или в 3, т.е. в определении X_{24} можно выбрать $k = 2$ или $k = 3$. Таким образом, нужно рассмотреть все тела в $X_{22}X_{34} \cup X_{23}X_{44}$. Этим множеством цепочек является $\{A, C\}\{S, C\} \cup \{B\}\{B\} = \{AS, AC, CS, CC, BB\}$. Из пяти цепочек этого множества только *CC* является телом; его голова — *B*. Таким образом, $X_{24} = \{B\}$. □

7.4.5. Обзор неразрешимых проблем КС-языков

В следующих главах излагается замечательная теория, позволяющая доказать формально, что существуют проблемы, которые нельзя разрешить никаким алгоритмом, выполняемым на компьютере. Используем ее для того, чтобы показать, что многие элементарные вопросы о грамматиках и КС-языках не имеют алгоритма решения; они называются “неразрешимыми проблемами”. Сейчас же ограничимся следующим списком наиболее значительных неразрешимых вопросов о контекстно-свободных грамматиках и языках.

1. Неоднозначна ли данная КС-грамматика G?
2. Является ли данный КС-язык существенно неоднозначным?
3. Пусто ли пересечение двух КС-языков?
4. Равны ли два данных КС-языка?
5. Равен ли Σ^* данный КС-язык, где Σ — алфавит этого языка?

Отметим, что вопрос 1 о неоднозначности отличается от остальных тем, что это вопрос о грамматике, а не о языке. Все остальные вопросы предполагают, что язык представлен грамматикой или МП-автоматом, но это все равно вопросы о языке (или языках). Например, в противоположность вопросу 1 вопрос 2 требует по данной грамматике G (или МП-автомату) определить, существует ли некоторая эквивалентная ей однозначная грамматика G'. Если G сама по себе однозначна, то ответом, безусловно, будет "да", но если G неоднозначна, то для языка грамматики G может существовать другая грамматика G', которая однозначна, как было с грамматиками выражений в примере 5.27.

7.4.6. Упражнения к разделу 7.4

7.4.1. Постройте алгоритмы разрешения следующих проблем:

а) (*) конечен ли язык $L(G)$ данной грамматики G? *Указание*. Используйте лемму о накачке;

б) (!) определить, содержит ли язык $L(G)$ данной грамматики G не менее 100 цепочек;

в) (!!) по данной грамматике G и ее переменной A определить, существует ли выводимая цепочка, которая начинается символом A. *Указание*. Напомним, что переменная A может впервые появиться в середине некоторой выводимой цепочки, а затем все символы слева от нее могут породить ε.

7.4.2. Используйте технику, описанную в разделе 7.4.3, для построения линейных по времени алгоритмов разрешения следующих вопросов о КС-грамматиках.

1. Какие символы встречаются в выводимых цепочках?
2. Какие символы являются ε-порождающими?

7.4.3. Примените CYK-алгоритм к грамматике G из примера 7.34, чтобы определить, принадлежат ли $L(G)$ следующие цепочки:

а) (*) *ababa*;

б) *baaab*;

в) *aabab*.

7.4.4. (*) Покажите, что для любой НФХ-грамматики все деревья разбора цепочек длиной n имеют $2n - 1$ внутренних узлов (отмеченных переменными).

7.4.5. (!) Измените CYK-алгоритм так, чтобы он сообщал, сколько различных деревьев вывода у данной цепочки, а не просто, принадлежит ли она языку грамматики.

Резюме

- *Удаление бесполезных символов*. Переменную можно удалить из КС-грамматики, если она не порождает ни одной терминальной цепочки или не встречается в цепочках, выводимых из стартового символа. Для корректного удаления таких бесполезных символов нужно сначала проверить, порождает ли каждая переменная терминальную цепочку, и удалить те, которые не порождают. Только после этого удаляются переменные, которые не выводятся из стартового символа.
- *Удаление цепных и ε-продукций*. По данной КС-грамматике G можно найти еще одну КС-грамматику, которая порождает тот же язык, за исключением цепочки ε, но не содержит цепных продукций (с единственной переменной в качестве тела) и ε-продукций (с телом ε).
- *Нормальная форма Хомского*. По данной КС-грамматике G можно найти еще одну КС-грамматику, которая порождает тот же язык, за исключением цепочки ε, и находится в нормальной форме Хомского: нет бесполезных символов, и тело каждой продукции состоит либо из двух переменных, либо из одного терминала.
- *Лемма о накачке*. В любой достаточно длинной цепочке КС-языка можно найти короткую подцепочку, два конца которой можно синхронно "накачивать", т.е повторять произвольное число раз. Хотя бы одна из накачиваемых цепочек не равна ε. Эта лемма, а также ее более мощная версия, которая называется леммой Огдена и приводится в упражнении 7.2.3, дают возможность доказывать, что многие языки не являются контекстно-свободными.
- *Операции, сохраняющие контекстно-свободные языки*. КС-языки замкнуты относительно подстановки, объединения, конкатенации, замыкания (*), обращения и обратного гомоморфизма. КС-языки не замкнуты относительно пересечения и дополнения, но пересечение КС-языка с регулярным всегда является КС-языком.
- *Проверка пустоты КС-языка*. Существует алгоритм, который по данной грамматике G определяет, порождает ли она какие-нибудь цепочки. Аккуратная реализация этой проверки выполняется за время, прямо пропорциональное размеру самой грамматики.
- *Проверка принадлежности КС-языку*. Алгоритм Кока-Янгера-Касами определяет, принадлежит ли данная цепочка данному КС-языку. Если язык зафиксирован, эта проверка требует времени $O(n^3)$, где n — длина проверяемой цепочки.

Литература

Нормальная форма Хомского впервые описана в [2], нормальная форма Грейбах — в [4], хотя конструкция, описанная в упражнении 7.1.11, принадлежит Полу (M. C. Paull).

Многие фундаментальные свойства контекстно-свободных языков установлены в [1]. Среди них лемма о накачке, основные свойства замкнутости, а также проверки для простых вопросов, таких как пустота и конечность КС-языка. Результаты о незамкнутости относительно пересечения и дополнения происходят из работы [6], а дополнительные результаты о замкнутости, включая замкнутость КС-языков относительно обратного гомоморфизма, — из [3]. Лемма Огдена предложена в [5].

Алгоритм Кока-Янгера-Касами имеет три независимых источника. Работа Кока распространялась частным образом и не была опубликована. Версия по сути того же алгоритма, записанная Касами, появилась только в закрытом докладе Воздушных Сил США. И лишь работа Янгера была опубликована в [7].

1. Y. Bar-Hillel, M. Perles, and E. Shamir, “On formal properties of simple phrase-structure grammars”, *Z. Phonetic. Sprachwiss. Kommunikationsforsch.* **14** (1961), pp. 143–172.
2. N. Chomsky, “On certain formal properties of grammars”, *Information and Control* **2**:2 (1959), pp. 137–167. (Хомский Н. О некоторых формальных свойствах грамматик. — Кибернетический сборник, вып. 5. — М.: ИЛ, 1962. — С. 279–311.)
3. S. Ginsburg and G. Rose, “Operations which preserve definability in languages”, *J. ACM* **10**:2 (1963), pp. 175–195. (Гинзбург С., Роуз Дж. Об инвариантности классов языков относительно некоторых преобразований. — Кибернетический сборник, Новая серия, вып. 5. — М.: Мир, 1968. — С. 138–166.)
4. S. Greibach, “A new normal-form theorem for context-free phrase structure grammars”, *J. ACM* **12**:1 (1965), pp. 42–52.
5. W. Ogden, “A helpful result for proving inherent ambiguity”, *Mathematical Systems Theory* **2**:3 (1969), pp. 31–42. (Огден У. Результат, полезный для доказательства существенной неоднозначности. — сб. “Языки и автоматы”. — М.: Мир, 1975. — С. 109–113.)
6. S. Scheinberg, “Note on the boolean properties of context-free languages”, *Information and Control* **3**:4 (1960), pp. 372–375.
7. D. H. Younger, “Recognition and parsing of context-free languages in time n^3”, *Information and Control* **10**:2 (1967), pp. 189–208. (Янгер Д. Распознавание и анализ контекстно-свободных языков за время n^3. — Сб. “Проблемы математической логики”. — М.: Мир, 1970. — С. 344–362.)

ГЛАВА 8

Введение в теорию машин Тьюринга

В этой главе направление наших усилий меняется. До сих пор нас интересовали в основном простые классы языков и пути их использования для относительно ограниченных задач, вроде анализа протоколов, поиска в текстах или синтаксического анализа. Теперь начнется исследование языков, которые вообще можно определить с помощью вычислительных устройств. Это равносильно изучению того, что может компьютер, поскольку распознавание цепочек языка является формальным способом выражения любой задачи, а решение задачи — это разумное представление того, что делает компьютер.

Мы предполагаем, что читатель знаком с программированием на языке C. Вначале, используя это знание, с помощью неформальных доводов покажем, что существуют определенные задачи (проблемы), которые с помощью компьютера решить невозможно. Эти задачи называются "неразрешимыми". Затем познакомимся с классической формальной моделью компьютера, которая называется машиной Тьюринга (MT). И хотя машины Тьюринга совершенно не похожи на компьютеры, и было бы весьма неэффективно их производить и продавать, тем не менее машина Тьюринга получила признание как точная модель того, что способно делать любое физическое вычислительное устройство.

В главе 9 машина Тьюринга используется для развития теории "неразрешимых" проблем, т.е. проблем, которые не может решить ни один компьютер. Мы покажем, что многие просто формулируемые задачи в действительности неразрешимы. Примером может служить распознавание, является ли данная грамматика неоднозначной, и ряд таких примеров будет продолжен.

8.1. Задачи, не решаемые компьютерами

Цель этого раздела — с помощью C-программирования неформально доказать, что существуют задачи, которые невозможно решить, используя компьютер. Мы рассмотрим частную задачу распознавания, является ли текст `hello, world` первым, что печатает C-программа. Можно подумать, будто имитация любой программы всегда позволит сказать, что она делает, однако в реальности нам придется "бороться" с программами, которые работают невообразимо долго, прежде чем что-нибудь вывести на печать. Эта про-

блема — незнание момента, в который что-то произойдет, если вообще произойдет, — является основной причиной нашей неспособности распознать, что делает программа. Однако доказательство того, что не существует алгоритма решения указанной задачи распознавания, весьма сложно и требует определенного формализма. В этом разделе предпочтение отдается формальным доказательствам, а не интуиции.

8.1.1. Программы печати „Hello, world“

На рис. 8.1 показана первая С-программа, представленная в классической книге Кернигана и Ритчи.[1] Несложно обнаружить, что данная программа печатает `hello, world` и останавливается. Она настолько прозрачна, что обычно знакомство с языками программирования начинается демонстрацией того, как на этих языках написать программу печати `hello, world`.

```
main()
{
   printf("hello, world\n");
}
```

Рис. 8.1. Программа Кернигана и Ритчи, приветствующая мир

Однако есть и другие программы, которые тоже печатают `hello, world`, причем отнюдь не очевидно, что они делают именно это. На рис. 8.2 представлена еще одна программа, которая, возможно, печатает `hello, world`. Она получает на вход n и ищет положительные целые решения уравнения $x^n + y^n = z^n$. Если находит, то печатает `hello, world`. Не обнаружив целых x, y и z, удовлетворяющих данному уравнению, она продолжает поиск до бесконечности и никогда не печатает `hello, world`.

Для того чтобы понять, как работает эта программа, сначала заметим, что `exp` является встроенной функцией вычисления экспоненты. Основной программе нужно искать тройки (x, y, z) в порядке, гарантирующем, что каждая тройка в конце концов достигается. Для корректного поиска используется четвертая переменная, `total`, которая инициируется значением 3 и увеличивается в while-цикле каждый раз на 1, так что ее значение в конце концов достигает любого натурального числа. Внутри while-цикла `total` разделяется на три положительных целых x, y и z, причем x изменяется в for-цикле от 1 до `total-2`, y внутри этого for-цикла изменяется от 1 до `total-x-1`, а остаток значения `total`, между 1 и `total-2`, становится значением z.

---

[1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 1978, Prentice-Hall, Englewood Cliffs, NJ. (Керниган Б., Ритчи Д. Язык программирования Си. — М.: Финансы и статистика, 1992. См. также Керниган Б., Ритчи Д., Фьюэр А. Язык программирования Си. Задачи по языку Си. — М.: Финансы и статистика, 1985.)

Во внутреннем цикле для тройки (x, y, z) проверяется равенство $x^n + y^n = z^n$. Если оно выполняется, то печатается `hello, world`, а если нет — не печатается ничего.

```
int exp(int i, n)
/* вычисление i в степени n */
{
    int ans, j;
    ans = 1;
    for (j=1; j<=n; j++) ans *= i;
    return(ans);
}

main()
{
    int n, total, x, y, z;
    scanf("%d", n);
    total = 3;
    while (1) {
        for (x=1; x<=total-2; x++)
            for (y=1; x<=total-x-1; y++){
                z = total - x - y;
                if (exp(x,n) + exp(y,n) == exp(z,n))
                    printf("hello, world\n");
            }
        total++;
    }
}
```

Рис. 8.2. Великая теорема Ферма, выраженная в программе приветствия мира

Если прочитано значение 2, то программа находит набор значений `total` = 12, $x = 3$, $y = 4$ и $z = 5$, для которого $x^n + y^n = z^n$. Таким образом, если на входе 2, то программа действительно печатает `hello, world`.

Однако для любого $n > 2$ программа никогда не найдет тройку положительных целых, удовлетворяющих $x^n + y^n = z^n$, и `hello, world` напечатано не будет. Интересно, что еще несколько лет назад не было известно, печатает ли данная программа `hello, world`, т.е. имеет ли уравнение $x^n + y^n = z^n$ для некоторого большого n. Утверждение, что это уравнение не имеет решений, было сформулировано Ферма более 300 лет назад, но до недавних пор доказательство не было найдено. Данное утверждение часто называется “великой теоремой Ферма” (“Fermat’s last theorem”).

Определим *проблему “hello, world”* следующим образом. По данной С-программе и ее входу определить, печатает ли она в качестве первых 12 символов `hello, world`. В дальнейшем для краткости утверждение о том, что программа печатает `hello, world`, употребляется в смысле, что она печатает `hello, world` как первые 12 символов.

Поскольку математикам понадобилось более 300 лет для решения вопроса, выраженного простенькой 22-строчной программой, то представляется правдоподобным, что общая задача распознавания, печатает ли `hello, world` данная программа с данным входом, должна быть действительно сложной. Таким образом, любую задачу, не решенную математиками до сих пор, можно поставить как вопрос вида "печатает ли `hello, world` данная программа с данным входом?". Было бы замечательно, если бы нам удалось написать программу, которая проверяет любую программу P и ее вход I и определяет, печатается ли `hello, world` при выполнении P со входом I. Мы докажем, что такой программы не может быть.

Почему должны существовать неразрешимые проблемы

Нелегко доказать, что какая-то определенная проблема, например, обсуждаемая здесь проблема "hello, world", должна быть неразрешимой. Однако легко понять, почему почти все проблемы должны быть неразрешимыми в рамках любой системы, включающей программирование. Вспомним, что "проблема" — это в действительности вопрос о принадлежности цепочки языку. Множество различных языков над любым алфавитом, в котором более одного символа, *несчетно*.[2] Таким образом, невозможно пронумеровать языки натуральными числами так, чтобы каждый язык получил номер, и каждое число было назначено одному языку.

С другой стороны, программы, будучи конечными цепочками в конечном алфавите (обычно это подмножество алфавита ASCII), допускают такую нумерацию, т.е. образуют *счетное* множество. Их можно упорядочить по длине, а программы одной длины расположить в лексикографическом порядке. Таким образом, можно говорить о программе с номером 1, 2 и вообще с номером i.

В результате можно утверждать, что проблем существует "бесконечно больше", чем программ. Если случайно выбрать язык, то почти наверняка он окажется неразрешимой проблемой. И то, что проблемы в большинстве своем *кажутся* разрешимыми, обусловлено лишь редкостью обращения к случайным проблемам. Наоборот, мы склонны к изучению относительно простых, хорошо структурированных проблем, и, действительно, они часто разрешимы. Однако даже среди проблем, которые интересуют нас и допускают ясную и краткую формулировку, можно найти много неразрешимых, и в том числе проблему "hello, world".

8.1.2. Гипотетическая программа проверки приветствия мира

Докажем невозможность создания программы проверки приветствия мира (печати "hello, world") методом от противного. Предположим, что существует программа, скажем, H, которая получает на вход программу P и ее вход I и сообщает, печатает ли P,

[2] Это справедливо и для односимвольного алфавита. — Прим. ред.

имея на входе *I*, текст `hello, world`. Работа *H* представлена на рис. 8.3. В частности, *H* всегда печатает либо `yes` (да), либо `no` (нет), и ничего более.

Если проблема имеет алгоритм, подобный программе *H*, который на любом экземпляре проблемы правильно отвечает "да" или "нет", то такая проблема называется "разрешимой". В противном случае проблема "неразрешима". Наша цель — доказать, что *H* не существует, т.е. проблема "`hello, world`" является неразрешимой.

Рис. 8.3. Гипотетическая программа обнаружения "hello, world"

Для доказательства этого утверждения методом от противного внесем в *H* несколько изменений, построив в итоге программу H_2, и докажем, что ее не существует. Поскольку изменения, вносимые в *H*, можно совершить с любой С-программой, единственным сомнительным утверждением будет существование *H*, т.е. утверждение, которое мы и опровергнем.

Для простоты доказательства сделаем несколько допущений о С-программах. Эти гипотезы упрощают, а не усложняют работу *H*, поэтому, если доказать, что программы проверки "`hello, world`" не существует для таких ограниченных С-программ, то для более широкого класса программ такой программы проверки тем более не может быть. Итак, допустим следующее.

1. Весь выход программ состоит из символов, т.е. графические и иные средства несимвольного представления информации не используются.
2. Все символы печатаются с помощью функции `printf`, без использования `putchar()` и других.

Теперь предположим, что *H* существует. Наша первая модификация изменяет выход `no`, который является откликом *H*, когда ее входная программа *P* со своим входом *I* не печатает `hello, world`. Как только *H* печатает "n", нам становится понятно, что далее рано или поздно появится "o".[3] Таким образом, каждый вызов `printf` в *H* изменим так, чтобы вместо "n" печаталось `hello, world`. Другие вызовы, которые печатают "o" без "n", опускаются. В результате получим программу H_1, которая ведет себя точно так же, как и *H*, но печатает `hello, world` вместо `no` (рис. 8.4).

[3] Наиболее вероятно, что программа печатает `no` в одном вызове `printf`, но она может печатать "n" в одном вызове `printf`, а "o" — в следующем.

Следующее преобразование программы несколько сложнее. Оно, по сути, представляет собой прием, позволивший Алану Тьюрингу доказать свой результат о неразрешимости для машин Тьюринга. Поскольку нас в действительности интересуют программы, которые получают на вход другие программы и что-то о них сообщают, ограничим H_1 следующим образом.

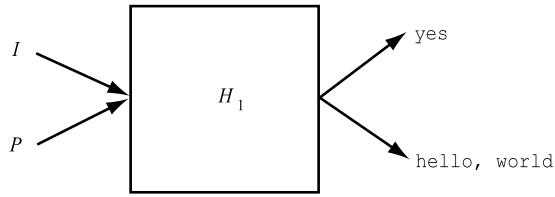


Рис. 8.4. H_1 *ведет себя, как* H, *но печатает* `hello, world` *вместо* `no`

1. Она получает на вход только *P*, а не *P* и *I*.
2. Она отвечает, что сделала бы *P*, если бы ее входом была она сама, т.е. что выдала бы H_1, имея на входе *P* в качестве как программы, так и входа *I*.

Для создания программы H_2, представленной на рис. 8.5, внесем в H_1 следующие изменения.

1. Вначале H_2 считывает весь вход *P* и запоминает его в массиве *A*, выделяя память под массив с помощью `malloc`[4].
2. Затем H_2 имитирует H_1, но вместо чтения из *P* или *I* программа H_2 читает из копии в массиве *A*. Для запоминания мест в *P* и *I*, до которых дочитала H_1, H_2 может содержать два курсора, отмечающих позиции в *A*.

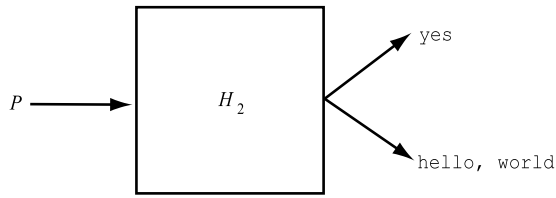


Рис. 8.5. H_2 *ведет себя, как* H_1, *но использует вход* P *в качестве как* P, *так и* I

Теперь все готово для доказательства того, что H_2 не существует. Таким образом, не существует H_1 и, аналогично, *H*. Для того чтобы в этом убедиться, нужно представить себе, что делает H_2, когда получает себя в качестве входа (рис. 8.6). Напомним, что H_2,

[4] Функция `malloc` системы UNIX распределяет блок памяти, размер которого задается в ее вызове. Эта функция используется, когда нужный размер памяти невозможно определить до начала выполнения программы, как и при чтении входа произвольной длины. Обычно `malloc` вызывается несколько раз по мере того, как возрастают объем прочитанного входа и потребность в памяти.

получив некоторую программу P на вход, печатает `yes`, если P печатает `hello, world`, получая себя в качестве входа. Кроме того, H_2 печатает `hello, world`, если P, получив себя на вход, не выдает `hello, world` в качестве начала своего выхода.

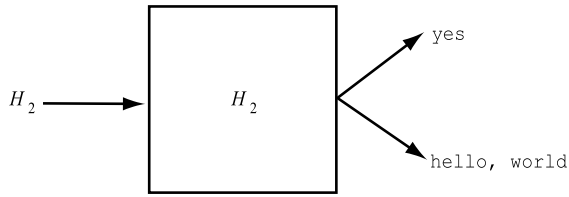


Рис. 8.6. Что делает H_2*, получая себя в качестве входа?*

Предположим, что H_2 в рамке (см. рис. 8.6) печатает `yes`. Тогда H_2 в рамке говорит о своем входе H_2, что H_2, получая себя в качестве входа, выдает вначале `hello, world`. Но мы только что предположили, что выход H_2 начинается текстом `yes`, а не `hello, world`.

Таким образом, получается, что выходом программы в рамке (см. рис. 8.6) является не `yes`, а `hello, world`, поскольку возможно либо одно, либо другое. Но если H_2, получая себя на вход, печатает `hello, world`, то выходом программы в рамке на рис. 8.6 должно быть `yes`. Каким бы ни был предполагаемый выход H_2, доказано, что она печатает противоположное.

Описанная ситуация противоречива, и мы приходим к выводу, что H_2 не может существовать. Это опровергает предположение о существовании H. Таким образом, доказано, что ни одна программа H не может сказать, печатает ли данная программа P со входом I текст `hello, world`.

8.1.3. Сведение одной проблемы к другой

Печатает ли данная программа в начале своего выхода текст `hello, world`? Мы уже знаем, что ни одна компьютерная программа этот вопрос разрешить не может. Проблема, которую нельзя разрешить с помощью компьютера, называется *неразрешимой*. Формальное определение “неразрешимого” будет дано в разделе 9.3, а пока этот термин используется неформально. Предположим, что нам нужно определить, разрешима ли некоторая новая проблема. Мы можем попытаться написать программу для ее разрешения, но, если нам непонятно, как это сделать, мы можем попробовать доказать, что такой программы не существует.

Неразрешимость новой проблемы можно было бы доказать аналогично тому, как это было сделано для проблемы “hello, world”: предположить, что программа разрешения существует, и построить противоречивую программу, которая должна делать одновременно две взаимно исключающие вещи, как программа H_2. Однако если у нас есть проблема, неразрешимость которой установлена, то нам не обязательно вновь доказывать противоречивость. Достаточно показать, что если новая проблема имеет решение, то его

можно использовать для разрешения уже известной неразрешимой проблемы. Такой подход представлен на рис. 8.7 и называется *сведением* P_1 к P_2.

Предположим, нам известно, что проблема P_1 неразрешима, а P_2 — новая проблема, неразрешимость которой нужно доказать. Допустим, что существует программа, которая представлена ромбом с отметкой "разрешить" (см. рис. 8.7) и печатает `yes` или `no` в зависимости от того, принадлежит или нет ее входной экземпляр проблемы P_2 языку этой проблемы.[5]

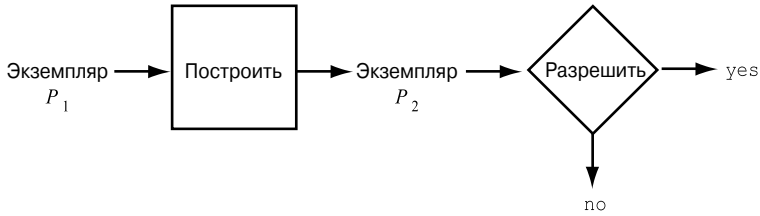


Рис. 8.7. Если бы можно было разрешить проблему P_2, *то ее решение мы могли бы использовать для разрешения проблемы* P_1

Для доказательства неразрешимости проблемы P_2 нужно создать алгоритм, который представлен квадратом на рис. 8.7 и преобразует любой экземпляр P_1 в экземпляр P_2, причем их ответы всегда совпадают. Имея такое преобразование, проблему P_1 можно разрешить следующим образом.

1. Применить алгоритм построения к данному экземпляру проблемы P_1, т.е. к строке w, которая может принадлежать или не принадлежать языку P_1, и получить строку x.
2. Проверить, принадлежит ли x языку P_2, и перенести ответ на w и P_1.

Пример 8.1. Применим описанный метод для доказательства, что вопрос "вызывает ли когда-нибудь программа Q с данным входом y функцию `foo`" неразрешим. Заметим, что в Q может не быть функции `foo`, и в этом случае задача решается просто. Однако она становится сложной, если Q имеет функцию `foo`, но с y на входе может как достичь, так и не достичь вызова `foo`. Поскольку нам известна только одна неразрешимая проблема, роль P_1 на рис. 8.7 играет проблема "hello, world". В качестве P_2 выступает описанная только что *проблема "calls-foo"*. Предположим, что существует программа разрешения этой проблемы. Наша задача — построить алгоритм, который преобразует проблему "hello, world" в проблему "calls-foo".

---

[5] Напомним, что проблема в действительности представляет собой язык. Говоря о проблеме разрешения, приводят ли данные программа и вход к печати `hello, world`, мы в действительности говорим о цепочках, образованных С-программами и их входными данными. Это множество цепочек является языком над алфавитом из символов ASCII.

Действительно ли компьютер может делать все это?

Если присмотреться к программе, представленной на рис. 8.2, можно спросить, действительно ли она находит контрпримеры к великой теореме Ферма. Как-никак, целые числа в типичном компьютере имеют всего 32 бит, и если самый маленький контрпример содержит числа, измеряемые биллионами, то произойдет ошибка переполнения, прежде чем отыщется решение. И вообще, компьютер со 128 Мбайт оперативной памяти и 30-гигабайтным диском имеет "всего" $256^{30128000000}$ состояний и является, таким образом, конечным автоматом.

Однако рассмотрение компьютера как конечного автомата (или мозга как конечного автомата, откуда, кстати, происходит идея КА) непродуктивно. Число состояний настолько велико, а пределы настолько размыты, что прийти к каким-либо полезным заключениям невозможно. В действительности, есть все причины поверить в то, что при желании множество состояний компьютера можно расширять до бесконечности.

Например, целые числа можно представить в виде связанных списков цифр произвольной длины. Если память исчерпывается, программа печатает запрос на замену заполненного диска новым, пустым. И такие запросы могут повторяться по мере надобности сколько угодно раз. Такая программа будет намного сложнее, чем приведенная на рис. 8.2, но все равно мы сможем ее написать. Подобные приемы позволят любой другой программе обойти ограничения на размер памяти, величину целых чисел и другие параметры.

Итак, имея программу Q и ее вход y, мы должны построить программу R и ее вход z так, чтобы R со входом z вызывала `foo` тогда и только тогда, когда Q со входом y печатает `hello, world`. Конструкция проста.

1. Если Q имеет вызываемую функцию `foo`, переименовать ее и все ее вызовы. Очевидно, новая программа Q_1 выполняет то же самое, что и Q.
2. Добавить в Q_1 функцию `foo`. Эта функция не делает ничего и не вызывается. Полученная программа называется Q_2.
3. Изменить Q_2 так, чтобы первые 12 символов ее печати запоминались в глобальном массиве A. Пусть полученная программа называется Q_3.
4. Изменить Q_3 так, чтобы после каждого выполнения инструкции печати с использованием массива A проверялось, напечатано ли не менее 12 символов, и если так, то не образуют ли первые 12 символов текст `hello, world`. В этом случае вызвать новую функцию `foo`, добавленную в п. 2. Полученная программа есть R, а ее вход z совпадает с y.

Предположим, что Q со входом y печатает `hello, world` в качестве начала своего выхода. Тогда построенная R вызывает `foo`. Однако если Q со входом y не печатает сначала `hello, world`, то R со входом z никогда не вызывает `foo`. Если можно решить,

вызывает ли R со входом z функцию `foo`, то можно и решить, печатает ли Q со входом y сначала `hello, world`. Поскольку нам известно, что алгоритма решения проблемы "`hello, world`" нет, и все 4 этапа построения R по Q можно выполнить, отредактировав код, то предположение о том, что программа разрешения проблемы "calls-foo" существует, ложно. Такой программы нет, и данная проблема неразрешима. □

Направление сведения является важным

Обычная ошибка — пытаться доказывать неразрешимость проблемы P_2 путем сведения ее к известной неразрешимой проблеме P_1, т.е. доказывать утверждение "если P_1 разрешима, то P_2 разрешима". Это утверждение, безусловно, истинное, бесполезно, поскольку предпосылка "P_1 разрешима" ложна.

Единственный путь доказать, что новая проблема P_2 неразрешима, состоит в том, чтобы свести к ней уже известную неразрешимую проблему, т.е. доказать утверждение "если P_1 неразрешима, то P_2 неразрешима". Поскольку неразрешимость P_1 уже установлена, приходим к выводу, что P_2 неразрешима.

8.1.4. Упражнения к разделу 8.1

8.1.1. Сведите проблему "`hello, world`" к каждой из следующих проблем. Используйте неформальный стиль данного раздела для описания правдоподобных преобразований программ и не беспокойтесь о реальных пределах размеров файла или памяти в компьютерах:

а) (**!***) по данной программе и ее входу определить, останавливается ли она когда-нибудь, т.е. не зацикливается ли при данном входе;

б) по данной программе и ее входу определить, печатает ли она вообще что-нибудь;

в) (**!**) по данным двум программам и входу определить, порождают ли они один и тот же выход при данном входе.

8.2. Машина Тьюринга

Цель теории неразрешимых проблем состоит не только в том, чтобы установить существование таких проблем (что само по себе не просто), но также в том, чтобы обеспечить программистов информацией, какие задачи программирования можно решить, а какие нельзя. Теория также имеет огромное практическое значение, когда рассматриваются проблемы, которые хотя и разрешимы, но требуют слишком большого времени для их решения (см. главу 10). Эти проблемы, называемые "трудно разрешимыми", или "труднорешаемыми", подчас доставляют программистам и разработчикам систем больше хлопот, чем неразрешимые проблемы. Причина в том, что неразрешимые проблемы

редко возникают на практике, тогда как трудно разрешимые встречаются каждый день. Кроме того, они часто допускают небольшие модификации условий или эвристические решения. Таким образом, разработчику постоянно приходится решать, относится ли данная проблема к классу труднорешаемых и что с ней делать, если это так.

Нам нужен инструмент, позволяющий доказывать неразрешимость или труднорешаемость повседневных вопросов. Методика, представленная в разделе 8.1, полезна для вопросов, касающихся программ, но ее нелегко перенести на проблемы, не связанные с программами. Например, было бы нелегко свести проблему "`hello, world`" к проблеме неоднозначности грамматики.

Таким образом, нужно перестроить нашу теорию неразрешимости, основанную не на программах в языке C или других языках, а на очень простой модели компьютера, которая называется машиной Тьюринга. Это устройство по существу представляет собой конечный автомат с бесконечной лентой, на которой он может как читать, так и записывать данные. Одно из преимуществ машин Тьюринга по сравнению с программами как представлением вычислений состоит в том, что машина Тьюринга достаточно проста и ее конфигурацию можно точно описать, используя нотацию, весьма похожую на МО МП-автоматов. Для сравнения, состояние C-программы включает все переменные, возникающие при выполнении любой цепочки вызовов функций, и нотация для описания этих состояний настолько сложна, что практически не позволяет проводить понятные формальные доказательства.

С использованием нотации машин Тьюринга будет доказана неразрешимость некоторых проблем, не связанных с программированием. Например, в разделе 9.4 будет показано, что "проблема соответствий Поста", простой вопрос о двух списках цепочек, неразрешим, и что эта проблема облегчает доказательство неразрешимости вопросов о грамматиках, вроде неоднозначности. Аналогично, когда будут введены трудно разрешимые проблемы, мы увидим, что к их числу принадлежат и определенные вопросы, казалось бы, не связанные с вычислениями, например, выполнимость булевских формул.

8.2.1. Поиски решения всех математических вопросов

В начале XX столетия великий математик Д. Гильберт поставил вопрос о поисках алгоритма, который позволял бы определить истинность или ложность любого математического утверждения. В частности, он спрашивал, есть ли способ определить, истинна или ложна произвольная формула в исчислении предикатов первого порядка с целыми числами. Исчисление предикатов первого порядка с целыми (арифметика) достаточно мощно, чтобы выразить утверждения типа "эта грамматика неоднозначна" или "эта программа печатает `hello, world`". Поэтому, окажись предположение Гильберта правильным, данные проблемы были бы разрешимыми.

Однако в 1931 г. К. Гедель опубликовал свою знаменитую теорему о неполноте. Он доказал, что существует истинная формула первого порядка с целыми, которую нельзя ни доказать, ни опровергнуть в исчислении предикатов первого порядка над целыми. Его

техника доказательства похожа на конструкцию противоречивой программы H_2 из раздела 8.1.2, но имеет дело с функциями целого аргумента, а не с С-программами.

Исчисление предикатов было не единственным понятием, применяемым для формализации “любого возможного вычисления”. В действительности, исчисление предикатов, будучи декларативным, а не вычислительным, конкурировало с различными нотациями, включая “частично рекурсивные функции”. В 1936 г. А. М. Тьюринг в качестве модели “любого возможного вычисления” предложил машину Тьюринга. Эта модель была не декларативной, а “машиноподобной”, хотя настоящие электронные и даже электромеханические машины появились лишь несколько лет спустя (и сам Тьюринг занимался их разработкой во время второй мировой войны).

Интересно, что все когда-либо предложенные серьезные вычислительные модели имеют одинаковую мощность, т.е. все они вычисляют одни и те же функции или распознают одни и те же множества. Недоказуемое предположение, что любой общий метод вычислений позволяет вычислять лишь частично рекурсивные функции (и их же способны вычислять машины Тьюринга или современные компьютеры), известно как *гипотеза Черча* (следуя логике А. Черча), или *тезис Черча-Тьюринга*.

8.2.2. Описание машин Тьюринга

Машина Тьюринга изображена на рис. 8.8. Машина состоит из *конечного управления*, которое может находиться в любом из конечного множества состояний. Есть *лента*, разбитая на *клетки*; каждая клетка может хранить любой символ из конечного их множества.

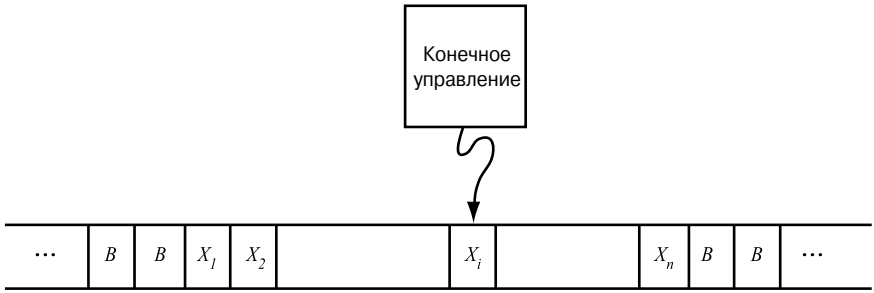


Рис. 8.8. Машина Тьюринта

Изначально на ленте записан *вход*, представляющий собой цепочку символов конечной длины. Символы выбраны из *входного алфавита*. Все остальные клетки, до бесконечности, слева и справа от входа содержат специальный символ, называемый *пустым символом*, или *пробелом*. Он является не входным, а *ленточным символом*. Кроме входных символов и пробела возможны другие ленточные символы.

Ленточная головка (далее просто *головка*) всегда устанавливается на какую-то из клеток ленты, которая называется *сканируемой*, или *обозреваемой*. Вначале обозревается крайняя слева клетка входа.

Переход машины Тьюринга — это функция, зависящая от состояния конечного управления и обозреваемого символа. За один переход машина Тьюринга должна выполнить следующие действия.

1. Изменить состояние. Следующее состояние может совпадать с текущим.
2. Записать ленточный символ в обозреваемую клетку. Этот символ замещает любой символ в этой клетке, в частности, символы могут совпадать.
3. Сдвинуть головку влево или вправо. В нашей формализации не допускается, чтобы головка оставалась на месте. Это ограничение не изменяет того, что могут вычислить машины Тьюринга, поскольку любая последовательность переходов с остающейся на месте головкой и последующим сдвигом может быть сжата до одного перехода с изменением состояния и ленточного символа и сдвигом головки влево или вправо.

Формальная запись, используемая для *машин Тьюринга* (МТ), похожа на запись конечных автоматов или МП-автоматов. МТ описывается семеркой

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F),$$

компоненты которой имеют следующий смысл.

Q — конечное множество *состояний* конечного управления.

Σ — конечное множество *входных символов*.

Γ — множество *ленточных символов*; Σ всегда есть подмножество Γ.

δ — *функция переходов*. Аргументами $\delta(q, X)$ являются состояние q и ленточный символ X, а значением, если оно определено, — тройка (p, Y, D). В этой тройке p есть следующее состояние из Q, Y — символ из Γ, который записывается вместо символа в обозреваемой клетке, а D — *направление* сдвига головки “влево” или “вправо”, обозначаемое, соответственно, как L или R.

q_0 — *начальное состояние* из Q, в котором управление находится в начале.

B — *пустой символ*, или *пробел*. Этот символ принадлежит Γ, но не Σ, т.е. не является входным. Вначале он записан во всех клетках, кроме конечного их числа, где хранятся входные символы. Остальные символы называются значащими.

F — множество *заключительных*, или *допускающих*, состояний; является подмножеством Q.

8.2.3. Конфигурации машин Тьюринга

Для формального описания работы машины Тьюринга нужно построить систему записи конфигураций, или *мгновенных описаний* (*МО*), подобную нотации, определенной для МП-автоматов. Поскольку МТ имеет неограниченно длинную ленту, можно подумать, что конечное описание конфигураций МТ невозможно. Однако после любого конечного числа шагов МТ может обозреть лишь конечное число клеток, хотя и ничем не ограниченное. Таким образом, в любом МО есть бесконечный префикс и бесконечный суффикс из клеток, которые еще не обозревались. Все эти клетки должны содержать или пробелы, или входные символы из конечного их множества. Таким образом, в МО вклю-

чаются только клетки между крайними слева и справа значащими символами. В отдельных случаях, когда головка обозревает один из пробелов перед или за участком значащих символов, конечное число пробелов также включается в МО.

Кроме ленты, нужно представить конечное управление и позицию головки. Для этого состояние помещается непосредственно слева от обозреваемой клетки. Во избежание неоднозначности получаемой цепочки, состояния обозначаются символами, отличными от ленточных. Таким образом, для представления МО используется цепочка $X_1X_2\ldots X_{i-1}qX_iX_{i+1}\ldots X_n$. Здесь q — состояние МТ, головка обозревает i-ю слева клетку, а $X_1X_2\ldots X_n$ представляет собой часть ленты между крайними слева и справа значащими символами. Как исключение, если головка находится слева или справа от значащих символов, некоторое начало или окончание $X_1X_2\ldots X_n$ пусто, а i имеет значение, соответственно, 1 или n.

Переходы МТ $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ описываются с помощью отношения $\underset{M}{\vdash}$, использованного для МП-автоматов. Подразумевая МТ M, для отображения переходов используем $\vdash$. Как обычно, для указания нуля или нескольких переходов МТ M используется отношение $\overset{*}{\underset{M}{\vdash}}$ или $\overset{*}{\vdash}$.

Пусть $\delta(q, X) = (p, Y, L)$, т.е. головка сдвигается влево. Тогда

$$X_1X_2\ldots X_{i-1}qX_iX_{i+1}\ldots X_n \underset{M}{\vdash} X_1X_2\ldots X_{i-2}pX_{i-1}YX_{i+1}\ldots X_n.$$

Заметим, как этот переход отображает изменение состояния на p и сдвиг головки на клетку $i - 1$. Здесь есть два важных исключения.

1. Если $i = 1$, то M переходит к пробелу слева от X_1. В этом случае

$$qX_1X_2\ldots X_n \underset{M}{\vdash} pBYX_2\ldots X_n.$$

2. Если $i = n$ и $Y = B$, то символ B, заменяющий X_n, присоединяется к бесконечной последовательности пробелов справа и не записывается в следующем МО. Таким образом,

$$X_1X_2\ldots X_{n-1}qX_n \underset{M}{\vdash} X_1X_2\ldots X_{n-2}pX_{n-1}.$$

Пусть теперь $\delta(q, X) = (p, Y, R)$, т.е. головка сдвигается вправо. Тогда

$$X_1X_2\ldots X_{i-1}qX_iX_{i+1}\ldots X_n \underset{M}{\vdash} X_1X_2\ldots X_{i-1}YpX_{i+1}\ldots X_n.$$

Этот переход отражает сдвиг головки в клетку $i + 1$. Здесь также есть два важных исключения.

1. Если $i = n$, то $(i + 1)$-я клетка содержит пробел и не является частью предыдущего МО. Таким образом,

$$X_1X_2\ldots X_{n-1}qX_n \underset{M}{\vdash} X_1X_2\ldots X_{n-1}YpB.$$

2. Если $i = 1$ и $Y = B$, то символ B, записываемый вместо X_1, присоединяется к бесконечной последовательности пробелов слева и опускается в следующем МО. Таким образом,

$$qX_1X_2\ldots X_n \underset{M}{\vdash} pX_2\ldots X_n.$$

Пример 8.2. Построим машину Тьюринга и посмотрим, как она ведет себя на типичном входе. Данная машина Тьюринга будет допускать язык $\{0^n1^n \mid n \geq 1\}$. Изначально на ее ленте записана конечная последовательность символов 0 и 1, перед и за которыми находятся бесконечные последовательности пробелов. МТ попеременно будет изменять 0 на X и 1 на Y до тех пор, пока все символы 0 и 1 не будут сопоставлены друг другу.

Более детально, начиная с левого конца входной последовательности, МТ циклично меняет 0 на X и движется вправо через все символы 0 и Y, пока не достигнет 1. Она меняет 1 на Y и движется вправо через все символы Y и 0, пока не найдет X. В этот момент она ищет 0 непосредственно справа и, если находит, меняет его на X и продолжает процесс, меняя соответствующую 1 на Y.

Если непустой вход не принадлежит $\mathbf{0}^*\mathbf{1}^*$, то МТ рано или поздно не сможет совершить следующий переход и остановится без допускания. Однако если она заканчивает работу, изменив все символы 0 на X в том же цикле, в котором она изменила последнюю 1 на Y, то ее вход имеет вид 0^n1^n, и она его допускает. Формальным описанием данной МТ является

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\}),$$

где δ представлена в таблице на рис. 8.9.

Состояние	Символ 0	1	X	Y	B
q_0	(q_1, X, R)	—	—	(q_3, Y, R)	—
q_1	$(q_1, 0, R)$	(q_2, Y, L)	—	(q_1, Y, R)	—
q_2	$(q_2, 0, L)$	—	(q_0, X, R)	(q_2, Y, L)	—
q_3	—	—	—	(q_3, Y, R)	(q_4, B, R)
q_4	—	—	—	—	—

Рис. 8.9. Машина Тьюринга, допускающая $\{0^n1^n \mid n \geq 1\}$

В процессе вычислений M часть ленты, на которой побывала ее головка, всегда содержит последовательность символов, описываемую регулярным выражением $X^*\mathbf{0}^*Y^*\mathbf{1}^*$. Таким образом, там есть последовательность символов X, заменивших 0, за которыми идут символы 0, еще не измененные на X. Затем идут символы Y, заменившие 1, и символы 1, еще не замененные Y. За этой последовательностью еще могут находиться символы 0 и 1.

Состояние q_0 является начальным, и в него же переходит M, возвращаясь к крайнему слева из оставшихся символов 0. Если M находится в состоянии q_0 и обозревает 0, то в соответствии с правилами (см. рис. 8.9) она переходит в состояние q_1, меняет 0 на X и сдвигается вправо. Попав в состояние q_1, M продолжает движение вправо через все символы 0 и Y. Если M видит X или B, она останавливается (“умирает”). Однако, если M в состоянии q_1 видит 1, она меняет ее на Y, переходит в состояние q_2 и начинает движение влево.

Находясь в состоянии q_2, M движется влево через все символы 0 и Y. Достигая крайнего справа X, который отмечает правый конец блока нулей, измененных на X, M возвращается в состояние q_0 и сдвигается вправо. Возможны два случая.

1. Если M видит 0, то она повторяет описанный только что цикл.
2. Если же M обозревает Y, то она уже изменила все нули на X. Если все символы 1 заменены Y, то вход имел вид 0^n1^n, и M должна допускать. Таким образом, M переходит в состояние q_3 и начинает движение вправо по символам Y. Если первым после Y появляется пробел, то символов 0 и 1 было поровну, поэтому M переходит в состояние q_4 и допускает. Если же M обнаруживает еще одну 1, то символов 1 слишком много, и M останавливается, не допуская. Если M находит 0, то вход имеет ошибочный вид, и M также "умирает".

Приведем пример допускающего вычисления M на входе 0011. Вначале M находится в состоянии q_0, обозревая 0, т.е. начальное МО имеет вид q_00011. Полная последовательность переходов образована следующими МО.

$$q_00011 \vdash Xq_1011 \vdash X0q_111 \vdash Xq_20Y1 \vdash q_2X0Y1 \vdash$$
$$Xq_00Y1 \vdash XXq_1Y1 \vdash XXYq_11 \vdash XXq_2YY \vdash Xq_2XYY \vdash$$
$$XXq_0YY \vdash XXYq_3Y \vdash XXYYq_3B \vdash XXYYBq_4B$$

Рассмотрим поведение M на входе 0010, который не принадлежит допускаемому языку.

$$q_00010 \vdash Xq_1010 \vdash X0q_110 \vdash Xq_20Y0 \vdash q_2X0Y0 \vdash$$
$$Xq_00Y0 \vdash XXq_1Y0 \vdash XXYq_10 \vdash XXY0q_1B$$

Это поведение похоже на обработку входа 0011 до МО $XXYq_10$, где M впервые обозревает последний 0. M должна двигаться вправо, оставаясь в состоянии q_1, что приводит к МО $XXY0q_1B$. Однако у M в состоянии q_1 по символу B перехода нет, поэтому она останавливается, не допуская.

8.2.4. Диаграммы переходов для машин Тьюринга

Переходы машин Тьюринга можно представить графически, как и переходы МП-автоматов. *Диаграмма переходов* состоит из множества узлов, соответствующих состояниям МТ. Дуга из состояния q в состояние p отмечена одним или несколькими элементами вида X/YD, где X и Y — ленточные символы, а D — направление (L или R). Таким образом, если $\delta(q, X) = (p, Y, D)$, то отметка X/YD находится на дуге из q в p. Направление на диаграммах представляется не буквами L и R, а стрелками $\leftarrow$ и $\rightarrow$, соответственно.

Как и в других видах диаграмм переходов, начальное состояние представлено словом "начало" и стрелкой, входящей в это состояние. Допускающие состояния выделены двойными кружками. Таким образом, непосредственно из диаграммы о МТ известно все, кроме того, какой символ обозначает пробел. В дальнейшем считается, что это B, если не оговорено иное.

Пример 8.3. На рис. 8.10 представлена диаграмма переходов для машины Тьюринга из примера 8.2. Ее функция переходов изображена на рис. 8.9. □

Пример 8.4. Сегодня машины Тьюринга рассматриваются чаще всего в качестве "распознавателей языков" или, что равносильно, "решателей проблем". Однако сам Тьюринг рассматривал свою машину как вычислитель натуральнозначных функций. В его схеме натуральные числа представлялись в единичной системе счисления, как блоки из одного и того же символа, и машина вычисляла, изменяя длину блоков или строя новые блоки где-нибудь на ленте. В данном простом примере будет показано, как машина Тьюринга может вычислить функцию $\dot{-}$, которая называется *монусом* (monus) или *усеченной разностью* (proper subtraction) и определяется соотношением $m \dot{-} n = \max(m - n, 0)$, т.е. $m \dot{-} n$ есть $m - n$, если $m \geq n$, и 0, если $m < n$.

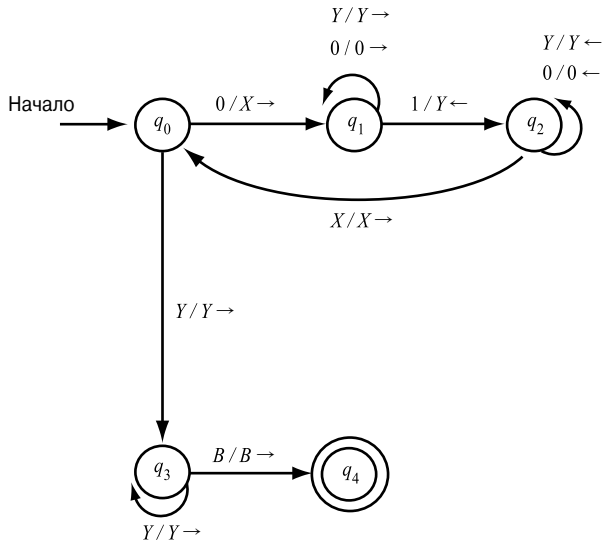


Рис. 8.10. Диаграмма переходов МТ, допускающая цепочки вида 0^n1^n

МТ, выполняющая эту операцию, определяется в виде

$$M = (\{q_0, q_1, \ldots, q_6\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B).$$

Отметим, что, поскольку МТ не используется для допускания входа, множество допускающих состояний не рассматривается. M начинает с ленты, состоящей из 0^m10^n и пробелов вокруг, и заканчивает лентой с $m \dot{-} n$ символами 0, окруженными пробелами.

M циклически находит крайний слева из оставшихся 0 и заменяет его пробелом. Затем движется вправо до 1. Найдя 1, M продолжает движение вправо до появления 0, который меняется на 1. Затем M возвращается влево в поисках крайнего слева 0, который идентифицируется после того, как M выходит на пробел и сдвигается вправо на одну клетку. Повторения заканчиваются в одной из следующих ситуаций.

1. В поисках 0 справа M встречает пробел. Это значит, что все n нулей в 0^n изменены на 1, и $n + 1$ нулей в 0^m заменены пробелами. Тогда M изменяет $n + 1$ единицу на

пробелы и добавляет 0, оставляя $m - n$ нулей на ленте. Поскольку в этом случае $m \geq n$, то $m - n = m \dot{-} n$.

2. Начиная цикл, M не может найти 0, чтобы заменить его пробелом, поскольку первые m нулей уже изменены на B. Это значит, что $n \geq m$, и $m \dot{-} n = 0$. M заменяет все оставшиеся символы 1 и 0 пробелами и заканчивает работу с пустой лентой.

На рис. 8.11 представлены правила функции переходов δ, а на рис. 8.12 — ее диаграмма. Роль каждого из семи ее состояний описывается следующим образом.

Состояние	Символ 0	Символ 1	Символ B
q_0	(q_1, B, R)	(q_5, B, R)	—
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	—
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
q_6	—	—	—

Рис. 8.11. Машина Тьюринга, вычисляющая функцию усеченной разности

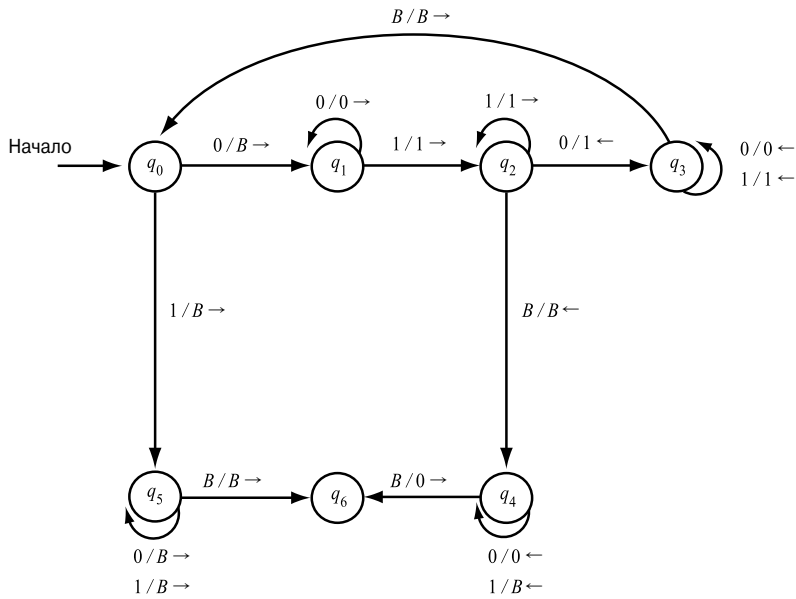


Рис. 8.12. Диаграмма переходов для МТ из примера 8.4

q_0 — данное состояние начинает цикл и прерывает его, когда нужно. Если M обозревает 0, цикл должен повториться. 0 меняется на B, головка сдвигается вправо, и M переходит в состояние q_1. Если же M обозревает 1, то все возможные соответствия между двумя группами нулей на ленте установлены, и M переходит в состояние q_5 для опустошения ленты.

q_1 — в этом состоянии M пропускает начальный блок из 0 в поисках 1. Найдя ее, M переходит в состояние q_2.

q_2 — M движется вправо, пропуская группу из 1 до появления 0. 0 меняется на 1, головка сдвигается влево, и M переходит в состояние q_3. Однако возможно также, что после блока из единиц символов 0 уже не осталось. Тогда M в состоянии q_2 обнаруживает B. Возникает ситуация 1, описанная выше, где n нулей второго блока использованы для удаления n из m нулей первого блока, и вычитание почти закончено. M переходит в состояние q_4, предназначенное для преобразования всех 1 в пробелы, а одного пробела — в 0.

q_3 — M движется влево, пропуская 0 и 1 до появления пробела. Тогда M сдвигается вправо и возвращается в состояние q_0, начиная новый цикл.

q_4 — вычитание закончено, но один лишний 0 из первого блока был ошибочно заменен пробелом. M движется влево, заменяя все 1 пробелами, до появления B. Последний меняется на 0, и M переходит в состояние q_6, где и останавливается.

q_5 — это состояние достигается из q_0, если обнаруживается, что все 0 из первого блока заменены пробелами. В случае, описанном выше в 2, усеченная разность равна 0. M заменяет все оставшиеся 0 и 1 пробелами и переходит в состояние q_6.

q_6 — единственной целью этого состояния является разрешить M остановиться после выполнения работы. Если бы вычисление разности было подпрограммой другой, более сложной функции, то q_6 начинало бы следующий шаг этого более объемного вычисления.

□

8.2.5. Язык машины Тьюринга

Способ допускания языка машиной Тьюринга уже описан интуитивно. Входная цепочка помещается на ленту, и головка машины начинает работу на крайнем слева символе. Если МТ в конце концов достигает допускающего состояния, то вход допускается, в противном случае — нет.

Более формально, пусть $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ — машина Тьюринга. Тогда $L(M)$ представляет собой множество цепочек w из Σ^*, для которых $q_0 w \overset{*}{\vdash} \alpha p \beta$ при некотором состоянии p из F и произвольных ленточных цепочках α и β. Это определение было принято при обсуждении машины Тьюринга в примере 8.2, допускавшей цепочки вида 0^n1^n.

Языки, допустимые с помощью машин Тьюринга, часто называются *рекурсивно перечислимыми* , или РП-языками. Термин “рекурсивно перечислимые” происходит от вы-

числительных формализмов, предшествовавших машинам Тьюринга, но определявших тот же класс языков или арифметических функций. Обсуждение источников этого термина представлено во врезке в разделе 9.2.1.

Соглашения по обозначениям машин Тьюринга

Обычно для машин Тьюринга используются такие же символы, как и для других видов автоматов.

1. Строчные буквы из начала алфавита обозначают входные символы.
2. Прописные буквы, обычно из конца алфавита, используются для ленточных символов, которые иногда могут быть и входными. Однако для пробела всегда используется B.
3. Строчные буквы из конца алфавита обозначают цепочки входных символов.
4. Греческие буквы используются для цепочек ленточных символов.
5. Буквы p, q и другие около них в алфавите обозначают состояния.

8.2.6. Машины Тьюринга и останов

Существует еще одно понятие “допустимости” для машин Тьюринга — допустимость по останову. Говорят, что машина Тьюринга *останавливается*, если попадает в состояние q, обозревая ленточный символ X, и в этом положении нет переходов, т.е. $\delta(q, X)$ не определено.

Пример 8.5. Машина Тьюринга M из примера 8.4 была построена для того, чтобы допускать язык; она не рассматривалась с точки зрения вычисления функции. Заметим, однако, что M останавливается на всех цепочках из символов 0 и 1, поскольку независимо от входной цепочки машина M в конце концов удаляет вторую группу нулей[6], если может ее найти, достигает состояния q_6 и останавливается. □

Всегда можно предполагать, что МТ останавливается, если допускает. Таким образом, без изменения допускаемого языка можно сделать $\delta(q, X)$ неопределенной, если q — допускающее состояние. Итак,

- предполагается, что МТ всегда останавливается в допускающем состоянии.

К сожалению, не всегда можно потребовать, чтобы МТ останавливалась, если не допускает. Язык машины Тьюринга, которая в конце концов останавливается независимо от того, допускает она или нет, называется *рекурсивным*, и его важные свойства будут рассматриваться, начиная с раздела 9.2.1. Машина Тьюринга, которая всегда останавливается, представляет собой хорошую модель “алгоритма”. Если алгоритм решения данной проблемы существует, то проблема называется “разрешимой”, поэтому машины

[6] Точнее, удаляется все, что находится после крайней слева группы нулей, возможно, усеченной. — *Прим. ред.*

Тьюринга, которые всегда останавливаются, имеют большое значение во введении в теорию разрешимости (см. главу 9).

8.2.7. Упражнения к разделу 8.2

8.2.1. Укажите конфигурации МТ (рис. 8.9) при обработке следующего входа:

а) (*) 00;

б) 000111;

в) 00111.

8.2.2. (!) Постройте машины Тьюринга для следующих языков:

а) (*) множество цепочек с одинаковыми количествами символов 0 и 1;

б) $\{a^n b^n c^n \mid n \geq 1\}$;

в) $\{ww^R \mid w$ — произвольная цепочка из символов 0 и 1$\}$.

8.2.3. Постройте машину Тьюринга, которая на вход получает натуральное число N и добавляет к нему 1 в двоичной записи. Точнее, изначально на ленте стоит знак \$, за которым записано N в двоичном виде. Вначале головка в состоянии q_0 обозревает \$. Ваша машина должна остановиться с двоичной записью $N + 1$ на ленте, обозревая ее крайний слева символ и находясь в состоянии q_f. При необходимости можно удалить \$, например, $q_0\$10011 \overset{*}{\vdash} \$q_f10100$ или $q_0\$11111 \overset{*}{\vdash} q_f100000$:

а) укажите переходы вашей МТ и объясните назначение каждого состояния;

б) укажите последовательность МО вашей МТ при обработке входа \$111.

8.2.4. (!*) В этом упражнении устанавливается эквивалентность вычисления функций и распознавания языков для машин Тьюринга. Для простоты рассматриваются только функции из множества неотрицательных целых чисел в множество неотрицательных целых чисел, но идеи этой задачи применимы к любым вычислимым функциям. Рассмотрим два основных определения.

- *Графиком* функции f называется множество всех цепочек вида $[x, f(x)]$, где x — неотрицательное целое число в двоичной записи, а $f(x)$ — значение функции f на аргументе x (также двоичное).
- Говорят, что машина Тьюринга *вычисляет* функцию f, если, начиная с двоичной записи произвольного неотрицательного целого x на ленте, она останавливается (в любом состоянии) с двоичным $f(x)$.

С помощью неформальных, но четких конструкций выполните следующее:

а) покажите, как по данной МТ, вычисляющей f, построить МТ, которая допускает график f в качестве языка;

б) покажите, как по данной МТ, допускающей график f, построить МТ, которая вычисляет f;

в) функция называется *частичной*, если она может быть неопределенной для некоторых аргументов. Распространяя идеи этого упражнения на частичные функции, мы не требуем, чтобы МТ, вычисляющая f, останавливалась, если ее вход x — одно из чисел, для которых $f(x)$ не определена. Работают ли ваши конструкции для пунктов *а* и *б*, если функция f частична? Если нет, объясните, как их нужно изменить, чтобы они работали.

8.2.5. Рассмотрим машину Тьюринга

$M = (\{q_0, q_1, q_2, q_f\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \{q_f\})$.

Неформально, но четко опишите язык $L(M)$, если δ состоит из следующего множества правил:

а) (*) $\delta(q_0, 0) = (q_1, 1, R)$; $\delta(q_1, 1) = (q_0, 0, R)$; $\delta(q_1, B) = (q_f, B, R)$;

б) $\delta(q_0, 0) = (q_1, B, R)$; $\delta(q_0, 1) = (q_1, B, R)$; $\delta(q_1, 1) = (q_1, B, R)$; $\delta(q_1, B) = (q_f, B, R)$;

в) (!) $\delta(q_0, 0) = (q_1, 1, R)$; $\delta(q_1, 1) = (q_2, 0, L)$; $\delta(q_2, 1) = (q_0, 1, R)$; $\delta(q_1, B) = (q_f, B, R)$.

8.3. Техника программирования машин Тьюринга

Наша цель — обосновать, что машину Тьюринга можно использовать для вычислений так же, как и обычный компьютер. В конечном счете, мы хотим убедить вас, что МТ равна по своей мощи обычному компьютеру. В частности, она может выполнять некоторые вычисления, имея на входе другие машины Тьюринга, подобно тому, как описанная в разделе 8.1.2 программа проверяла другие программы. Именно это свойство “интроспективности” как машин Тьюринга, так и компьютерных программ позволяет доказывать неразрешимость проблем.

Для иллюстрации возможностей МТ представим многочисленные приемы интерпретации ленты и конечного управления машины Тьюринга. Ни один из этих приемов не расширяет базовую модель МТ; они лишь делают запись более удобной. В дальнейшем они используются для имитации расширенных моделей машин Тьюринга с дополнительными свойствами, например, с несколькими лентами, на базовой модели МТ.

8.3.1. Память в состоянии

Конечное управление можно использовать не только для представления позиции в “программе” машины Тьюринга, но и для хранения конечного объема данных. Рис. 8.13 иллюстрирует этот прием, а также идею многодорожечной ленты. При этом конечное управление содержит не только “управляющее” состояние q но и три элемента данных A, B и C. Данная техника не требует никакого расширения модели МТ; мы просто рассматриваем состоя-

ние в виде кортежа. На рис. 8.13 состояние имеет вид $[q, A, B, C]$. Такой подход позволяет описывать переходы более систематично, что зачастую проясняет программу МТ.

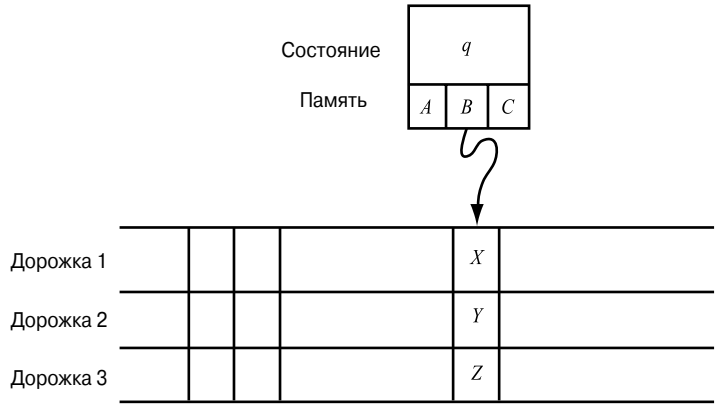


Рис. 8.13. Машина Тьюринга с памятью в конечном управлении и несколькими дорожками

Пример 8.6. Построим МТ

$$M = (Q, \{0, 1\}, \{0, 1, B\}, \delta, [q_0, B], B, \{[q_1, B]\}),$$

которая запоминает в своем конечном управлении первый увиденный символ (0 или 1) и проверяет, не встречается ли он еще где-нибудь во входной цепочке. Таким образом, M допускает язык 01^*+10^*. Допускание регулярных языков (вроде данного) не сужает возможностей машин Тьюринга, а служит лишь простым примером.

Множество состояний Q есть $\{q_0, q_1\} \times \{0, 1, B\}$, т.е. состояния рассматриваются как пары из двух следующих компонентов.

1. Управляющая часть (q_0 или q_1) запоминает, что делает МТ. Управляющее состояние q_0 сигнализирует о том, что M еще не прочитала свой первый символ, а q_1 — что *уже прочитала*, и проверяет, не встречается ли он где-нибудь еще, продвигаясь вправо до достижения пустой клетки.
2. В части данных хранится первый увиденный символ (0 или 1). Пробел B в этом компоненте означает, что никакой символ еще не прочитан.

Функция переходов δ определена следующим образом.

1. $\delta([q_0, B], a) = ([q_1, a], a, R)$ для $a = 0$ или $a = 1$. Вначале управляющим состоянием является q_0, а частью данных — B. Обозреваемый символ копируется во второй компонент состояния, и M сдвигается вправо, переходя при этом в управляющее состояние q_1.
2. $\delta([q_1, a], \bar{a}) = ([q_1, a], \bar{a}, R)$, где $\bar{a}$ — "дополнение" a, т.е. 0 при $a = 1$ и 1 при $a = 0$. В состоянии q_1 машина M пропускает каждый символ 0 или 1, который отличается от хранимого в состоянии, и продолжает движение вправо.

3. $\delta([q_1, a], B) = ([q_1, B], B, R)$ для $a = 0$ или $a = 1$. Достигая первого пробела, M переходит в допускающее состояние $[q_1, B]$.

Заметим, что M не имеет определения переходов $\delta([q_1, a], B)$ для $a = 0$ или $a = 1$. Таким образом, если M обнаруживает второе появление символа, который был записан в ее память конечного управления, она останавливается, не достигнув допускающего состояния. □

8.3.2. Многодорожечные ленты

Еще один полезный прием состоит в том, чтобы рассматривать ленту МТ как образованную несколькими дорожками. Каждая дорожка может хранить один символ (в одной клетке), и алфавит МТ состоит из кортежей, с одним компонентом для каждой "дорожки". Например, клетка, обозреваемая ленточной головкой на рис. 8.13, содержит символ $[X, Y, Z]$. Как и память в конечном управлении, множественные дорожки не расширяют возможностей машин Тьюринга. Это просто описание полезной структуры ленточных символов.

Пример 8.7. Типичное использование многодорожечных лент состоит в том, что одна дорожка хранит данные, а другая — отметку. Можно отмечать каждый символ, использованный определенным образом, или небольшое число позиций в данных. Данный прием фактически применялся в примерах 8.2 и 8.4, хотя лента там и не рассматривалась явно как многодорожечная. В данном примере вторая дорожка используется явно для распознавания следующего языка, который не является контекстно-свободным.

$$L_{wcw} = \{wcw \mid w \text{ принадлежит } (\mathbf{0}+\mathbf{1})^+\}$$

Построим машину Тьюринга

$$M = (Q, \Sigma, \Gamma, \delta, [q_0, B], [B, B], \{[q_9, B]\}),$$

компоненты которой имеют следующий смысл.

Q — множество состояний, которое представляет собой $\{q_1, q_2, \ldots, q_9\} \times \{0, 1, B\}$, т.е. множество пар, состоящих из управляющего состояния q_i и компонента данных 0, 1 или B. Вновь используется разрешенное выше запоминание символа в конечном управлении.

Γ — множество ленточных символов $\{B, *\} \times \{0, 1, c, B\}$. Первый компонент, т.е. дорожка, может быть пустым или "отмеченным", что представлено, соответственно, пробелом или *. Символ * используется для отметки символов первой и второй групп из 0 и 1, в итоге подтверждающей, что цепочки слева и справа от центрального маркера c совпадают. Второй компонент ленточного символа представляет то, чем как бы является сам по себе ленточный символ, т.е. $[B, X]$ рассматривается как ленточный символ X для $X = 0, 1, c, B$.

Σ — входными символами являются $[B, 0]$ и $[B, 1]$, которые, как только что указано, обозначают, соответственно, 0 и 1.

δ — функция переходов определена следующими правилами, в которых a и b могут обозначать как 0, так и 1.

1. $\delta([q_1, B], [B, a]) = ([q_2, a], [*, a], R)$. В начальном состоянии M берет символ a (которым может быть 0 или 1), запоминает его в конечном управлении и переходит в управляющее состояние q_2. Затем “отмечает” обозреваемый символ, изменив его первый компонент с B на *, и сдвигается вправо.
2. $\delta([q_2, a], [B, b]) = ([q_2, a], [B, b], R)$. M движется вправо в поисках символа c. Напомним, что a и b независимо друг от друга могут быть 0 или 1, но не могут быть c.
3. $\delta([q_2, a], [B, c]) = ([q_3, a], [B, c], R)$. Обнаружив c, M продолжает двигаться вправо, но меняет управляющее состояние на q_3.
4. $\delta([q_3, a], [*, b]) = ([q_3, a], [*, b], R)$. В состоянии q_3 продолжается пропуск всех отмеченных символов.
5. $\delta([q_3, a], [B, a]) = ([q_4, B], [*, a], L)$. Если первый неотмеченный символ, найденный M, совпадает с символом в ее управлении, она отмечает его, поскольку он является парным к соответствующему символу из первого блока нулей и единиц. M переходит в управляющее состояние q_4, выбрасывая символ из управления, и начинает движение влево.
6. $\delta([q_4, B], [*, a]) = ([q_4, B], [*, a], L)$. На отмеченных символах M движется влево.
7. $\delta([q_4, B], [B, c]) = ([q_5, B], [B, c], L)$. Обнаружив символ c, M переходит в состояние q_5 и продолжает движение влево. В состоянии q_5 она должна принять решение, зависящее от того, отмечен или нет символ непосредственно слева от c. Если отмечен, то первый блок из нулей и единиц уже полностью рассмотрен. Если же символ слева от c не отмечен, то M ищет крайний слева неотмеченный символ, запоминает его, и после этого в состоянии q_1 начинается новый цикл.
8. $\delta([q_5, B], [B, a]) = ([q_6, B], [B, a], L)$. Если символ слева от c не отмечен, начинается соответствующая ветка вычислений. M переходит в состояние q_6 и продолжает движение влево в поисках отмеченного символа.
9. $\delta([q_6, B], [B, a]) = ([q_6, B], [B, a], L)$. Пока символы не отмечены, M остается в состоянии q_6 и движется влево.
10. $\delta([q_6, B], [*, a]) = ([q_1, B], [B, a], R)$. Обнаружив отмеченный символ, M переходит в состояние q_1 и движется вправо, чтобы взять первый неотмеченный символ.
11. $\delta([q_5, B], [*, a]) = ([q_7, B], [*, a], R)$. Теперь рассмотрим ветку вычислений, когда в состоянии q_5 непосредственно слева от c обнаружен отмеченный символ. Начинается движение вправо в состоянии q_7.
12. $\delta([q_7, B], [B, c]) = ([q_8, B], [B, c], R)$. В состоянии q_7 обозревается c. Происходит переход в состояние q_8 и продолжается движение вправо.
13. $\delta([q_8, B], [*, a]) = ([q_8, B], [*, a], R)$. В состоянии q_8 машина движется вправо, пропуская все отмеченные символы.

14. $\delta([q_8, B], [B, B]) = ([q_9, B], [B, B], R)$. Если M достигает пробела в состоянии q_8, не обнаружив ни одного неотмеченного символа, то она допускает. Если же она сначала находит неотмеченный символ 0 или 1, то блоки слева и справа от c не совпадают, и M останавливается без допускания.

□

8.3.3. Подпрограммы

Машины Тьюринга — это программы, и их полезно рассматривать как построенные из набора взаимодействующих компонентов, или "подпрограмм". Подпрограмма машины Тьюринга представляет собой множество состояний, выполняющее некоторый полезный процесс. Это множество включает в себя стартовое состояние и еще одно состояние, которое не имеет переходов и служит состоянием "возврата" для передачи управления какому-либо множеству состояний, вызвавшему данную подпрограмму. "Вызов" подпрограммы возникает везде, где есть переход в ее начальное состояние. Машины Тьюринга не имеют механизма для запоминания "адреса возврата", т.е. состояния, в которое нужно перейти после завершения подпрограммы. Поэтому для реализации вызовов одной и той же МТ из нескольких состояний можно создавать копии подпрограммы, используя новое множество состояний для каждой копии. "Вызовы" ведут в начальные состояния разных копий подпрограммы, и каждая копия "возвращает" в соответствующее ей состояние.

Пример 8.8. Построим МТ для реализации функции "умножение". Наша МТ начинает с 0^m10^n1 на ленте и заканчивает с 0^{mn}. Опишем вкратце ее работу.

1. В начале каждого из m циклов работы лента содержит одну непустую цепочку вида $0^i10^n10^{kn}$ для некоторого k [7].
2. За один цикл 0 из первой группы меняется на B, и n нулей добавляются к последней группе, приводя к цепочке вида $0^{i-1}10^n10^{(k+1)n}$.
3. В результате группа из n нулей копируется m раз с изменением каждый раз одного 0 в первой группе на B. Когда первая группа нулей целиком превратится в пробелы, в последней группе будет mn нулей.
4. Заключительный шаг — заменить пробелами 10^n1 в начале, и работа закончена.

"Сердцем" этого алгоритма является подпрограмма, которая называется `Copy`. Она реализует шаг 2, копируя блок из n нулей в конец. Точнее, `Copy` преобразует МО вида $0^{m-k}1q_10^n10^{(k-1)n}$ в МО $0^{m-k}1q_50^n10^{kn}$. Переходы подпрограммы `Copy` представлены на рис. 8.14. Она заменяет первый 0 маркером X, в состоянии q_2 движется вправо до появления пробела, копирует в эту клетку 0, и в состоянии q_3 движется влево, пока не появится маркер X. На маркере она переходит вправо и возвращается в q_1. Она повторяет данный цикл до тех пор, пока в состоянии q_1 не встретит 1 вместо 0. Тогда она использует состояние q_4 для обратной замены маркеров X нулями и заканчивает в состоянии q_5.

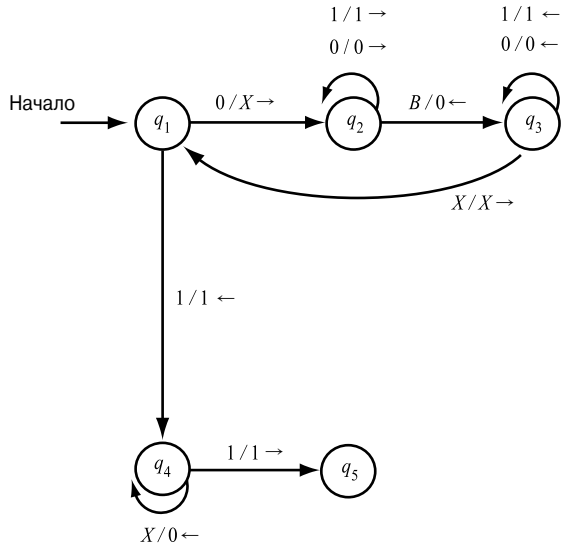


Рис. 8.14. Подпрограмма `Copy`

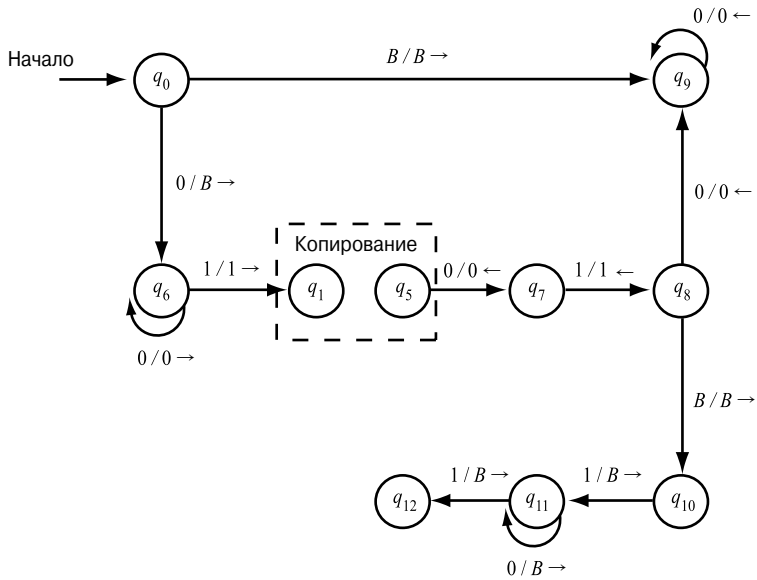


Рис. 8.15. Полная программа умножения использует подпрограмму `Copy`

Вся машина Тьюринга для умножения начинает в состоянии q_0. Вначале она за несколько шагов переходит от МО $q_0 0^m 1 0^n 1$ к МО $0^{m-1} 1 q_1 0^n 1$. Необходимые переходы показаны на рис. 8.15 слева от вызова подпрограммы; в них участвуют только состояния q_0 и q_6.

---

[7] Перед первым циклом $i = m$ и $k = 0$. — *Прим. ред.*

Справа от вызова подпрограммы (см. рис. 8.15) представлены состояния q_7–q_{12}. Состояния q_7, q_8 и q_9 предназначены для получения управления после того, как Copy скопировала блок из n нулей и находится в МО $0^{m-k}1q_50^n10^{kn}$. Эти состояния приводят к конфигурации $q_00^{m-k}10^n0^{kn}$. В этот момент опять начинается цикл, удаляется крайний слева 0 и вызывается Copy для нового копирования блока из n нулей.

Как исключение, в состоянии q_8 МТ может обнаружить, что все m нулей заменены пробелами, т.е. $k = m$. В данном случае производится переход в состояние q_{10}. Это состояние с помощью состояния q_{11} заменяет символы 10^n1 в начале ленты пробелами, после чего достигается состояние останова q_{12}. В этот момент МТ имеет МО $q_{12}0^{mn}$, и ее работа завершена. □

8.3.4. Упражнения к разделу 8.3

8.3.1. (!) Переделайте ваши машины Тьюринга из упражнения 8.2.2, используя преимущества техники программирования, обсуждаемой в данном разделе.

8.3.2. (!) Обычные операции в программах типа машин Тьюринга используют “сдвиг” (“shifting over”). Предположим, в текущей позиции головки нужно создать дополнительную клетку, в которую можно было бы записать некоторый символ. Однако изменять ленту таким способом нельзя. Придется сдвинуть содержимое ленты справа от головки на одну клетку вправо и затем вернуться к текущей позиции. Покажите, как выполнить данную операцию. *Указание*. Зарезервируйте специальный символ для отметки позиции, в которую нужно вернуться.

8.3.3. (∗) Постройте подпрограмму перемещения головки МТ из ее текущей позиции вправо с пропуском нулей до достижения первой 1 или пробела. Если текущая позиция не содержит 0, то МТ останавливается. Можно предполагать, что ленточными символами являются только 0, 1 и B (пробел). Используйте полученную подпрограмму для построения МТ, допускающей все цепочки из 0 и 1, в которых нет двух 1 подряд.

8.4. Расширения базовой машины Тьюринга

В этом разделе представлены некоторые вычислительные модели, связанные с машинами Тьюринга и имеющие такую же мощность (в смысле распознавания языков), что и базовая модель МТ, с которой мы работали до сих пор. Одна из них, многоленточная МТ, позволяет легко имитировать работу компьютера или других видов машин Тьюринга. И хотя многоленточные машины не мощнее базовой модели, речь также пойдет об их способности допускать языки.

Затем рассматриваются недетерминированные машины Тьюринга — расширение основной модели, в котором разрешается совершать любой из конечного множества переходов в данной ситуации. Это расширение в некоторых случаях также облегчает “прог-

раммирование" машин Тьюринга, не добавляя ничего к распознавательной мощности базовой модели.

8.4.1. Многоленточные машины Тьюринга

Многоленточная машина Тьюринга представлена на рис. 8.16. Устройство имеет конечное управление (состояния) и некоторое конечное число лент. Каждая лента разделена на клетки, и каждая клетка может содержать любой символ из конечного ленточного алфавита. Как и у одноленточных МТ, множество ленточных символов включает пробел, а также имеет подмножество входных символов, к числу которых пробел не принадлежит. Среди состояний есть начальное и допускающие. Начальная конфигурация такова.

1. Вход (конечная последовательность символов) размещен на первой ленте.
2. Все остальные клетки всех лент содержат пробелы.
3. Конечное управление находится в начальном состоянии.
4. Головка первой ленты находится в левом конце входа.
5. Головки всех других лент занимают произвольное положение. Поскольку все ленты, кроме первой, пусты, начальное положение головок на них не имеет значения (все клетки "выглядят" одинаково).

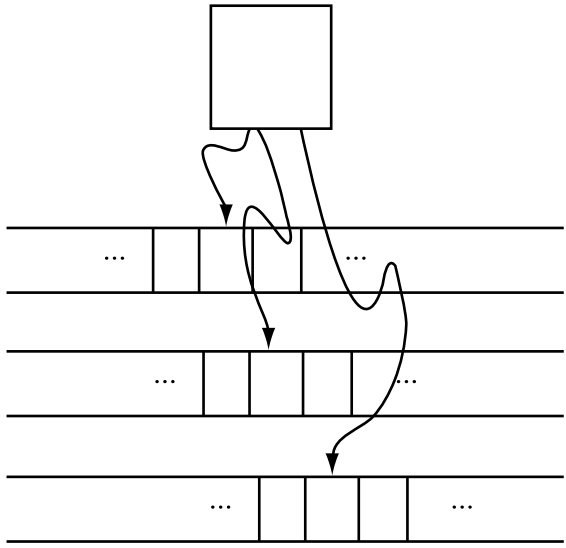


Рис. 8.16. Многоленточная машина Тьюринга

Переход многоленточной МТ зависит от состояния и символа, обозреваемого каждой головкой. За один переход многоленточная МТ совершает следующие действия.

1. Управление переходит в новое состояние, которое может совпадать с предыдущим.

2. На каждой ленте в обозреваемую клетку записывается новый символ. Любой из них может совпадать с символом, бывшим там раньше.
3. Каждая из ленточных головок сдвигается влево или вправо или остается на месте. Головки сдвигаются независимо друг от друга, поэтому разные головки могут двигаться в разных направлениях, а некоторые вообще оставаться на месте.

Формальная запись переходов не приводится — ее вид является непосредственным обобщением записи для одноленточной МТ, за исключением того, что сдвиги теперь обозначаются буквами *L*, *R* или *S*. Возможность оставлять головку на месте не была предусмотрена для одноленточных МТ, поэтому в их записи не было *S*. Постарайтесь сами представить подходящую запись МО (конфигураций) многоленточных МТ; формально она здесь также не приводится. Многоленточные МТ, как и одноленточные, допускают, попадая в допускающее состояние.

8.4.2. Эквивалентность одноленточных и многоленточных машин Тьюринга

Напомним, что рекурсивно перечислимые языки определяются как языки, допускаемые одноленточными МТ. Очевидно, что многоленточные МТ допускают все рекурсивно перечислимые языки, поскольку одноленточная МТ является частным случаем многоленточной. Но существуют ли не рекурсивно перечислимые языки, допускаемые многоленточными МТ? Ответом является “нет”, и мы докажем это, показав, как многоленточная МТ имитируется с помощью одноленточной.

Теорема 8.9. Каждый язык, допускаемый многоленточной МТ, рекурсивно перечислим.

Доказательство. Идею доказательства иллюстрирует рис. 8.17. Предположим, что язык *L* допускается *k*-ленточной МТ *M*. Она имитируется с помощью одноленточной МТ *N*, лента которой состоит из 2*k* дорожек. Половина этих дорожек содержит ленты машины *M*, а каждая из дорожек второй половины содержит один-единственный маркер, указывающий позицию, в которой в данный момент времени находится головка соответствующей ленты. Рис. 8.17 соответствует тому, что $k = 2$. Вторая и четвертая дорожки хранят содержимое первой и второй лент машины *M*, дорожка 1 хранит позицию головки на ленте 1, а дорожка 3 — позицию на ленте 2.

Для имитации перехода МТ *M* головка МТ *N* должна посетить *k* головочных маркеров. Чтобы не “заблудиться”, *N* должна помнить, сколько маркеров находятся слева от ее головки каждый раз; этот учет ведется с помощью компонента конечного управления *N*. После посещения каждого головочного маркера и запоминания обозреваемого символа в компоненте управления *N* знает, какие символы обозреваются каждой из головок *M*. *N* также знает состояние *M*, которое запоминается в конечном управлении *N*. Таким образом, *N* знает, какой переход сделает *M*.

Теперь *N* еще раз посещает каждый из головочных маркеров на своей ленте, изменяет символ в дорожке, представляющей соответствующую ленту, и при необходимости пе-

ремещает головочные маркеры влево или вправо. Наконец, N изменяет состояние M, записанное в конечном управлении N. В этот момент N проимитировала один переход M.

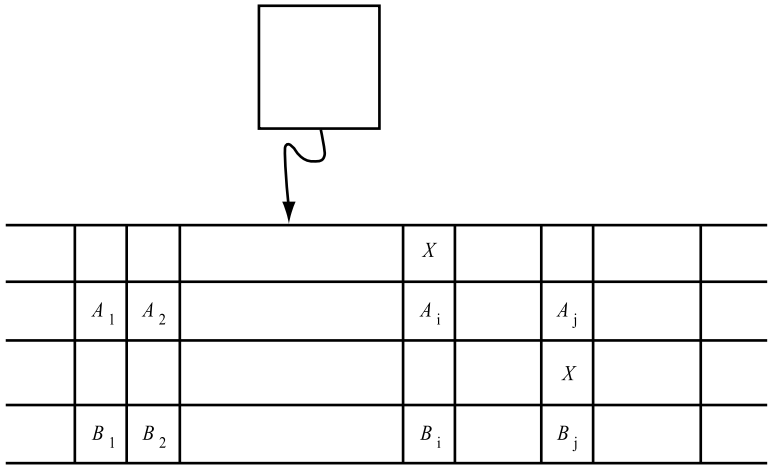


Рис. 8.17. Имитация двухленточной МТ с помощью одноленточной

В качестве допускающих состояний N выбираются все те ее состояния, в которых запомнено допускающее состояние M. Таким образом, как только имитируемая M допускает, N также допускает, и не допускает в противном случае. □

Напоминание о конечности

Обычное заблуждение — путать значение, конечное в каждый момент времени, с конечным множеством значений. Конструкция сведения многих лент к одной помогает оценить разницу между этими "конечностями". В данной конструкции использованы дорожки на ленте для запоминания позиций ленточных головок. Почему нельзя записать эти позиции в виде целых чисел в конечное управление? Будучи небрежным, кто-то мог бы утверждать, что после n переходов головки МТ находятся в позициях, отличающихся не более, чем на n от начальной, и поэтому достаточно записать целые не больше n.

Проблема состоит в том, что, хотя позиции конечны в каждый момент времени, все множество позиций может быть и бесконечным. Если состояние должно представлять любую позицию головки, то в состоянии должен быть компонент данных, имеющий любое целое в качестве значения. Из-за этого компонента множество состояний должно быть бесконечным, даже если только конечное число состояний используется в любой конечный момент времени. Определение же машин Тьюринга требует, чтобы множество состояний было конечным. Таким образом, запомнить позицию ленточной головки в конечном управлении нельзя.

8.4.3. Время работы и конструкция „много лент к одной“

Здесь представлено понятие, которое в дальнейшем окажется чрезвычайно важным, а именно: “временн*ая* сложность”, или “время работы” машины Тьюринга. *Временем работы* машины Тьюринга на входе w называется число шагов, которые M совершает до останова. Если M не останавливается на w, то время работы бесконечно. *Временная сложность* МТ M есть функция $T(n)$, которая представляет собой максимум времени работы M на w по всем входам w длины n. Для МТ, не останавливающихся на всех своих входах, $T(n)$ может быть бесконечным для некоторых или даже для всех n. Однако особое внимание будет уделено машинам Тьюринга, которые обязательно останавливаются на всех своих входах, и в частности, тем машинам, которые имеют полиномиальную временн*ую* сложность. Они будут изучаться, начиная с раздела 10.1.

Конструкция теоремы 8.9 кажется неуклюжей. Действительно, построенной одноленточной МТ может понадобится гораздо больше времени для работы, чем многоленточной. Однако время работы этих двух машин соразмерно в том смысле, что время работы одноленточной МТ есть не более чем квадрат времени работы многоленточной. Хотя “возведение в квадрат” не является хорошей соразмерностью, оно сохраняет свойство времени работы быть полиномиальным. В главе 10 будут представлены следующие факты.

1. Разница между полиномиальным временем и более высокими степенями его возрастания — это в действительности граница между тем, что можно решить с помощью компьютера, и тем, что практически нерешаемо.
2. Несмотря на обширные исследования, время, необходимое для решения многих проблем, не удается определить точнее, чем “некоторое полиномиальное”. Таким образом, когда изучается время, необходимое для решения некоторой проблемы, использование многоленточной или одноленточной МТ не является критичным.

Докажем, что время работы многоленточной МТ является не более чем квадратом времени работы одноленточной МТ.

Теорема 8.10. Время, необходимое одноленточной МТ N для имитации n переходов k-ленточной МТ M, есть $O(n^2)$.

Доказательство. После n переходов машины M головочные маркеры не могут быть разделены более, чем $2n$ клетками. Таким образом, если N стартует на крайнем слева маркере, ей нужно сдвинуться не более, чем на $2n$ клеток вправо, чтобы найти все головочные маркеры. Затем ей нужно совершить проход влево, изменяя содержимое лент M и сдвигая головочные маркеры вправо или влево, если необходимо. Выполнение этого требует не более $2n$ сдвигов влево, плюс не более $2k$ переходов для изменения направления движения и записи маркера X в клетку справа (когда головка M сдвигается вправо).

Таким образом, число переходов N, необходимых для имитации одного из первых n переходов, не более $4n + 2k$. Поскольку k — константа, не зависящая от числа имитируемых переходов, это чиñло переходов есть $O(n)$. Имитация n переходов требует времени не более, чем в n раз больше, т.е. $O(n^2)$.

8.4.4. Недетерминированные машины Тьюринга

Недетерминированная машина Тьюринга (*НМТ*) отличается от изученных нами детерминированных тем, что ее $\delta(q, X)$ для каждого состояния q и ленточного символа X представляет собой множество троек

$$\{(q_1, Y_1, D_1), (q_2, Y_2, D_2), \ldots, (q_k, Y_k, D_k)\},$$

где k — натуральное число. НМТ может выбирать на каждом шаге любую из троек для следующего перехода. Она не может, однако, выбрать состояние из одной тройки, ленточный символ из другой, а направление из какой-нибудь еще.

Язык, допускаемый НМТ M, определяется по аналогии с другими недетерминированными устройствами, вроде НКА или МП-автоматов. Таким образом, M допускает вход w, если существует некоторая последовательность выборов переходов, ведущая из начального МО с w на входе в МО с допускающим состоянием. Существование других выборов, которые *не ведут* в допускающее состояние, не имеет значения, как и для НКА или МП-автоматов.

Недетерминированные МТ допускают те же языки, что и детерминированные (или *ДМТ*, если нам нужно подчеркнуть их детерминированность). Доказательство основано на том, что для любой НМТ M_N можно построить ДМТ M_D, которая исследует конфигурации, достигаемые M_N с помощью любой последовательности переходов. Если M_D находит хотя бы одно МО с допускающим состоянием, то сама переходит в допускающее состояние. M_D должна помещать новые МО в очередь, а не в магазин, чтобы после некоторого конечного времени были проимитированы все последовательности длиной до k ($k = 1, 2, \ldots$).

Теорема 8.11. Если M_N — недетерминированная машина Тьюринга, то существует детерминированная машина Тьюринга M_D, у которой $L(M_D) = L(M_N)$.

Доказательство. Построим M_D в виде многоленточной МТ, общий вид которой представлен на рис. 8.18. Первая лента M_D хранит последовательность МО M_N, включая состояние M_N. Одно МО M_N отмечено как “текущее”; следующие за ним МО должны быть исследованы после него. На рис. 8.18 третье МО отмечено символом x вместе с разделителем МО — символом *. Все МО слева от текущего уже исследованы, и в дальнейшем их можно игнорировать.

Для обработки текущего МО машина M_D совершает следующие действия.

1. M_D проверяет состояние и обозреваемый символ текущего МО. Информация о том, какие выборы перехода есть у M_N для каждого состояния и символа, хранится в конечном управлении M_D. Если состояние в текущем МО является допускающим, то M_D допускает и прекращает имитацию.

2. Если состояние в текущем МО не допускающее, а пара состояние-символ имеет k переходов, то M_D использует свою вторую ленту для копирования МО и создания k копий этого МО в конце очереди МО на ленте 1.
3. M_D изменяет каждое из этих k МО в соответствии с k различными выборами переходов, которые есть у M_N из текущего МО.
4. M_D возвращается к отмеченному (текущему) МО, удаляет отметку и перемещает ее на следующее МО справа. Цикл повторяется с шага 1.

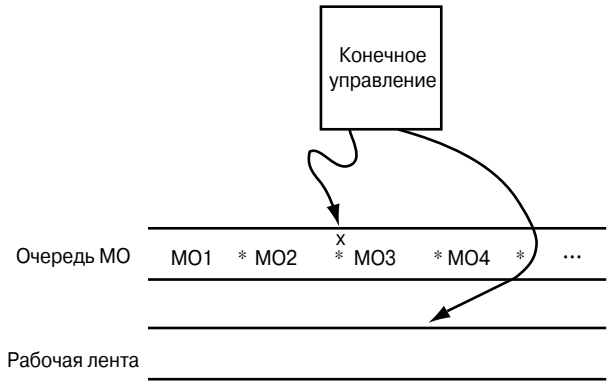


Рис. 8.18. Имитация НМТ с помощью ДМТ

Очевидно, что имитация точна в том смысле, что M_D допускает только тогда, когда находит, что M_N может перейти в допускающее МО. Однако нужно обосновать, что если M_N попадает в допускающее МО после n переходов, то M_D в конце концов породит это МО в качестве текущего и допустит.

Предположим, что m есть максимальное число выборов у M_N в любой конфигурации. Тогда существует одно начальное МО M_N, не более m МО, достижимых за 1 шаг, не более m^2 МО, достижимых за 2 шага, и т.д. Таким образом, после n переходов M_N может достичь не более $1 + m + m^2 + \ldots + m^n$ МО, что не превышает nm^n.

Порядок, в котором M_D исследует МО M_N, называется “поиск в ширину” (“breadth first”), т.е. она исследует все МО, достижимые за 0 переходов (начальное МО), затем все МО, достижимые за 1 переход, за 2 перехода, и т.д. В частности, M_D сделает текущими все МО, достижимые не более, чем за n переходов, и создаст все следующие за ними МО, перед тем, как делать текущими МО, достижимые более, чем за n переходов.

Как следствие, допускающее МО M_N рассматривается M_D в числе первых nm^n МО. Нам нужно знать лишь то, что M_D рассматривает это МО через некоторое конечное время, и этой границы достаточно, чтобы убедиться, что допускающее МО в конце концов действительно рассматривается. Таким образом, если M_N допускает, то M_D также допускает. Поскольку по построению M_D допускает только потому, что допускает M_N, можно заключить, что $L(M_D) = L(M_N)$. □

Отметим, что построенная детерминированная МТ может потребовать времени, экспоненциально большего, чем время работы недетерминированной МТ. Неизвестно, является ли это экспоненциальное соотношение необходимым. Этому вопросу и некоторым его производным, связанным с поисками лучшего способа детерминированной имитации НМТ, посвящена глава 10.

8.4.5. Упражнения к разделу 8.4

8.4.1. Опишите неформально, но четко и ясно многоленточные машины Тьюринга, допускающие один из языков, приведенных в упражнении 8.2.2. Постарайтесь построить каждую машину так, чтобы время ее работы было прямо пропорционально длине входа.

8.4.2. Недетерминированная МТ $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \{q_2\})$ представлена следующей функций переходов.

	Символ		
δ	0	1	B
q_0	$\{(q_0, 1, R)\}$	$\{(q_1, 0, R)\}$	$\varnothing$
q_1	$\{(q_1, 0, R), (q_0, 0, L)\}$	$\{(q_1, 1, R), (q_0, 1, L)\}$	$\{(q_2, B, R)\}$
q_2	$\varnothing$	$\varnothing$	$\varnothing$

Покажите, какие МО достижимы из начального МО, если входом является следующая цепочка:

а) (*) 01;

б) 001.

8.4.3. (!) Опишите неформально, но четко и ясно недетерминированные машины Тьюринга, возможно, многоленточные, которые допускают следующие языки. Постарайтесь использовать преимущества недетерминизма, чтобы избежать итераций и сократить время работы в недетерминированном смысле, т.е. лучше, чтобы ваша машина имела много разветвлений, но ветви были короткими:

а) (*) язык всех цепочек из символов 0 и 1, содержащих некоторую подцепочку длиной 100, которая повторяется, причем не обязательно подряд. Формально, данный язык есть множество цепочек из 0 и 1 вида $wxyxz$, где $|x| = 100$, а w, y и z имеют произвольную длину;

б) язык всех цепочек вида $w_1\#w_2\#\ldots\#w_n$ для произвольного n, где каждая из w_i есть цепочка из символов 0 и 1, причем для некоторого j цепочка w_j является двоичной записью числа j;

в) язык всех цепочек того же вида, что и в п. *б*, но хотя бы для двух значений j цепочки w_j представляют собой двоичную запись j.

8.4.4. (!) Рассмотрим недетерминированную машину Тьюринга

$M = (\{q_0, q_1, q_2, q_f\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \{q_f\})$.

Неформально, но ясно опишите язык $L(M)$, если δ состоит из следующих множеств правил: $\delta(q_0, 0) = \{(q_0, 1, R), (q_1, 1, R)\}$, $\delta(q_1, 1) = \{(q_2, 0, L)\}$, $\delta(q_2, 1) = \{(q_0, 1, R)\}$, $\delta(q_1, B) = \{(q_f, B, R)\}$.

8.4.5. (*) Рассмотрим недетерминированную МТ, лента которой бесконечна в обоих направлениях. В некоторый момент времени вся лента пуста, за исключением одной клетки, в которой записан символ $. Головка находится в некоторой пустой клетке, а состоянием является q:

а) напишите переходы, позволяющие НМТ перейти в состояние p, обозревая $;

б) (!) предположим, что МТ детерминирована. Как бы вы позволили ей найти $ и перейти в состояние p?

8.4.6. Постройте следующую 2-ленточную МТ, допускающую язык всех цепочек из 0 и 1, в которых этих символов поровну. Первая лента содержит вход и просматривается слева направо. Вторая лента используется для запоминания излишка нулей по отношению к единицам или наоборот в прочитанной части входа. Опишите состояния и переходы, а также неформальное предназначение каждого состояния.

8.4.7. В данном упражнении с помощью специальной 3-ленточной МТ реализуется магазин.

1. Первая лента используется только для хранения и чтения входа. Входной алфавит состоит из символа $\uparrow$, который интерпретируется, как “вытолкнуть из магазина”, и символов a и b, интерпретируемых как “поместить a (соответственно, b) в магазин”.
2. Вторая лента используется для запоминания магазина.
3. Третья лента является выходной. Каждый раз, когда из магазина выталкивается символ, он записывается на выходную ленту вслед за ранее записанными.

Машина Тьюринга должна начинать работу с пустым магазином и реализовывать последовательность операций помещения в магазин и выталкивания, заданных входом, читая его слева направо. Если вход приводит к тому, что МТ пытается вытолкнуть из пустого магазина, то она должна остановиться в специальном состоянии ошибки q_e. Если весь прочитанный вход оставляет магазин пустым, то вход допускается путем перехода в заключительное состояние q_f. Опишите неформально, но четко и ясно функцию переходов данной МТ. Вкратце опишите предназначение каждого использованного состояния.

8.4.8. На рис. 8.17 была представлена имитация k-ленточной МТ с помощью одноленточной в общем случае.

а) (*) Предположим, что данная техника использована для имитации 5-ленточной МТ, ленточный алфавит которой состоит из 7 символов. Сколько ленточных символов будет у одноленточной машины?

б) (*) Альтернативный способ имитации k лент с помощью одной состоит в использовании $(k + 1)$-й дорожки для хранения позиций головок всех k лент, причем первые k дорожек имитируют k лент очевидным образом. Заметим, что с $(k + 1)$-й дорожкой нужно быть аккуратным, чтобы различать головки и допускать возможность того, что две и более головок могут находиться в одной клетке. Сокращает ли данный метод число ленточных символов, необходимых для одноленточной МТ?

в) Еще один способ имитации k лент с помощью одной вообще не требует запоминания позиций головок. Вместо этого $(k + 1)$-я дорожка используется для отметки только одной клетки ленты. Каждая имитируемая лента все время позиционируется на своей дорожке так, что головка находится на отмеченной клетке. Если k-ленточная МТ перемещает головку на ленте i, то имитирующая одноленточная МТ смещает все непустое содержимое i-й дорожки на одну клетку в противоположном направлении, так что клетка, обозреваемая головкой i-й клетки в k-ленточной машине, остается в отмеченной клетке. Помогает ли данный метод сократить число ленточных символов одноленточной МТ? Есть ли у него недостатки по сравнению с другими описанными здесь методами?

8.4.9. (!) Машина Тьюринга, называемая *k-головочной*, имеет k головок, читающих клетки на одной ленте. Переход такой МТ зависит от состояния и символов, обозреваемых головками. За один переход эта МТ может изменить состояние, записать новый символ в клетку, обозреваемую каждой из головок, и сдвинуть каждую головку влево, вправо или оставить на месте. Поскольку несколько головок могут одновременно обозревать одну и ту же клетку, предполагается, что головки пронумерованы от 1 до k, и символ, который в действительности записывается в клетку, есть символ, записываемый головкой с наибольшим номером. Докажите, что множества языков, допускаемых k-головочными и обычными машинами Тьюринга, совпадают.

8.4.10. (!!) *Двухмерная* машина Тьюринга имеет обычное конечное управление, но ее лента представляет собой двухмерное поле из клеток, бесконечное во всех направлениях. Вход помещается в одной строке поля, с головкой, как обычно, на левом конце входа и управлением в начальном состоянии. Вход допускается, как обычно, с помощью заключительного состояния. Докажите, что множества языков, допускаемых двухмерными и обычными машинами Тьюринга, совпадают.

8.5. Машины Тьюринга с ограничениями

Мы увидели обобщения машин Тьюринга, которые в действительности не добавляют никакой мощности в смысле распознаваемых языков. Теперь рассмотрим несколько примеров ограничений МТ, которые также не изменяют их распознавательной мощности. Первое ограничение невелико, но полезно во многих конструкциях, которые мы увидим позже: бесконечная в обе стороны лента заменяется бесконечной только вправо. Ограниченным МТ запрещается также записывать пробел вместо других ленточных символов. Это ограничение позволяет считать, что МО состоят только из значащих символов и всегда начинаются левым концом ленты.

Затем исследуются определенные виды многоленточных МТ, которые обобщают МП-автоматы. Во-первых, ленты МТ ограничиваются и ведут себя, как магазины. Во-вторых, ленты ограничиваются еще больше, становясь "счетчиками", т.е. они могут представлять лишь одно целое число, а МТ имеет возможность только отличать 0 от любого ненулевого числа. Значение этих ограничений в том, что существует несколько очень простых разновидностей автоматов, обладающих всеми возможностями компьютеров. Более того, неразрешимые проблемы, связанные с машинами Тьюринга и описанные в главе 9, в равной мере относятся и к этим простым машинам.

8.5.1. Машины Тьюринга с односторонними лентами

Мы допускали, что ленточная головка машины Тьюринга может находиться как слева, так и справа от начальной позиции, однако достаточно того, что головка может находиться на ней или только справа от нее. В действительности, можно считать, что лента является *бесконечной в одну сторону*, или *односторонней* (semi-infinite), т.е. слева от начальной позиции головки вообще нет клеток. В следующей теореме приводится конструкция, показывающая, что МТ с односторонней лентой может имитировать обычную МТ с бесконечной в обе стороны лентой.

В основе этой конструкции лежит использование двух дорожек на односторонней ленте. Верхняя дорожка представляет клетки исходной МТ, находящиеся справа от начальной позиции, и ее саму. Нижняя дорожка представляет позиции слева от начальной, но в обратном порядке. Точное упорядочение показано на рис. 8.19. Верхняя дорожка представляет клетки X_0, X_1, …, где X_0 — начальная позиция головки, а X_1, X_2, … — клетки справа от нее. Клетки X_{-1}, X_{-2}, и последующие представляют клетки слева от начальной позиции. Обратите внимание на * в левой клетке на нижней дорожке. Этот символ служит концевым маркером и предохраняет головку односторонней ленты от случайного выхода за левый конец ленты.

X_0	X_1	X_2	…
*	X_{-1}	X_{-2}	…

Рис. 8.19. Односторонняя лента может имитировать двустороннюю бесконечную ленту

Можно усилить ограничение нашей машины Тьюринга, чтобы она никогда не записывала пробелов. Это простое ограничение, вместе с лентой, бесконечной только в одну сторону, означает, что лента всегда представляет собой префикс из непустых символов, за которым следует бесконечная последовательность пробелов. Кроме того, крайний слева непустой символ всегда находится в начальной позиции ленты. В теоремах 9.19 и 10.9 будет видно, насколько полезно предполагать, что МО имеют именно такой вид.

Теорема 8.12. Каждый язык, допускаемый МТ M_2, допускается также МТ M_1 со следующими ограничениями.

1. Головка M_1 никогда не смещается влево от своей начальной позиции.
2. M_1 никогда не записывает пробелы.

Доказательство. Условие 2 обосновать легко. Введем новый ленточный символ B', выполняющий роль пробела, но отличный от него. Таким образом, если M_2 имеет правило $\delta_2(q, X) = (p, B, D)$, изменим его на $\delta_2(q, X) = (p, B', D)$. Положим $\delta_2(q, B')$ для любого состояния q равным $\delta_2(q, B)$.

Условие 1 требует бо́льших усилий. Пусть

$$M_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, q_2, B, F_2) —$$

МТ M_2, модифицированная, как описано выше, т.е. она никогда не записывает пробел B. Построим

$$M_1 = (Q_1, \Sigma \times \{B\}, \Gamma_1, \delta_1, q_0, [B, B], F_1),$$

компоненты которой определены следующим образом.

Q_1 — состояниями M_1 являются $\{q_0, q_1\} \cup (Q_2 \times \{U, L\})$, т.е. начальное q_0, еще одно q_1 и все состояния M_2 со вторым компонентом U или L (*upper* или *lower* — верхний или нижний). Второй компонент указывает на дорожку, в которой находится клетка, обозреваемая M_2 (см. рис. 8.19), т.е. U означает, что головка M_2 находится в начальной позиции или справа от нее, а L — что слева.

Γ_1 — ленточными символами являются все пары символов из Γ_2, т.е. $\Gamma_2 \times \Gamma_2$. Входными символами M_1 являются пары вида $[a, B]$, где a из Σ. Пробел M_1 имеет пробелы в обоих компонентах. Кроме того, для каждого символа X из Γ_2 в Γ_1 есть пара $[X, *]$, где * — новый символ, который отсутствует в Γ_2 и отмечает левый конец ленты M_1.

δ_1 — переходы M_1 определены следующим образом.

1. $\delta_1(q_0, [a, B]) = (q_1, [a, *], R)$ для каждого a из Σ. Первый переход машины M_1 помещает маркер * на нижнюю дорожку левой клетки. Состоянием становится q_1, а головка сдвигается вправо, поскольку не может двигаться влево или оставаться на месте.
2. $\delta_1(q_1, [X, B]) = ([q_2, U], [X, B], L)$ для каждого X из Γ_2. M_1 в состоянии q_1 устанавливает начальное состояние M_2, возвращая головку в начальную позицию и изменяя состояние на $[q_2, U]$, т.е. начальное состояние M_2 с головкой M_1 на первой дорожке.

3. Если $\delta_2(q, X) = (p, Y, D)$, то для каждого Z из Γ_2

   $\delta_1([q, U], [X, Z]) = ([p, U], [Y, Z], D)$ *и*

   $\delta_1([q, L], [Z, X]) = ([p, L], [Z, Y], \overline{D})$,

   где $\overline{D}$ — направление, противоположное D, т.е. L, если $D = R$, и R, если $D = L$. Если M_1 не находится в крайней слева клетке, то она имитирует M_2 на подходящей для этого дорожке — верхней, если вторым компонентом состояния является U, и нижней — если L. Заметим, что, работая с нижней дорожкой, M_1 сдвигает головку в направлении, противоположном сдвигу головки M_2. Такой выбор направления естественен, поскольку левая половина ленты M_2 хранится в нижней дорожке M_1 в обратном направлении.

4. Если $\delta_2(q, X) = (p, Y, R)$, то

   $\delta_1([q, L], [X, *]) = \delta_1([q, U], [X, *]) = ([p, U], [Y, *], R)$.

   Это правило распространяется на один случай обработки левого концевого маркера *. Если M_2 из начальной позиции движется вправо, то, независимо от того, была ли она раньше слева или справа от этой позиции (второй компонент состояния M_1 может быть как L, так и U), M_1 должна двигаться вправо и обрабатывать верхнюю дорожку. Таким образом, на следующем шаге M_1 окажется в позиции, представленной X_1 на рис. 8.19.

5. Если $\delta_2(q, X) = (p, Y, L)$, то

   $\delta_1([q, L], [X, *]) = \delta_1([q, U], [X, *]) = ([p, L], [Y, *], R)$.

   Это правило подобно предыдущему, но связано со случаем, когда M_2 сдвигается влево от своей начальной позиции. M_1 должна двигаться вправо от своего концевого маркера, но обрабатывать нижнюю дорожку, т.е. клетку, представленную X_{-1} на рис. 8.19.

F_1 — множеством допускающих состояний является $F_2 \times \{U, L\}$, т.е. все состояния M_1, первым компонентом которых является допускающее состояние M_2. В момент допускания M_1 может обрабатывать как верхнюю, так и нижнюю дорожку.

Доказательство теоремы по существу завершено. С помощью индукции по числу переходов, совершаемых M_2, можно заметить, что M_1 имитирует МО M_2 на своей ленте, если взять нижнюю дорожку, развернуть и дописать к ней верхнюю.[8] Заметим также, что M_1 попадает в одно из допускающих состояний тогда и только тогда, когда это же делает M_2. Таким образом, $L(M_1) = L(M_2)$. □

[8] Не забудем и о *, которую нужно удалить. — *Прим. перев.*

8.5.2. Мультистековые машины

Теперь рассмотрим несколько вычислительных моделей, основанных на обобщениях магазинных автоматов. Вначале рассмотрим, что происходит, если МП-автомат имеет несколько магазинов. Из примера 8.7 уже известно, что МТ может допускать языки, не допускаемые никаким МП-автоматом с одним магазином. Однако оказывается, что если снабдить МП-автомат двумя магазинами, то он может допустить любой язык, допускаемый МТ.

Рассмотрим также класс машин, называемых “счетчиковыми машинами”. Они обладают возможностью запоминать конечное число целых чисел (“счетчиков”) и совершать различные переходы в зависимости от того, какие из счетчиков равны 0 (если таковые вообще есть). Счетчиковая машина может только прибавить 1 к счетчику или вычесть 1 из него, но отличить значения двух различных ненулевых счетчиков она не способна. Следовательно, счетчик похож на магазин с двухсимвольным алфавитом, состоящим из маркера дна (он появляется только на дне) и еще одного символа, который можно заносить в магазин и выталкивать из него.

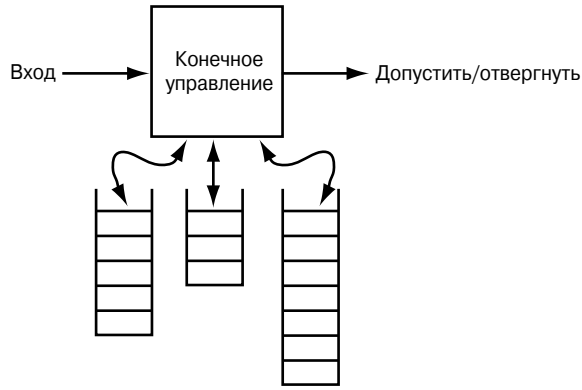


Рис. 8.20. Машина с тремя магазинами

Формальное описание работы *мультистековой*, или *многомагазинной* машины здесь не приводится, но идея иллюстрируется рис. 8.20; k-магазинная машина представляет собой детерминированный МП-автомат с k магазинами. Он получает свои входные данные, как и МП-автомат, из некоторого их источника, а не с ленты или из магазина, как МТ. Мультистековая машина имеет конечное управление, т.е. конечное множество состояний, и конечный магазинный алфавит, используемый для всех магазинов. Переход мультистековой машины основывается на состоянии, входном символе и верхних символах всех магазинов.

Мультистековая машина может совершить ε-переход, не используя входной символ, но для того, чтобы машина была детерминированной, в любой ситуации не должно быть возможности выбора ε-перехода или перехода по символу.

За один переход мультистековая машина может изменить состояние и заменить верхний символ в каждом из магазинов цепочкой из нуля или нескольких магазинных символов. Обычно для каждого магазина бывает своя замещающая цепочка.

Итак, типичное правило перехода для k-магазинной машины имеет вид

$$\delta(q, a, X_1, X_2, \ldots, X_k) = (p, \gamma_1, \gamma_2, \ldots, \gamma_k).$$

Его смысл состоит в том, что, находясь в состоянии q, с X_i на вершине i-го магазина, $i = 1, 2, \ldots, k$, машина может прочитать на входе a (либо символ алфавита, либо ε), перейти в состояние p и заменить X_i на вершине i-го магазина цепочкой γ_i, $i = 1, 2, \ldots, k$. Мультистековая машина допускает, попав в заключительное состояние.

Добавим одно свойство, которое упрощает обработку входа описанной детерминированной машиной. Будем предполагать, что есть специальный символ \$, называемый *концевым маркером* (*маркером конца входа*), который встречается только в конце входа и не является его частью. Присутствие маркера позволяет узнать, когда прочитан весь доступный вход. В доказательстве следующей теоремы будет показано, каким образом концевой маркер облегчает имитацию машины Тьюринга на мультистековой машине. Заметим, что обычная МТ не нуждается в специальном маркере конца, поскольку в этой роли выступает первый пробел.

Теорема 8.13. Если язык L допускается машиной Тьюринга, то L допускается двухмагазинной машиной.

Доказательство. Основная идея состоит в том, что два магазина могут имитировать одну ленту машины Тьюринга, причем один магазин хранит содержимое ленты слева от головки, а второй — то, что находится справа, за исключением бесконечных цепочек пробелов, окружающих значащие символы ленты. Более детально, пусть L есть $L(M)$ для некоторой (одноленточной) МТ M. Наша двухмагазинная машина S выполняет следующие действия.

1. S начинает с *маркером дна* в каждом из магазинов. Этот маркер может быть стартовым символом для магазинов и не должен появляться больше нигде в магазине. В дальнейшем будем говорить, что “магазин пуст”, когда он содержит лишь маркер дна.
2. Пусть w\$ является входом S. S копирует w в первый магазин, прекращая копирование, прочитав маркер конца входа.
3. S выталкивает каждый символ по очереди из первого магазина и помещает его во второй магазин. Теперь первый магазин пуст, а второй содержит w с ее левым концом на вершине.
4. S переходит в начальное (имитируемое) состояние M. Первый магазин S пуст, свидетельствуя о том, что у M слева от головки нет ничего, кроме пробелов. Второй магазин S содержит w, отражая тот факт, что справа от головки M находится w.
5. S имитирует переход M следующим образом:

а) S знает состояние M, скажем, q, поскольку S имитирует состояние M в своем собственном конечном управлении;

б) S знает символ X, обозреваемый головкой M; это символ на вершине второго магазина S. Как исключение, если второй магазин содержит только маркер дна, то M только что переместилась на пробел; S интерпретирует символ, обозреваемый M, как пробел;

в) итак, S знает следующий переход M;

г) следующее состояние M записывается в компоненте конечного управления S вместо предыдущего состояния;

д) если M меняет X на Y и сдвигается вправо, то S помещает Y в свой первый магазин, имитируя то, что Y теперь слева от головки M. X выталкивается из второго магазина S. Однако есть два исключения.

i. Если второй магазин содержит только маркер дна (следовательно, X — пробел), то второй магазин не изменяется; M сдвинулась еще на один пробел вправо.

ii. Если Y — пробел, и первый магазин пуст, то этот магазин остается пустым. Причина в том, что слева от головки M находятся только пробелы;

е) если M меняет X на Y и сдвигается влево, то S выталкивает символ с вершины первого магазина, скажем, Z, затем меняет X на ZY во втором магазине. Это изменение отражает, что символ Z, ранее расположенный слева от головки M, теперь обозревается ею. Как исключение, если Z является маркером дна, то S должна поместить BY во второй магазин, не выталкивая ничего из первого.

6. S допускает, если новое состояние M является допускающим. В противном случае S имитирует еще один переход M таким же способом.

8.5.3. Счетчиковые машины

Счетчиковую машину можно представить одним из двух способов.

1. Счетчиковая машина имеет такую же структуру, как и мультистековая (см. рис. 8.20), но вместо магазинов у нее счетчики. Счетчики содержат произвольные неотрицательные целые числа, но отличить можно только ненулевое от нулевого. Таким образом, переход счетчиковой машины зависит от ее состояния, входного символа и того, какие из счетчиков являются нулевыми. За один переход машина может изменить состояние и добавить или отнять 1 от любого из счетчиков. Однако счетчик не может быть отрицательным, поэтому отнимать 1 от счетчика со значением 0 нельзя.
2. Счетчиковая машина также может рассматриваться, как мультистековая машина со следующими ограничениями:

а) есть только два магазинных символа, Z_0 (*маркер дна*) и X;

б) вначале Z_0 находится в каждом магазине;

в) Z_0 можно заменить только цепочкой вида X^iZ_0 для некоторого $i \geq 0$;

г) X можно заменить только цепочкой вида X^i для некоторого $i \geq 0$. Таким образом, Z_0 встречается только на дне каждого магазина, а все остальные символы (если есть) — это символы X.

Для счетчиковых машин будем использовать определение 1, хотя оба они, очевидно, задают машины одинаковой мощности. Причина в том, что магазин X^iZ_0 может быть идентифицирован значением i. В определении 2 значение 0 можно отличить от остальных, поскольку значению 0 соответствует Z_0 на вершине магазина, в противном случае там помещается X. Однако отличить два положительных числа невозможно, поскольку обоим соответствует X на вершине магазина.

8.5.4. Мощность счетчиковых машин

О языках счетчиковых машин стоит сделать несколько очевидных замечаний.

- Каждый язык, допускаемый счетчиковой машиной, рекурсивно перечислим. Причина в том, что счетчиковые машины являются частным случаем магазинных, а магазинные — частным случаем многоленточных машин Тьюринга, которые по теореме 8.9 допускают только рекурсивно перечислимые языки.
- Каждый язык, допускаемый односчетчиковой машиной, является КС-языком. Заметим, что счетчик, с точки зрения определения 2, является магазином, поэтому односчетчиковая машина представляет собой частный случай одномагазинной, т.е. МП-автомата. Языки односчетчиковых машин допускаются детерминированными МП-автоматами, хотя доказать это на удивление сложно. Трудность вызывает тот факт, что мультистековые и счетчиковые машины имеют маркер $ в конце входа. Недетерминированный МП-автомат может “догадаться”, что он видит последний входной символ, и следующим будет маркер. Таким образом, ясно, что недетерминированный МП-автомат без концевого маркера может имитировать ДМП-автомат с маркером. Однако доказать, что ДМП-автомат без концевого маркера может имитировать ДМП-автомат *с маркером*, весьма трудно, и мы этого не делаем.

Удивительно, но для имитации машины Тьюринга и, следовательно, для допускания любого рекурсивно перечислимого языка, достаточно двух счетчиков. Для обоснования этого утверждения вначале доказывается, что достаточно трех счетчиков, а затем три счетчика имитируются с помощью двух.

Теорема 8.14. Каждый рекурсивно перечислимый язык допускается трехсчетчиковой машиной.

Доказательство. Начнем с теоремы 8.13, утверждавшей, что каждый рекурсивно перечислимый язык допускается двухмагазинной машиной. С ее использованием нам достаточно показать, как магазин имитируется с помощью счетчиков. Предположим, магазинная машина использует $r - 1$ ленточных символов. Можно обозначить символы цифрами от 1 до $r - 1$ и рассматривать магазин $X_1X_2\ldots X_n$ как целое число по основанию r, т.е. этот магазин (его вершина, как обычно, слева) представлен целым числом $X_n r^{n-1} + X_{n-1} r^{n-2} + \ldots + X_2 r + X_1$.

Используем два счетчика для хранения целых чисел, представляющих каждый из двух магазинов. Третий счетчик используется для установки двух других. В частности, третий счетчик нужен при делении или умножении числа на r.

Операции над магазином можно разделить на три вида: вытолкнуть верхний символ из магазина, изменить магазинный символ и поместить символ в магазин. Переход двухмагазинной машины может вовлекать несколько таких операций; в частности, замену верхнего символа X цепочкой символов нужно разделить на замену X и затем помещение дополнительных символов в магазин. Эти операции выполняются над магазином, который представлен числом i, следующим образом. Заметим, что для выполнения операций, требующих подсчета чисел от 0 до $r - 1$, можно использовать конечное управление мультистековой машины.

1. Для выталкивания из магазина нужно изменить i на i/r, отбрасывая остаток, которым является X_1. Начиная с нулевого значения третьего счетчика, число i несколько раз сокращается на r, а третий счетчик увеличивается на 1. Когда счетчик, первоначально имевший значение i, достигает 0, происходит остановка. Затем исходный счетчик увеличивается на 1 и третий счетчик уменьшается на 1 до тех пор, пока третий счетчик снова не станет нулевым. В этот момент счетчик, в котором вначале было i, содержит i/r.
2. Для изменения X на Y на вершине магазина, представленного целым i, число i увеличивается или уменьшается на небольшую величину, заведомо не больше r. Если Y и X рассматриваются как цифры, и $Y > X$, то i увеличивается на $Y - X$; если $Y < X$, то i уменьшается на $X - Y$.
3. Для помещения X в магазин, содержащий i, нужно поменять i на $ir + X$. Вначале умножаем i на r. Для этого несколько раз уменьшаем значение i на 1 и увеличиваем третий счетчик (который, как всегда, начинается с 0) на r. Когда исходный счетчик достигнет 0, в третьем счетчике будет ir. Третий счетчик копируется в исходный и вновь обнуляется, как в п. 1. Наконец, исходный счетчик увеличивается на X.

Для завершения конструкции нужно инициализировать счетчики, чтобы имитировать магазины из их начального состояния, в котором они содержат только начальный символ двухмагазинной машины. Этот шаг реализуется путем увеличения двух основных счетчиков на некоторое число от 1 до $r - 1$, соответствующее начальному символу. □

Теорема 8.15. Каждый рекурсивно перечислимый язык допускается двухсчетчиковой машиной.

Доказательство. С учетом предыдущей теоремы нужно лишь показать, как имитировать три счетчика с помощью двух. Идея состоит в том, чтобы представить три счетчика, скажем, i, j и k, одним целым числом. Этим числом будет $m = 2^i3^j5^k$. Один счетчик будет хранить это число, а второй использоваться в качестве вспомогательного для умножения и деления m на одно из трех простых чисел 2, 3 или 5. Для имитации трехсчетчиковой машины нужно реализовать следующие операции.

1. Увеличить i, j и/или k. Для того чтобы увеличить i на 1, нужно умножить m на 2. В теореме 8.14 уже показано, как умножить содержимое счетчика на константу r, используя второй счетчик. Аналогично j увеличивается путем умножения m на 3, а k — на 5.
2. Различить, какие из чисел, i, j или k, равны 0. Для того чтобы выяснить, что $i = 0$, нужно определить, делится ли m без остатка на 2. Число m копируется во второй счетчик с использованием состояния машины, чтобы запомнить, уменьшено m четное или нечетное число раз. Если m уменьшено нечетное число раз к тому моменту, когда оно стало равно 0, то $i = 0$. В таком случае m копируется из второго счетчика в первый. Аналогично, равенство $j = 0$ проверяется путем определения, делится ли m на 3, а $k = 0$ — делится ли оно на 5.
3. Уменьшить i, j и/или k. Для этого m делится на 2, 3 или 5, соответственно. В доказательстве теоремы 8.14 описано, как выполнить деление на любую константу с использованием дополнительного счетчика. Трехсчетчиковая машина не может уменьшить число 0 (это ошибка, ведущая к останову без допускания), поэтому если m не делится нацело на соответствующую константу, то имитирующая двухсчетчиковая машина также останавливается без допускания.

□

8.5.5. Упражнения к разделу 8.5

8.5.1. Неформально, но четко и ясно опишите счетчиковые машины, которые допускают следующие языки. В каждом случае используйте как можно меньше счетчиков, но не более двух:

а) (*) $\{0^n1^m \mid n \geq m \geq 1\}$;

б) $\{0^n1^m \mid 1 \leq n \leq m\}$;

в) (*!) $\{a^ib^jc^k \mid i = j$ или $i = k\}$;

г) (!!) $\{a^ib^jc^k \mid i = j$ или $i = k$ или $j = k\}$.

8.5.2. (!!) Цель этого упражнения — показать, что мощность односчетчиковых машин с маркером конца входа не превосходит мощности детерминированных МП-автоматов. $L\$$ — это конкатенация языка L с языком, содержащим одну-

единственную цепочку \$, т.е. L\$ состоит из всех строк w\$, где w принадлежит L. Покажите, что если язык L\$ допускается ДМП-автоматом, где \$ — концевой маркер, не встречающийся в цепочках из L, то L также допускается некоторым ДМП-автоматом. *Указание*. Этот вопрос совпадает с вопросом замкнутости языков, допускаемых ДМП-автоматами, относительно операции L/a, определенной в упражнении 4.2.2. Нужно модифицировать ДМП-автомат P для L\$ путем замены каждого из его магазинных символов X всеми возможными парами (X, S), где S есть множество состояний. Если P имеет магазин $X_1X_2\ldots X_n$, то построенный для L ДМП-автомат имеет магазин $(X_1, S_1)(X_2, S_2)\ldots(X_n, S_n)$, где каждое S_i есть множество состояний q, в которых P, начав с МО $(q, a, X_iX_{i+1}\ldots X_n)$, допускает.

8.6. Машины Тьюринга и компьютеры

Теперь сравним машины Тьюринга с обычным компьютером. Эти модели кажутся совершенно разными, но они допускают одни и те же языки — рекурсивно перечислимые. Поскольку понятие “обычный компьютер” математически не определено, аргументация данного раздела не является формальной. Нам приходится обращаться к интуиции читателя по поводу того, что могут компьютеры, в частности, если рассматриваемые числа превосходят обычные пределы, обусловленные архитектурой компьютеров (например, 32-битовым представлением чисел). Утверждения данного раздела можно разделить на две части.

1. Компьютер может имитировать машину Тьюринга.
2. Машина Тьюринга может имитировать компьютер, причем за время, которое можно выразить в виде некоторого полинома от числа шагов, совершаемых компьютером.

8.6.1. Имитация машины Тьюринга на компьютере

Вначале посмотрим, как компьютер имитирует машину Тьюринга. Имея некоторую конкретную машину Тьюринга M, мы должны написать программу, которая работает, как M. Одним из составляющих M является ее конечное управление. Поскольку M имеет только конечное число состояний и конечное число правил перехода, наша программа может закодировать состояния в виде цепочек символов и использовать таблицу переходов, которую она просматривает для определения каждого перехода. Аналогично, ленточные символы можно закодировать в виде цепочек символов фиксированной длины, поскольку ленточных символов также конечное число.

Серьезный вопрос возникает, когда мы решаем, как программа должна имитировать ленту машины Тьюринга. Эта лента может стать бесконечно длинной, но память компь-

ютера — главная память, диск или другие устройства — конечна. Можно ли проимитировать бесконечную ленту с помощью фиксированного объема памяти?

Если у нас нет возможности заменять запоминающие устройства, то проимитировать действительно нельзя; в таком случае компьютер был бы конечным автоматом, и допускал бы только регулярные языки. Однако обычные компьютеры имеют заменяемые запоминающие устройства, например, “Zip”-диски. В действительности обычный жесткий диск можно снять и заменить пустым, но идентичным диском.

Поскольку очевидного предела на количество используемых дисков нет, будем считать, что доступны столько дисков, сколько может потребоваться компьютеру. Тогда эти диски можно разместить в двух магазинах (рис. 8.21). Один магазин содержит данные в удаленных клетках ленты машины Тьюринга, расположенных слева от ее головки, а другой — в дальних клетках справа от головки. Чем дальше в магазине расположены данные, тем дальше они от головки на ленте.

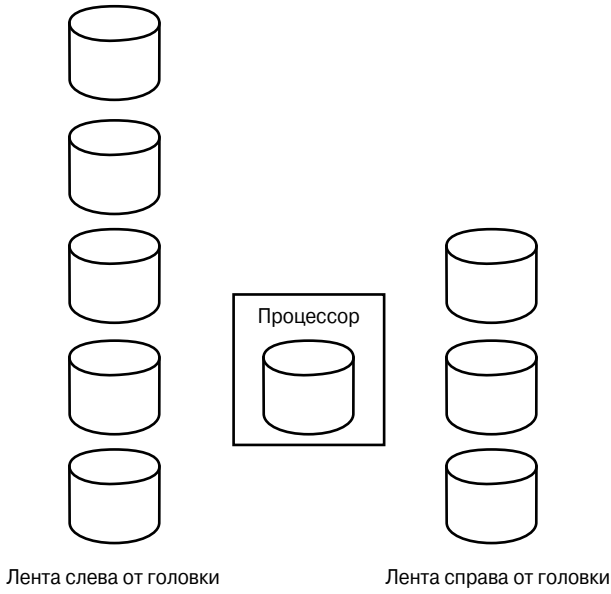


Рис. 8.21. Имитация машины Тьюринга на обычном компьютере

Если ленточная головка МТ сдвигается далеко влево и достигает клеток, не представленных в текущем смонтированном диске, то программа печатает сообщение “обмен слева”. Смонтированный диск снимается человеком-оператором и помещается на вершину правого магазина. На компьютер монтируется диск с вершины левого магазина, и вычисления продолжаются.

Аналогично, если ленточная головка МТ достигает удаленных клеток, не представленных на текущем смонтированном диске, то печатается сообщение “обмен справа”. Человек перемещает текущий диск на вершину левого магазина и монтирует на компьютер диск с

вершины правого магазина. Ситуация, когда какой-либо магазин пуст, а компьютер требует, чтобы диск из него был смонтирован, означает, что МТ достигла области пробелов на ленте. В этом случае оператору придется купить новый диск и смонтировать его.

8.6.2. Имитация компьютера на машине Тьюринга

Нужно также провести обратное сравнение: существуют ли вещи, которые может сделать обычный компьютер, а машина Тьюринга — нет. Важным вторичным вопросом является вопрос, может ли компьютер делать определенные вещи существенно быстрее, чем машина Тьюринга. В этом разделе обосновывается, что МТ может имитировать компьютер, а в разделе 8.6.3 — что эта имитация может быть достаточно быстрой в том смысле, что "всего лишь" полиномиальная зависимость разделяет время работы компьютера и МТ для решения какой-либо проблемы. Напомним читателю еще раз, что существуют немаловажные причины считать подобными все времена работы, лежащие в пределах полиномиальной зависимости друг от друга, тогда как экспоненциальные отличия во времени работы "слишком велики". Теория, в которой проводится сравнение полиномиального и экспоненциального времени работы, будет рассмотрена в главе 10.

Проблема очень больших ленточных алфавитов

Доводы раздела 8.6.1 становятся сомнительными, если число ленточных символов настолько велико, что код одного ленточного символа не умещается на диске. Для этого нужно действительно очень много ленточных символов, поскольку 30-гигабайтный диск, например, может представить любой из $2^{240000000000}$ символов. Аналогично, число состояний могло бы быть настолько большим, что состояние нельзя было представить, используя целый диск.

Для одного из разрешений этой проблемы следует ограничить число ленточных символов, используемых МТ. Любой ленточный алфавит можно закодировать в двоичном виде. Таким образом, любая МТ M может быть проимитирована с помощью другой МТ M', использующей только ленточные символы 0, 1 и B. Однако у M' должно быть очень много состояний, поскольку для имитации перехода M МТ M' должна сканировать ее ленточный символ и запомнить в своем конечном управлении все биты, показывающие, какой ленточный символ сканируется. Таким же образом мы сталкиваемся с огромными множествами состояний, и компьютер, который имитирует M', должен монтировать и демонтировать несколько дисков, решая, каким же является и каким будет следующее состояние M'. Естественно, о компьютерах, решающих задачи подобного рода, никто и не задумывается, поэтому обычные операционные системы не имеют поддержки для таких задач. Однако при желании мы могли бы запрограммировать "голый", лишенный операционной системы компьютер, снабдив его соответствующими возможностями.

К счастью, вопрос о том, как имитировать МТ с огромным числом состояний или ленточных символов, разрешим. В разделе 9.2.3 будет показано, что можно построить МТ, которая по существу есть МТ с "запоминаемой программой". Эта МТ, называемая "универсальной", считывает функцию переходов любой МТ, закодированную в двоичном виде на ленте, и имитирует ее. Универсальная МТ имеет вполне разумное число состояний и ленточных символов. С помощью имитации универсальной МТ обычный компьютер может быть запрограммирован для допускания какого угодно рекурсивно перечислимого языка без необходимости имитировать количества состояний, которые нарушают пределы того, что может быть записано на диске.

Приведем реалистичную, хотя и неформальную, модель функционирования обычного компьютера.

1. Предполагается, что память компьютера состоит из неопределенно длинной последовательности *слов*, каждое из которых имеет *адрес*. В реальном компьютере слова были бы длиной 32 или 64 бит, но мы длину слова ограничивать не будем. В качестве адресов используются целые 0, 1, 2 и т.д. В реальном компьютере последовательными целыми нумеровались бы отдельные байты, поэтому адреса слов были бы кратны 4 или 8, но это различие несущественно. В реальном компьютере количество слов "памяти" также было бы ограничено, но мы хотим учесть содержимое произвольного числа дисков или других запоминающих устройств, поэтому предполагаем, что количество слов ничем не ограничено.
2. Предполагается, что программа компьютера записывается в некоторые слова памяти. Каждое из этих слов представляет простую инструкцию, как в машинном или ассемблерном языке обычного компьютера. Примерами служат инструкции перемещения данных из одного слова в другое или прибавления одного слова к другому. Допускается "непрямая адресация", т.е. инструкция может ссылаться на другое слово и использовать его содержимое в качестве адреса слова, к которому применяется операция. Эта возможность, обычная для современных компьютеров, нужна для доступа к массивам, для следования по связям списков и вообще для выполнения операций с указателями.
3. Предполагается, что каждая инструкция использует ограниченное (конечное) число слов и изменяет значение не более одного слова.
4. Обычный компьютер имеет *регистры* — слова памяти с особо быстрым доступом. Часто такие операции, как сложение, применяются только к регистрам, но мы подобные ограничения не вводим и считаем, что с любым словом можно произвести любую операцию. Относительная скорость операций на различных словах значения не имеет, поскольку вообще не нужна при сравнении возможностей компьютеров и машин Тьюринга по распознаванию языков. Даже если нас интересует полиноми-

альная связь между временами работы, разница в скорости доступа к различным словам остается неважной, поскольку влияет “лишь” на константный сомножитель.

Возможная конструкция машины Тьюринга для имитации компьютера представлена на рис. 8.22. Эта МТ имеет несколько лент, но с помощью построений из раздела 8.4.1 ее можно преобразовать в одноленточную. Первая лента представляет всю память компьютера. Мы использовали код, в котором адреса слов памяти в порядке их номеров перемежаются со значениями (содержимым) этих слов. И адреса, и значения записаны в двоичном виде. Маркеры * и # используются для того, чтобы легко было найти конец адреса и значения слова и различить, является ли двоичная цепочка адресом или значением. Еще один маркер, $, отмечает начало последовательности адресов и значений.

Вторая лента представляет собой “счетчик инструкций”. Эта лента содержит одно двоичное целое, представляющее одну из позиций в памяти на первой ленте. Оно интерпретируется как адрес инструкции, которая должна быть выполнена следующей.

Третья лента содержит “адрес памяти” или значение по нему после того, как этот адрес устанавливается на первой ленте. Для выполнения инструкции МТ должна найти значение по одному или нескольким адресам памяти, где хранятся данные, участвующие в вычислении. Нужный адрес копируется на ленту 3 и сравнивается с адресами на ленте 1 до совпадения. Значение по этому адресу копируется на третью ленту и перемещается в нужное место, как правило, по одному из начальных адресов, представляющих регистры компьютера.

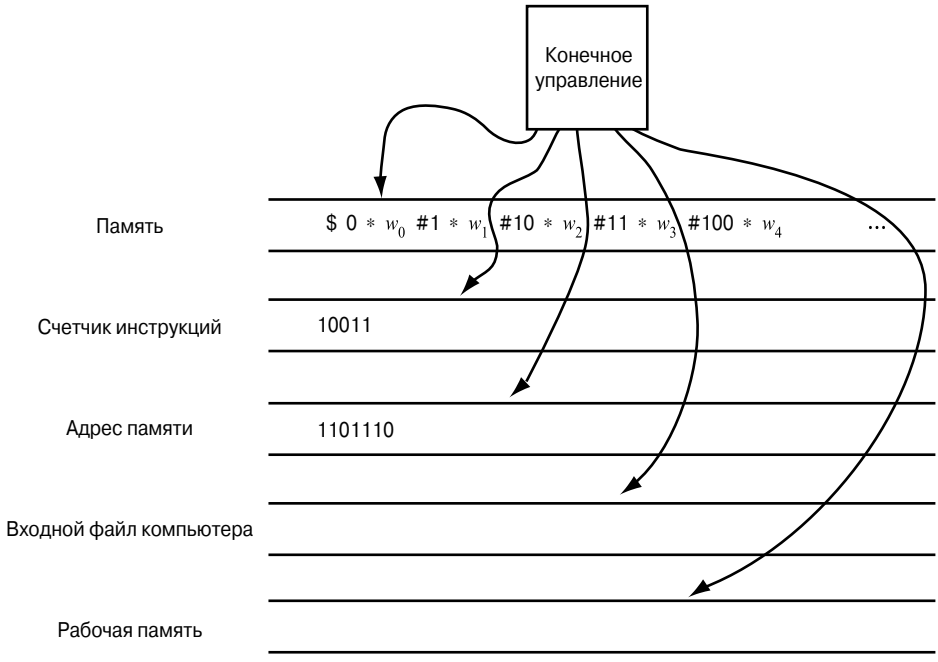


Рис. 8.22. Машина Тьюринга, имитирующая обычный компьютер

Наша МТ имитирует *цикл инструкции* (instruction cycle) компьютера следующим образом.

1. На первой ленте ищется адрес, совпадающий с номером инструкции на ленте 2. Начиная с маркера $, движемся вправо, сравнивая каждый адрес с содержимым ленты 2. Сравнить адреса на двух лентах легко, поскольку нужно лишь синхронно сдвигать ленточные головки вправо, проверяя совпадение обозреваемых символов.
2. Обнаружив адрес инструкции, исследуем значение по нему. Предположим, что слово представляет инструкцию, его первые биты задают действие (например, копировать, прибавить, ветвиться), а оставшиеся биты кодируют адрес или адреса, используемые в этом действии.
3. Если в инструкции используется значение по некоторому адресу, то этот адрес является частью инструкции. Он копируется на третью ленту, а позиция инструкции отмечается с помощью второй дорожки первой ленты (не показанной на рис. 8.22), поэтому при необходимости к инструкции легко вернуться. Ищем адрес на первой ленте, и значение по нему копируем на ленту 3, хранящую адрес памяти.
4. Выполняем инструкцию или ее часть, использующую данное значение. С новым значением можно сделать следующее:
    а) скопировать значение по другому адресу. Второй адрес извлекается из инструкции, помещается на ленту 3 и находится на ленте 1, как описано выше. Когда второй адрес найден, значение по нему копируется в пространство, зарезервированное для него. Если для нового значения нужно больше или меньше памяти, чем для старого, доступное пространство изменяется путем *сдвига* (shifting over). Он осуществляется так.
        i. На рабочую ленту копируется вся значащая (без пробелов) часть ленты 1 справа от того места, куда нужно поместить новое значение.
        ii. Новое значение записывается в пространство нужного объема.
        iii. Рабочая лента копируется обратно, на ленту 1, непосредственно справа от нового значения.

    В особых случаях адрес может не встретиться на первой ленте, поскольку не использовался ранее. Тогда на первой ленте находится место, к которому относится данный адрес, при необходимости делается сдвиг, и в это место записывается адрес и новое значение;

    б) прибавить найденное значение к значению по другому адресу. Для получения второго адреса возвращаемся к инструкции. Ищем адрес на ленте 1. Выполняем двоичное сложение значения по этому адресу и значения, записанного на ленте 3. Сканируя два значения справа налево, МТ может выполнить

сложение с переносом. Если для результата нужно больше места, на ленте 1 используется техника сдвига;

в) перейти к выполнению инструкции по адресу, заданному значением на ленте 3. Для этого лента 3 просто копируется на ленту 2, и цикл инструкции начинается вновь.

5. Выполнив инструкцию и выяснив, что она не является переходом, к счетчику инструкций на ленте 2 прибавляем 1 и вновь начинаем цикл инструкции.

В имитации компьютера с помощью МТ есть и другие подробности. На рис. 8.22 изображена четвертая лента, содержащая имитируемый вход компьютера, поскольку компьютер должен читать из файла свой вход — слово, проверяемое на принадлежность языку. МТ вместо файла может использовать ленту.

Показана также рабочая лента. Имитация некоторых инструкций компьютера может эффективно использовать рабочую ленту или ленты для вычисления таких арифметических операций, как умножение.

Наконец, можно предполагать, что компьютер порождает выход, говорящий о допустимости входных данных. Для перевода этого действия в термины МТ предположим, что у компьютера есть "допускающая" инструкция, которая, возможно, соответствует функции, вызываемой компьютером для печати `yes` в выходном файле. Когда МТ имитирует выполнение этой инструкции компьютера, она переходит в собственное допускающее состояние и останавливается.

Хотя представленная выше аргументация далека от полного формального доказательства того, что МТ может имитировать обычный компьютер, она достаточно подробна, чтобы убедить, что МТ является подходящим представлением действий компьютера. Итак, в качестве формального представления того, что может быть вычислено на любом типе вычислительных устройств, в дальнейшем будет использоваться только машина Тьюринга.

8.6.3. Сравнение времени работы компьютеров и машин Тьюринга

Теперь обратимся к вопросу о времени работы на машине Тьюринга, имитирующей компьютер. Как и раньше, мы исходим из следующих утверждений.

- Вопрос о времени работы важен, поскольку МТ используется для исследования вопросов не только о том, что вообще можно вычислить, но и о том, что можно вычислить с эффективностью, достаточной для практического использования компьютерного решения проблемы.
- Проблемы делятся на *легко разрешимые* (их можно решить эффективно) и *трудно разрешимые* (они могут быть решены, но настолько медленно, что их решением нельзя воспользоваться) на основе того, что можно решить за полиномиальное время, и что требует времени большего, чем любое полиномиальное.

- Таким образом, нужно убедиться, что если проблему можно решить за полиномиальное время на обычном компьютере, то ее можно решить за полиномиальное время на машине Тьюринга, и наоборот. Вследствие этой полиномиальной эквивалентности наши выводы о том, что могут и чего не могут машины Тьюринга, адекватно применимы и к компьютерам.

Напомним: в разделе 8.4.3 было определено, что разница между временем работы на одноленточной и многоленточной МТ является полиномиальной, точнее, квадратичной. Таким образом, достаточно показать, что все, что может сделать компьютер, может и многоленточная МТ, описанная в разделе 8.6.2, причем за время, полиномиальное относительно времени работы компьютера. Тогда то же самое будет справедливо и для одноленточной МТ.

Перед тем как доказать, что машина Тьюринга, описанная выше, может имитировать n шагов компьютера за время $O(n^3)$, нам нужно исследовать вопрос умножения как машинной инструкции. Проблема в том, что предельное число битов в одном машинном слове не было установлено. Если, скажем, компьютер начал бы со слова, содержащего 2, и должен был бы n раз последовательно умножить это слово само на себя, то оно имело бы значение 2^{2^n}. Это число требует $2^n + 1$ битов для представления, поэтому время, необходимое машине Тьюринга для имитации этих n инструкций, будет, как минимум, экспоненциальным.

Один из подходов разрешения этой проблемы заключается в том, чтобы настаивать, что слова имеют фиксированную максимальную длину, скажем, 64 бит. Тогда умножения (или другие операции), порождающие слишком длинные слова, будут приводить компьютер к остановке, и машине Тьюринга не нужно будет имитировать его дальше. Мы сделаем более вольное предположение — компьютер использует слова, длина которых может неограниченно возрастать, но одной компьютерной инструкции позволено породить слово, которое на один бит длиннее, чем ее операнды.

Пример 8.16. При действии последнего ограничения сложение является допустимой операцией, поскольку длина результата может быть лишь на один бит больше, чем максимальная длина слагаемых. Умножение недопустимо, поскольку два m-битовых слова могут дать произведение длиной $2m$. Однако умножение m-битовых целых можно имитировать с помощью m последовательных сложений, перемежаемых сдвигами сомножителей на один бит влево (что является еще одной операцией, увеличивающей длину слова на 1). Таким образом, мы все еще можем умножать неограниченно длинные слова, но время, нужное компьютеру, пропорционально квадрату длины операндов. □

Предположив, что при выполнении компьютерной инструкции длина возрастает не более, чем на один бит, можно доказать полиномиальную взаимосвязь между двумя временами работы. Идея доказательства — заметить, что после выполнения n инструкций количество слов, упоминаемых на ленте памяти МТ, есть $O(n)$, и каждое компьютерное слово требует $O(n)$ клеток МТ для его представления. Таким образом, лента содержит

$O(n^2)$ клеток, и МТ может найти конечное число слов, необходимое для выполнения одной инструкции компьютера, за время $O(n^2)$.

Существует, однако, одно дополнительное требование к инструкциям. Даже если инструкция не порождает длинное слово в качестве результата, она может занимать очень много времени для его вычисления. Поэтому делается дополнительное предположение, что сама по себе инструкция, применяемая к словам длиной не более k, может быть выполнена за $O(k^2)$ шагов на многоленточной машине Тьюринга. Несомненно, что обычные операции компьютера, такие как сложение, сдвиг или сравнение значений, могут быть выполнены за $O(k)$ шагов многоленточной МТ, поэтому мы даже слишком либеральны, допуская, что может делать компьютер в одной инструкции.

Теорема 8.17. Пусть компьютер обладает следующими свойствами.

1. У него есть только инструкции, увеличивающие максимальную длину слова не более, чем на 1.
2. У него есть только инструкции, которые многоленточная МТ может выполнить на словах длиной k за $O(k^2)$ или меньшее число шагов.

Тогда машина Тьюринга, описанная в разделе 8.6.2, может имитировать n шагов компьютера за $O(n^3)$ своих шагов.

Доказательство. Начнем с замечания, что первая лента (лента памяти) машины Тьюринга (см. рис. 8.22) вначале содержит только программу компьютера. Эта программа может быть длинной, но длина ее фиксирована и является константой, не зависящей от n — числа шагов выполнения ее инструкций. Таким образом, существует некоторая константа c, представляющая собой наибольшее из компьютерных слов или адресов, встречающихся в программе. Существует также константа d — количество слов, занимаемых программой.

Итак, после выполнения n шагов компьютер не может породить слово длиннее, чем $c + n$, и не может создать или использовать адрес, занимающий больше $c + n$ битов. Каждая инструкция порождает не более одного нового адреса, получающего значение, поэтому общее число адресов после выполнения n инструкций не превышает $d + n$. Поскольку любая комбинация “адрес-слово” требует не более $2(c + n) + 2$ разрядов, включая адрес, содержимое и два маркера для их разделения, общее число клеток, занятых после имитации n инструкций, не больше $2(d + n)(c + n + 1)$. Поскольку c и d — константы, это число клеток есть $O(n^2)$.

Нам известно, что каждый из фиксированного числа просмотров адресов, используемых в одной инструкции компьютера, может быть выполнен за время $O(n^2)$. Поскольку слова имеют длину $O(n)$, допустим, что сами по себе инструкции могут быть выполнены на МТ за время $O(n^2)$. Остается еще цена инструкции, которая составлена временем, необходимым машине Тьюринга для создания нового пространства, чтобы сохранить новое или расширенное слово. Однако сдвиг включает копирование данных объемом не более $O(n^2)$ с ленты 1 на рабочую ленту и обратно. Таким образом, сдвиг также требует лишь $O(n^2)$ времени на одну инструкцию компьютера.

Заключаем, что МТ имитирует один шаг компьютера за $O(n^2)$ своих шагов. Таким образом, как утверждается в теореме, n шагов компьютера можно проимитировать за $O(n^3)$ шагов машины Тьюринга. □

В заключение отметим, что возведение в куб числа шагов позволяет многоленточной МТ имитировать компьютер. Из материала раздела 8.4.3 известно, что одноленточная МТ может имитировать многоленточную не более, чем за квадрат числа шагов. Таким образом, имеет место следующее утверждение.

Теорема 8.18. Выполнение n шагов работы компьютера типа, описанного в теореме 8.17, можно проимитировать на одноленточной машине Тьюринга с использованием не более $O(n^6)$ шагов. □

Резюме

- *Машина Тьюринга*. МТ представляет собой абстрактную вычислительную машину, мощность которой совпадает с мощностью как реальных компьютеров, так и других математических определений того, что может быть вычислено. МТ состоит из управления с конечным числом состояний и бесконечной ленты, разделенной на клетки. Каждая клетка хранит один из конечного числа ленточных символов, и одна из клеток является текущей позицией ленточной головки. МТ совершает переходы на основе своего текущего состояния и ленточного символа в клетке, обозреваемой головкой. За один переход она изменяет состояние, переписывает обозреваемый символ и сдвигает головку на одну клетку вправо или влево.
- *Допускание машиной Тьюринга*. МТ начинает работу над входом, цепочкой входных символов конечной длины на ее ленте, причем остальные клетки на ленте заполнены пробелами. Пробел является одним из ленточных символов, и вход образуется из подмножества ленточных символов, не включающего пробел. Эти символы называются входными. МТ допускает свой вход, попадая в допускающее состояние.
- *Рекурсивно-перечислимые языки*. Языки, допускаемые МТ, называются рекурсивно-перечислимыми (РП-языками). Таким образом, РП-языки — это языки, которые могут быть распознаны или допущены вычислительным устройством какого-либо вида.
- *Мгновенные описания МТ*. Текущую конфигурацию МТ можно описать цепочкой конечной длины, которая включает все клетки ленты между крайними слева и справа значащими символами (отличными от пробела). Состояние и позиция головки указываются помещением состояния в последовательность ленточных символов непосредственно слева от обозреваемой клетки.
- *Запоминание в конечном управлении*. Иногда построение МТ для некоторого языка облегчается за счет того, что состояние представляется как имеющее несколько

компонентов. Один из них является управляющим и функционирует, как состояние. Другие компоненты содержат данные, которые нужно запомнить машине.

- *Многодорожечные машины*. Часто полезно рассматривать ленточные символы в виде кортежей с фиксированным числом компонентов. Каждый компонент можно изобразить как элемент отдельной дорожки на ленте.
- *Многоленточные машины Тьюринга*. Расширенная модель МТ имеет некоторое фиксированное число лент (более одной). Переход МТ основан на состоянии и кортеже символов, обозреваемых головками на каждой ленте. За один переход многоленточная МТ изменяет состояние, переписывает символы в клетках, обозреваемых каждой головкой, и сдвигает любую из головок на одну клетку в любом направлении. Многоленточная МТ способна распознавать только рекурсивно перечислимые языки, хотя делает это быстрее, чем обычная одноленточная.
- *Недетерминированные машины Тьюринга*. НМТ имеет конечное число выборов следующего перехода (состояние, новый символ и сдвиг головки) для каждого состояния и обозреваемого символа. Она допускает свой вход, если хотя бы одна последовательность выборов ведет в МО с допускающим состоянием. Хотя НМТ и кажется более мощной, чем детерминированная МТ, она также распознает лишь рекурсивно перечислимые языки.
- *Машины Тьюринга с односторонней лентой*. МТ можно ограничить так, чтобы ее лента была бесконечной только справа и не имела клеток слева от начальной позиции головки. Такая МТ может допустить любой рекурсивно перечислимый язык.
- *Многомагазинные (мультистековые) машины Тьюринга*. Ленты многоленточной МТ можно ограничить так, чтобы они вели себя, как магазины. Вход размещен на отдельной ленте и читается один раз слева направо, как у конечных или МП-автоматов. Одномагазинная машина в действительности является ДМП-автоматом, хотя машина с двумя магазинами может допустить любой рекурсивно перечислимый язык.
- *Счетчиковые машины*. Магазины мультистековых машин можно ограничить вообще одним символом, отличным от маркера дна магазина. Таким образом, каждый магазин функционирует, как счетчик, позволяющий хранить неотрицательное целое и проверять, совпадает ли хранимое целое с 0, но ничего более. Машины с двумя счетчиками достаточно для того, чтобы допустить любой рекурсивно перечислимый язык.
- *Имитация машины Тьюринга на реальном компьютере*. Имитация МТ на компьютере в принципе возможна, если допустить, что для имитации значащей части ленты существует потенциально бесконечный запас сменных запоминающих устройств вроде диска. Поскольку физические ресурсы, необходимые для создания дисков, конечны, данный довод сомнителен. Однако, поскольку пределы памяти Вселенной неизвестны или, без сомнения, обширны, предположение о бесконечности ресурсов (как для ленты МТ) является практически реалистичным и в целом допустимо.

- *Имитация компьютера на машине Тьюринга*. МТ может имитировать память и управление реального компьютера путем использования одной ленты для записи всех элементов памяти и их содержимого — регистров, основной памяти, дисков и других запоминающих устройств. Таким образом, можно быть уверенным, что все, не выполнимое машиной Тьюринга, не может быть сделано и компьютером.

Литература

Понятие машины Тьюринга взято из [8]. Приблизительно в то же самое время для описания вычислимости было предложено несколько других, менее машиноподобных моделей (работы Черча [1], Клини [5] и Поста [7]). Всем им предшествовала работа Геделя [3], которая в конечном счете показывала, что с помощью компьютера ответить на все математические вопросы невозможно.

Изучение многоленточных МТ и, в частности, рассмотрение вопроса о том, как их время работы сравнивается с одноленточной моделью, началось в статье [4]. Исследование многомагазинных и счетчиковых машин исходит из [6], хотя конструкции, приведенные здесь, взяты из [2].

Метод использования “hello, world” как заменителя допускания с помощью машины Тьюринга или ее останова появился в неопубликованных записках Рудича (S. Rudich).

1. A. Church, “An undecidable problem in elementary number theory”, *American J. Math*. **58** (1936), pp. 345–363.
2. P. C. Fischer, “Turing machines with restricted memory access”, *Information and Control* **9**:4 (1966), pp. 364–379.
3. K. Goedel, “Uber formal unentscheidbare satze der Principia Mathematica und verwander systeme”, *Monatschefte fur Mathematic und Physik* **38** (1931), pp. 173–198.
4. J. Hartmanis and R. E. Stearns, “On the computational complexity of algorithms”, *Transactions of the AMS* **117** (1965), pp. 285–306.
5. S. C. Kleene, “General recursive functions of natural numbers”, *Mathematische Annalen* **112** (1936), pp. 727–742.
6. M. L. Minsky, “Recursive unsolvability of Post’s problem of ‘tag’ and other topics in the theory of Turing machines”, *Annals of Mathematics* **74**:3 (1961), pp. 437–455.
7. E. Post, “Finite combinatory processes — formulation I”, *J. Symbolic Logic* **1** (1936), pp. 103–105. (Пост Э.Л. Финитные комбинаторные процессы — формулировка 1.// Успенский В.А. Машина Поста. — М.: Наука. — С. 89–95.)
8. A. M. Turing, “On computable numbers with an application to the Entscheidungsproblem”, *Proc. London Math. Society* **2**:42 (1936), pp. 230-265. *ibid*. **2**:43, pp. 544–546.

ГЛАВА 9

Неразрешимость

Эта глава начинается повторением рассуждений из раздела 8.1, которые неформально обосновывали существование проблем, не разрешимых с помощью компьютера. Недостатком тех “правдоподобных рассуждений” было вынужденное пренебрежение реальными ограничениями, возникающими при реализации языка C (или другого языка программирования) на любом реальном компьютере. Однако эти ограничения, вроде конечности адресного пространства, не являются фундаментальными. Наоборот, с течением времени мы вправе ожидать от компьютеров неограниченного роста таких количественных характеристик, как размеры адресного пространства, оперативной памяти и т.п.

Сосредоточившись на машинах Тьюринга, для которых ограничения такого рода отсутствуют, можно лучше понять главное — принципиальные возможности вычислительных устройств, если не сегодня, то в перспективе. В этой главе дается формальное доказательство того, что никакая машина Тьюринга не может решить следующую задачу.

- Допускает ли данная машина Тьюринга сама себя (свой код) в качестве входа?

Из раздела 8.6 известно, что на машинах Тьюринга можно имитировать работу реальных компьютеров, даже не имеющих сегодняшних ограничений. Поэтому независимо от того, насколько ослаблены эти практические ограничения, у нас будет строгое доказательство, что указанную задачу нельзя решить с помощью компьютера.

Далее мы разделим проблемы, разрешимые с помощью машин Тьюринга, на два класса. В первый класс войдут те из них, которые имеют *алгоритм* решения (т.е. машину Тьюринга, которая останавливается независимо от того, допускает она свой вход или нет). Во второй класс войдут проблемы, разрешимые лишь с помощью таких машин Тьюринга, которые с недопустимыми входами могут работать бесконечно. В последнем случае проверить допустимость входа весьма проблематично, поскольку независимо от того, сколь долго работает МТ, неизвестно, допускается вход или нет. Поэтому мы сосредоточимся на методах доказательства “неразрешимости” проблем, т.е. что для них не существует алгоритма решения независимо от того, допускаются ли они машиной Тьюринга, которая не останавливается на некоторых входах, или нет.

Будет доказано, что неразрешима следующая проблема.

- Допускает ли данная машина Тьюринга данный вход?

Затем из неразрешимости этой проблемы будет выведена неразрешимость ряда других проблем. Например, будет показано, что все нетривиальные задачи, связанные с языком машины Тьюринга, неразрешимы, как и многие проблемы, в которых речь вообще не идет о машинах Тьюринга, программах или компьютерах.

9.1. Неперечислимый язык

Напомним, что язык L является *рекурсивно-перечислимым* (РП-языком), если $L = L(M)$ для некоторой МТ M. В разделе 9.2 будут введены так называемые "рекурсивные", или "разрешимые", языки, которые не только рекурсивно перечислимы, но и допускаются некоторой МТ, останавливающейся на всех своих входах независимо от их допустимости.

Наша дальняя цель — доказать неразрешимость языка, состоящего из пар (M, w), которые удовлетворяют следующим условиям.

1. M — машина Тьюринга (в виде соответствующего двоичного кода) с входным алфавитом $\{0, 1\}$.
2. w — цепочка из символов 0 и 1.
3. M допускает вход w.

Если эта проблема неразрешима при условии, что алфавит — двоичный, то она, безусловно, будет неразрешимой и в более общем виде, при произвольном входном алфавите.

Прежде всего, необходимо сформулировать эту проблему как вопрос о принадлежности некоторому конкретному языку. Поэтому нужно определить такое кодирование машин Тьюринга, в котором использовались бы только символы 0 и 1 независимо от числа состояний МТ. Имея такие коды, можно рассматривать любую двоичную цепочку как машину Тьюринга. Если цепочка не является правильным представлением какой-либо МТ, то ее можно считать представлением МТ без переходов.

Для достижения промежуточной цели, представленной в данном разделе, используется "язык диагонализации" L_d. Он состоит из всех цепочек w, каждая из которых не допускается машиной Тьюринга, представленной этой цепочкой. Мы покажем, что такой машины Тьюринга, которая бы допускала L_d, вообще не существует. Напомним: доказать, что язык не допускается никакой машиной Тьюринга, значит доказать нечто большее, чем то, что данный язык неразрешим (т.е. что для него не существует алгоритма, или МТ, которая останавливается на всех своих входах).

Роль языка L_d аналогична роли гипотетической программы H_2 из раздела 8.1.2, которая печатает `hello, world`, если ей на вход подается программа, *не печатающая* `hello, world` при чтении собственного кода. Точнее, как не существует H_2, так и L_d не может быть допущен никакой машиной Тьюринга. Реакция первой на саму себя как на вход приводит к противоречию. Аналогично, если бы L_d допускался некоторой машиной Тьюринга, то она противоречила бы самой себе при обработке своего собственного кода.

9.1.1. Перечисление двоичных цепочек

В дальнейшем нам понадобится приписать всем двоичным цепочкам целые числа так, чтобы каждой цепочке соответствовало одно целое число и каждому числу — одна цепочка. Если w — двоичная цепочка, то $1w$ рассматривается как двоичное представление целого числа i. Тогда w называется i-й цепочкой. Таким образом, ε есть первая цепочка,

0 — вторая, 1 — третья, 00 — четвертая, 01 — пятая, и так далее. Это равносильно тому, что цепочки упорядочены по длине, а цепочки равной длины упорядочены лексикографически. В дальнейшем *i*-я цепочка обозначается через w_i.

9.1.2. Коды машин Тьюринга

Наша следующая цель — разработать для машин Тьюринга такой код, чтобы всякую МТ с входным алфавитом {0, 1} можно было рассматривать как двоичную цепочку. Мы только что показали, как перечислить двоичные цепочки. Поэтому у нас есть идентификация машин Тьюринга целыми числами, и можно говорить об "*i*-ой машине Тьюринга M_i". Для того чтобы представить МТ $M = (Q, \{0, 1\}, \Gamma, \delta, q_1, B, F)$ как двоичную цепочку, нужно приписать целые числа состояниям, ленточным символам и направлениям L и R.

- Будем предполагать, что состояниями являются $q_1, q_2, \ldots, q_r$ при некотором r. Состояние q_1 всегда будет начальным, а q_2 — единственным допускающим. Заметим, что поскольку можно считать, что МТ останавливается, попадая в допускающее состояние, то одного такого состояния всегда достаточно.
- Предположим, что ленточными символами являются $X_1, X_2, \ldots, X_s$ при некотором s. X всегда будет символом 0, X_2 — 1, а X_3 — B, пробелом (пустым символом). Остальным же ленточным символам целые числа могут быть приписаны произвольным образом.
- Направление L обозначается как D_1, а направление R — как D_2.

Поскольку состояниям и ленточным символам любой МТ M можно приписать целые числа не единственным образом, то и кодировок МТ будет, как правило, более одной. Но в дальнейшем это не будет играть роли, так как мы покажем, что МТ M с $L(M) = L_d$ вообще непредставима никаким кодом.

Установив, какие целые числа соответствуют каждому из состояний, символов и направлений, можно записать код функции переходов δ. Пусть $\delta(q_i, X_j) = (q_k, X_l, D_m)$ есть одно из правил перехода, где i, j, k, l, m — некоторые целые числа. Это правило кодируется цепочкой $0^i10^j10^k10^l10^m$. Заметим, что каждое из чисел i, j, k, l, m не меньше единицы, так что в коде каждого отдельного перехода нет двух 1 подряд.

Код МТ M состоит из кодов всех ее переходов, расположенных в некотором порядке и разделенных парами единиц:

$$C_1 11 C_2 11 \ldots C_{n-1} 11 C_n,$$

где каждое C означает код одного перехода M.

Пример 9.1. Рассмотрим МТ

$$M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\}),$$

где функция δ состоит из следующих правил.

$\delta(q_1, 1) = (q_3, 0, R)$
$\delta(q_3, 0) = (q_1, 1, R)$
$\delta(q_3, 1) = (q_2, 0, R)$
$\delta(q_3, B) = (q_3, 1, L)$

Переходы кодируются соответственно следующим образом.

0100100010100
0001010100100
00010010010100
0001000100010010

Первое правило, например, можно записать как $\delta(q_1, X_2) = (q_3, X_1, D_2)$, поскольку $1 = X_2$, $0 = X_1$ и $R = D_2$. Поэтому его кодом будет цепочка $0^1 10^2 10^3 10^1 10^2$, как и записано выше. Кодом всей M является следующая цепочка.

010010001010011000101010010011000100100101001100010001000100010010

Заметим, что существуют другие возможные коды машины M. В частности, коды четырех ее переходов можно переставить 4! способами, что дает 24 кода для M. □

В разделе 9.2.3 нам понадобится кодировать пары вида (M, w), состоящие из МТ и цепочки. В качестве такого кода последовательно записываются код M, 111 и цепочка w. Поскольку правильный код МТ не может содержать три единицы подряд, первое вхождение 111 гарантированно отделяет код M от w. Например, если M — это МТ из примера 9.1, а w — цепочка 1011, то кодом (M, w) будет цепочка, представленная в конце примера 9.1, с дописанной к ней последовательностью 1111011.

9.1.3. Язык диагонализации

В разделе 9.1.2 были определены коды машин Тьюринга, и теперь у нас есть вполне конкретное понятие M_i, "i-й машины Тьюринга". Это МТ M, кодом которой является i-я двоичная цепочка w_i. Многие целые числа вообще не соответствуют машинам Тьюринга. Например, 11001 не начинается с 0, а цепочка 0010111010010100 содержит три 1 подряд. Если w_i не является правильным кодом МТ, то будем считать, что M_i — МТ, у которой одно состояние и нет переходов, т.е. M_i для этих значений i есть МТ, которая сразу останавливается на любом входе. Итак, если w_i не является правильным кодом МТ, то $L(M_i)$ есть $\varnothing$.

Теперь можно дать весьма существенное определение.

- *Язык диагонализации* L_d — это множество всех цепочек w_i, не принадлежащих $L(M_i)$.

Таким образом, L_d состоит из всех цепочек w, каждая из которых не допускается соответствующей машиной Тьюринга с кодом w.

Из рис. 9.1 видно, почему L_d называется языком "диагонализации". Для всех i и j эта таблица показывает, допускает ли МТ M_i входную цепочку w_j; 1 означает "да, допуска-

ет", а 0 — "нет, не допускает"[1]. Можно считать i-ю строку таблицы *характеристическим вектором* языка $L(M_i)$; единицы в этой строке указывают на цепочки, которые принадлежат данному языку.

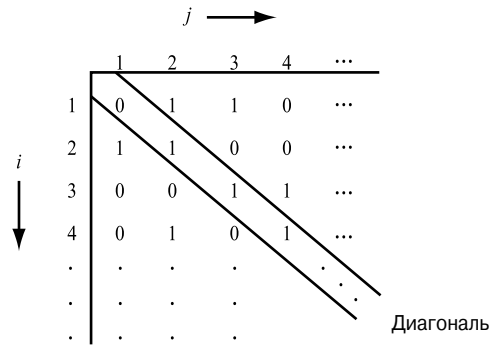


Рис. 9.1. Таблица, представляющая допустимость цепочек машинами Тьюринга

Числа на диагонали показывают, допускает ли M_i цепочку w_i. Чтобы построить язык L_d, нужно взять дополнение диагонали. Например, если бы таблица на рис. 9.1 была корректной, то дополнение диагонали имело бы начало 1, 0, 0, 0, …. Таким образом, L_d должен был бы содержать $w_1 = \varepsilon$ и не содержать цепочки с w_2 по w_4, т.е. 0, 1 и 00 и т.д.

Операция с дополнением диагонали для построения характеристического вектора языка, которому не может соответствовать никакая строка, называется *диагонализацией*. Она приводит к желаемому результату, поскольку дополнение диагонали само по себе является характеристическим вектором, описывающим принадлежность некоторому языку, а именно — L_d. Действительно, этот характеристический вектор отличается от каждой из строк таблицы на рис. 9.1 хотя бы в одном столбце. Поэтому дополнение диагонали не может быть характеристическим вектором никакой машины Тьюринга.

9.1.4. Доказательство неперечислимости L_d

Развивая наши рассуждения о характеристических векторах и диагонали, формально докажем основной результат, касающийся машин Тьюринга, — МТ, допускающей язык L_d, не существует.

Теорема 9.2. Язык L_d не является рекурсивно-перечислимым, т.е. не существует машины Тьюринга, которая допускала бы L_d.

Доказательство. Допустим, что $L_d = L(M)$ для некоторой МТ M. Так как L_d — язык над алфавитом $\{0, 1\}$, M должна содержаться в построенной нами последовательности машин

[1] Заметим, что в действительности эта таблица выглядит совсем не так, как на рисунке. Ее верхние строки должны быть заполнены нулями, поскольку все небольшие целые числа не могут быть правильными кодами МТ.

Тьюринга, поскольку эта последовательность содержит все МТ с входным алфавитом $\{0, 1\}$. Следовательно, в ней есть, по крайней мере, один код машины M, скажем, i, т.е. $M = M_i$.

Выясним теперь, принадлежит ли w_i языку L_d.

- Если w_i принадлежит L_d, то M_i допускает w_i. Но тогда (по определению L_d) w_i не принадлежит L_d, так как L_d содержит лишь такие w_j, для которых M_j *не допускает* w_j.
- Точно так же, если w_i не принадлежит L_d, то M_i не допускает w_i. Но тогда (по определению L_d) w_i *принадлежит* L_d.

Поскольку w_i не может одновременно и принадлежать, и не принадлежать L_d, приходим к противоречию с нашим предположением о том, что M существует. Таким образом, L_d не является рекурсивно-перечислимым языком. □

9.1.5. Упражнения к разделу 9.1

9.1.1. Запишите следующие цепочки:

а) (*) w_{37};

б) w_{100}.

9.1.2. Запишите один из возможных кодов машины Тьюринга, изображенной на рис. 8.9.

9.1.3. (!) Ниже приводятся определения двух языков, похожих на L_d, но все же отличных от него. Используя операции типа диагонализации, покажите, что каждый из них не может допускаться никакой машиной Тьюринга. Отметим, что при этом нужно использовать не диагональ, а какую-то другую бесконечную последовательность элементов матрицы, изображенной на рис. 9.1:

а) (*) множество всех w_i, для которых M_{2i} не допускает w_i;

б) множество всех w_i, для которых M_i не допускает w_{2i}.

9.1.4. (!) Машины Тьюринга, рассмотренные до сих пор, имели входной алфавит $\{0, 1\}$. Допустим, мы хотели бы приписать целые числа всем машинам Тьюринга, независимо от их входного алфавита. Это, вообще говоря, невозможно. Ведь в то время, как имена состояний или рабочие символы на ленте (которые не являются входными), произвольны, каждый входной символ имеет значение. Например, языки $\{0^n1^n \mid n \geq 1\}$ и $\{a^nb^n \mid n \geq 1\}$ похожи, однако не совпадают и допускаются разными МТ. Допустим тем не менее, что у нас есть бесконечное множество символов $\{a_1, a_2, \ldots\}$, из которого выбираются все алфавиты МТ. Покажите, как можно приписать целые числа всем МТ, входной алфавит которых является конечным подмножеством этих символов.

9.2. Неразрешимая РП-проблема

Итак, мы убедились, что существует проблема (язык диагонализации L_d), которая не допускается никакой машиной Тьюринга. Наша следующая цель — уточнить структуру ре-

курсивно-перечислимых языков (РП-языков), т.е. допустимых машинами Тьюринга, разбив их на два класса. В первый класс войдут языки, которым соответствует то, что обычно понимается как алгоритм, т.е. МТ, которая не только распознает язык, но и сообщает, что входная цепочка не принадлежит этому языку. Такая машина Тьюринга в конце концов всегда останавливается независимо от того, достигнуто ли допускающее состояние.

Второй класс языков состоит из тех РП-языков, которые не допускаются никакой машиной Тьюринга, останавливающейся на всех входах. Эти языки допускаются несколько неудобным образом: если входная цепочка принадлежит языку, то мы в конце концов об этом узнаем, но если нет, то машина Тьюринга будет работать вечно. Поэтому никогда нельзя быть уверенным, что данная входная цепочка не будет когда-нибудь допущена. Примером языка данного типа, как будет показано ниже, является множество таких закодированных пар (M, w), в которых МТ M допускает вход w.

9.2.1. Рекурсивные языки

Язык L называется *рекурсивным*, если $L = L(M)$ для некоторой машины Тьюринга M, удовлетворяющей следующим условиям.

1. Если w принадлежит L, то M попадает в допускающее состояние (и, следовательно, останавливается).
2. Если w не принадлежит L, то M в конце концов останавливается, хотя и не попадает в допускающее состояние.

МТ этого типа соответствует интуитивному понятию “алгоритма” — правильно определенной последовательности шагов, которая всегда заканчивается и приводит к некоторому ответу. Если мы рассматриваем язык L как “проблему”, то проблема L называется *разрешимой*, если она является рекурсивным языком. В противном случае проблема называется *неразрешимой*.

Существование алгоритма зачастую важнее, чем наличие некоторой МТ, решающей данную проблему. Как упоминалось ранее, машины Тьюринга, которые могут не останавливаться, не дают информации, необходимой для утверждения, что цепочка не принадлежит языку, т.е. они “не решают проблему”. Таким образом, разделение проблем и языков на разрешимые (для них существует алгоритм решения) и неразрешимые часто важнее, чем разделение на рекурсивно перечислимые (для них существует МТ какого-либо типа) и неперечислимые (для них вообще не существует МТ). На рис. 9.2 представлено соотношение трех классов языков.

1. Рекурсивные языки.
2. Языки, которые рекурсивно перечислимы (РП), но не рекурсивны.
3. Неперечислимые (*не-РП*) языки.

На рисунке указаны правильные положения не РП-языка L_d и “универсального языка” L_u, который, как мы вскоре докажем, является РП, но не рекурсивным.

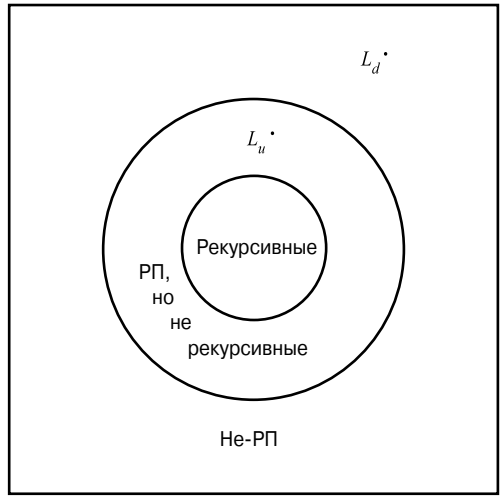


Рис. 9.2. Соотношение рекурсивных, неперечислимых и РП-языков

Почему “рекурсивные”?

Современные программисты знакомы с понятием рекурсивной функции. Остается непонятным, что общего между рекурсивными функциями и машинами Тьюринга, которые всегда останавливаются. Еще хуже, что нерекурсивными, или неразрешимыми, называются языки, которые не распознаются никаким алгоритмом, хотя под “нерекурсивными” мы привыкли понимать вычисления настолько простые, что они не требуют обращений к рекурсивным функциям.

Термин “рекурсивный” как синоним слова “разрешимый” возник в период развития математики, предшествовавший появлению компьютеров. В качестве понятия вычисления обычно использовались формализмы, основанные на рекурсии (но не итерации или цикле). В этих системах понятий (не рассматриваемых здесь) было нечто от вычислений в таких языках функционального программирования, как LISP или ML. В этом смысле выражение “проблема рекурсивна” означало, что она “проста настолько, что можно записать рекурсивную функцию, которая всегда приводит к ее решению за конечное число шагов”. И в наши дни применительно к машинам Тьюринга этот термин имеет в точности тот же смысл.

Термин “рекурсивно-перечислимый” — из того же семейства понятий. С помощью некоторой функции элементы языка можно выписать в некотором порядке, т.е. “перечислить” их. Языки, элементы которых можно перечислить в некотором порядке, — это именно те языки, которые допускаются некоторой МТ (возможно, работающей бесконечно на недопустимых входах).

9.2.2. Дополнения рекурсивных и РП-языков

Для доказательства того, что некоторый язык принадлежит второму кольцу на рис. 9.2 (т.е. является РП, но не рекурсивным), часто используется дополнение этого языка. Покажем, что рекурсивные языки замкнуты относительно дополнения. Поэтому, если язык L является РП, а его дополнение $\overline{L}$ — нет, то L не может быть рекурсивным. Если бы L был рекурсивным, то $\overline{L}$ также был бы рекурсивным, а следовательно, и РП. Докажем это важное свойство замкнутости рекурсивных языков.

Теорема 9.3. Если L — рекурсивный язык, то язык $\overline{L}$ также рекурсивен.

Доказательство. Пусть $L = L(M)$ для некоторой всегда останавливающейся МТ M. Согласно схеме на рис. 9.3 построим МТ $\overline{M}$, у которой $\overline{L} = L(\overline{M})$, т.е. $\overline{M}$ ведет себя так же, как и M; изменения касаются лишь допускающих состояний.

1. Допускающие состояния M становятся недопускающими состояниями $\overline{M}$, не имеющими переходов, т.е. в этих состояниях $\overline{M}$ останавливается, не допуская.
2. $\overline{M}$ имеет новое допускающее состояние r, из которого нет переходов.
3. Для каждой комбинации из недопускающего состояния и ленточного символа M, в которой M не имеет перехода (т.е. останавливается, не допуская), добавляется переход в допускающее состояние r.

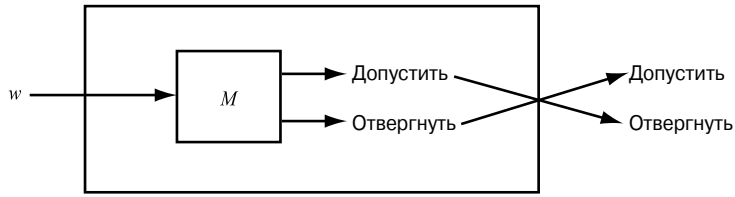


Рис. 9.3. Строение МТ, допускающей дополнение рекурсивного языка

Поскольку M всегда останавливается, то всегда останавливается и $\overline{M}$. Кроме того, $\overline{M}$ допускает множество именно тех цепочек, которые не допускаются M, т.е. $\overline{L}$. □

С дополнениями языков связан еще один важный факт. Он еще больше ограничивает область на диаграмме (см. рис. 9.2), в которую могут попасть язык и его дополнение. Это ограничение утверждается следующей теоремой.

Теорема 9.4. Если язык L и его дополнение являются РП, то L рекурсивен. Отметим, что тогда по теореме 9.3 язык $\overline{L}$ также рекурсивен.

Доказательство. Доказательство представлено на рис. 9.4. Пусть $L = L(M_1)$ и $\overline{L} = L(M_2)$. Параллельная работа машин имитируется МТ M. Чтобы эта модель стала простой и очевидной, можно взять МТ с двумя лентами и затем преобразовать ее в одноленточную. Одна из них имитирует ленту M_1, а вторая — ленту M_2. Состояния M_1 и M_2 являются компонентами состояния M.

Если вход w машины M принадлежит L, то M_1 в конце концов попадает в допускающее состояние; тогда M допускает и останавливается. Когда же в допускающее со-

стояние попадает M_2, M останавливается, не допуская. Таким образом, M останавливается при любом входе, и $L(M)$ — это в точности L. Отсюда заключаем, что язык L — рекурсивный. □

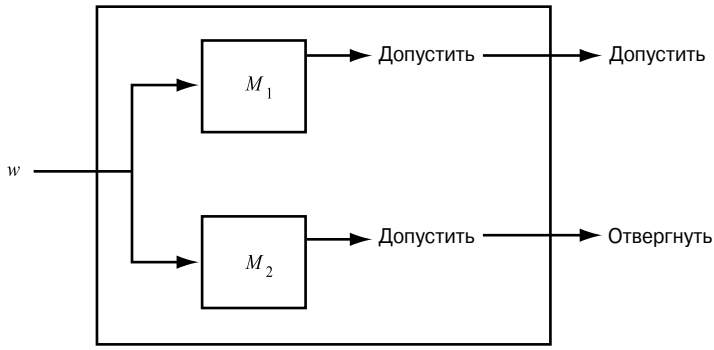


Рис. 9.4. Моделирование двух МТ, одна из которых допускает некоторый язык, а вторая — его дополнение

Теоремы 9.3 и 9.4 можно объединить следующим образом. Из девяти способов расположения языка L и его дополнения $\overline{L}$ (см. диаграмму на рис. 9.2) возможны лишь следующие четыре[2].

1. Оба языка L и $\overline{L}$ рекурсивны, т.е. оба они находятся во внутреннем кольце.
2. Ни L, ни $\overline{L}$ не являются РП, т.е. оба они находятся во внешнем кольце.
3. L является РП, но не рекурсивным, и $\overline{L}$ не является РП, т.е. один находится в среднем кольце, а другой — во внешнем.
4. $\overline{L}$ является РП, но не рекурсивным, и L не является РП, т.е. L и $\overline{L}$ меняются местами по отношению к случаю 3.

В качестве доказательства отметим, что теорема 9.3 исключает возможность того, что один из языков (L или $\overline{L}$) является рекурсивным, а второй принадлежит какому-либо из оставшихся классов. При этом теорема 9.4 исключает возможность того, что оба языка являются РП, но не рекурсивными.

Пример 9.5. Рассмотрим язык L_d, который, как мы знаем, не является РП. Поэтому язык $\overline{L}_d$ не может быть рекурсивным. Однако $\overline{L}_d$ может быть либо не-РП, либо РП, но не рекурсивным. На самом деле верно последнее.

$\overline{L}_d$ есть множество цепочек w_i, допускаемых соответствующими M_i. Этот язык похож на универсальный язык L_u, состоящий из всех пар (M, w), где M допускает w, который, как мы покажем в разделе 9.2.3, является РП. Точно так же можно показать, что $\overline{L}_d$ является РП. □

[2] По существу, их всего три. — *Прим. ред.*

9.2.3. Универсальный язык

В разделе 8.6.2 уже неформально обсуждалось, как можно использовать машину Тьюринга в качестве модели компьютера с загруженной в него произвольной программой. Иными словами, отдельно взятая МТ может использоваться как "компьютер с записанной программой", который считывает свою программу и данные с одной или нескольких лент, содержащих входную информацию. В данном разделе будет формализована идея того, что машина Тьюринга является представлением программы, записанной в память компьютера.

Универсальный язык L_u определяется как множество двоичных цепочек, которые являются кодами (в смысле определения из раздела 9.1.2) пар (M, w), где M — МТ с двоичным входным алфавитом, а w — цепочка из $(\mathbf{0+1})^*$, принадлежащая $L(M)$. Таким образом, L_u — это множество цепочек, представляющих некоторую МТ и допускаемый ею вход. Покажем, что существует МТ U, которую часто называют *универсальной машиной Тьюринга*, для которой $L_u = L(U)$. Поскольку входом U является двоичная цепочка, то в действительности U — это некоторая M_j в списке машин Тьюринга с двоичным входом (см. раздел 9.1.2).

Проще всего U описывается как многоленточная машина Тьюринга в стиле рис. 8.22. В машине U переходы M вначале хранятся на первой ленте вместе с цепочкой w. Вторая лента используется для моделирования ленты машины M, в том же формате, что и у кода M. Таким образом, ленточный символ X_i машины M кодируется как 0^i, а коды разделяются одиночными 1. На третью ленту U записывается состояние M, причем состояние q_i представляется в виде i нулей. Схема U представлена на рис. 9.5.

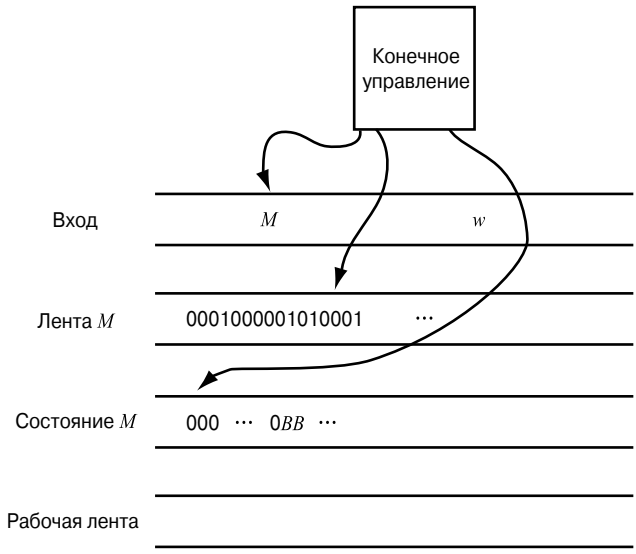


Рис. 9.5. Структура универсальной машины Тьюринга

Машина U производит следующие операции.

1. Исследует вход для того, чтобы убедиться, что код M является правильным кодом МТ. Если это не так, U останавливается, не допуская. Это действие корректно, поскольку неправильные коды по договоренности представляют МТ без переходов, а такие МТ не допускают никакого входа.
2. Записывает на вторую ленту код входной цепочки w. Таким образом, каждому символу 0 из w на второй ленте ставится в соответствие цепочка 10, а каждой 1 — цепочка 100. Отметим, что пустые символы, которые находятся на ленте самой M и представляются цепочками вида 1000, на этой ленте появляться не будут. Все клетки, кроме занятых символами цепочки w, будут заполняться пустыми символами машины U. Но при этом U знает, что если она ищет имитируемый символ M и находит свой собственный пустой символ, то она должна заменить его последовательностью 1000, имитирующей пустой символ M.
3. На третьей ленте записывает 0, т.е. начальное состояние M, и перемещает головку второй ленты U в первую имитируемую клетку.
4. Для того чтобы отобразить переход M, U отыскивает на своей первой ленте переход $0^i10^j10^k10^l10^m$, у которого 0^i есть состояние на ленте 3, а 0^j — ленточный символ M, начинающийся с позиции на ленте 2, обозреваемой U. Это переход, который M совершает на следующем шаге. Машина U должна выполнить следующие действия:
    а) изменить содержимое ленты 3 на 0^k, т.е. отобразить изменение состояния M. Для этого U вначале заменяет все символы 0 на ленте 3 пустыми символами, а затем копирует 0^k с ленты 1 на ленту 3;
    б) заменить 0^j на ленте 2 символами 0^l, т.е. поменять ленточный символ M. Если потребуется больше или меньше места (т.е. $j \neq l$), то для управления пространством используются рабочая лента и техника переноса (см. раздел 8.6.2);
    в) переместить головку на ленте 2 в ту позицию слева или справа, в которой находится ближайший символ 1. При $m = 1$ совершается сдвиг влево, а при $m = 2$ — вправо. Таким образом, U имитирует движение M влево или вправо.
5. Если M не имеет перехода, соответствующего имитируемым состоянию и ленточному символу, то в п. 4 переход не будет найден. Таким образом, в данной конфигурации M останавливается, и то же самое должна делать U.
6. Если M попадает в свое допускающее состояние, то U допускает.

Этими действиями U имитирует работу M со входом w. U допускает закодированную пару (M, w) тогда и только тогда, когда M допускает w.

Более эффективная универсальная МТ

Машина U, эффективно имитирующая M, т.е. не требующая сдвига ленточных символов, должна вначале определять число ленточных символов, используемых M. Если число символов лежит в пределах от 2^{k-1} до $2^k - 1$, то однозначно представить различные ленточные символы можно с помощью k-битового двоичного кода. Каждой клетке M можно поставить в соответствие k клеток машины U. Для того чтобы упростить имитацию, можно переписать в машине U данные переходы M и вместо описанного нами унарного кода переменной длины использовать бинарный код фиксированной длины.

9.2.4. Неразрешимость универсального языка

Представим проблему, которая является РП, но не рекурсивной. Это язык L_u. Неразрешимость L_u (т.е. его нерекурсивность) во многих отношениях важнее, чем наш предыдущий вывод о том, что язык L_d не является РП. Причина состоит в следующем. Сведением L_u к другой проблеме P можно показать, что алгоритма, решающего P, не существует, независимо от того, является ли P РП. Однако сведение L_d к P возможно лишь тогда, когда P не является РП, поэтому L_d не может использоваться при доказательстве неразрешимости проблем, которые являются РП, но не рекурсивными. Вместе с тем, если нужно показать, что проблема не является РП, то можно использовать только L_d, в то время как язык L_u оказывается бесполезным, поскольку он *является* РП.

Теорема 9.6. L_u является РП, но не рекурсивным.

Доказательство. В разделе 9.2.3 доказано, что язык L_u является РП. Допустим, что L_u рекурсивен. Тогда по теореме 9.3 $\overline{L}_u$ (дополнение L_u) — также рекурсивный язык. Но если существует МТ M, допускающая $\overline{L}_u$, то, используя описанный ниже метод, можно построить МТ, допускающую L_d. Поскольку нам известно, что L_d не является РП, приходим к противоречию с предположением, что язык L_u является рекурсивным.

Проблема останова

Проблема останова машины Тьюринга считается подобной проблеме L_u; она также является РП, но не рекурсивной. В действительности, определенная А. М. Тьюрингом машина допускала, не попадая в допускающее состояние, а останавливаясь. Для МТ M можно определить $H(M)$ как множество входов w, на которых M останавливается независимо от того, допускает ли M вход w. Тогда проблема останова состоит в определении множества таких пар (M, w), у которых w принадлежит $H(M)$. Это еще один пример проблемы/языка, которая является РП, но не рекурсивной.

Предположим, что $L(M) = \overline{L}_u$. На рис. 9.6 показано, как можно преобразовать МТ M в МТ M', которая допускает L_d с помощью следующих действий.

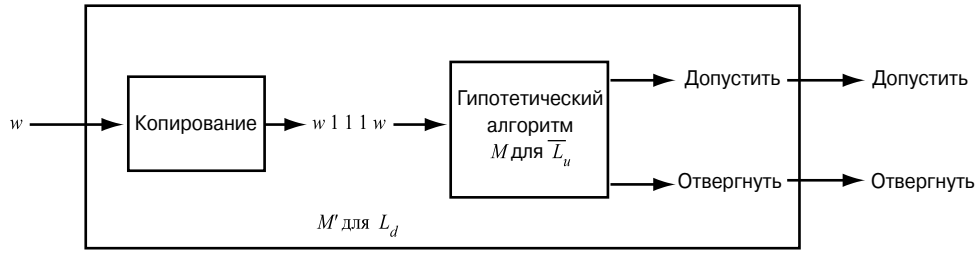


Рис. 9.6. Сведение L_d *к* $\overline{L}_u$

1. *M´* преобразует входную цепочку *w* в *w*111*w*. В качестве упражнения читатель может написать программу для выполнения этого шага на одной ленте. Однако легче это сделать, используя для копии *w* вторую ленту, и затем преобразовать двухленточную МТ в одноленточную.
2. *M´* имитирует *M* на новом входе. Если *w* есть w_i в нашем перечислении, то *M´* определяет, допускает ли M_i вход w_i. Поскольку *M* допускает $\overline{L}_u$, то она допускает тогда и только тогда, когда M_i не допускает w_i, т.е. когда w_i принадлежит L_d.

Таким образом, *M´* допускает *w* тогда и только тогда, когда *w* принадлежит L_d. Поскольку по теореме 9.2 машины *M´* не существует, приходим к выводу, что язык L_u не является рекурсивным. □

9.2.5. Упражнения к разделу 9.2

9.2.1. Покажите, что проблема останова, т.е. проблема определения множества пар (*M*, *w*), где *M* останавливается (допуская или нет) на входе *w*, является РП, но не рекурсивной. (См. врезку “Проблема останова” в разделе 9.2.4.)

9.2.2. Во врезке “Почему “рекурсивные”?” из раздела 9.2.1 говорилось о “рекурсивных функциях” как модели вычислимости, используемой наравне с машинами Тьюринга. В данном упражнении рассмотрим пример записи рекурсивных функций. *Рекурсивная функция* — это функция *F*, определенная конечным набором правил. Каждое правило устанавливает значение функции *F* для определенных аргументов; в правиле могут присутствовать переменные, неотрицательные целочисленные константы, функция прибавления единицы (successor), сама функция *F* и выражения, построенные композицией перечисленных функций. Например, *функция Аккермана* определяется следующим образом.

1. $A(0, y) = 1$ для всех $y \geq 1$.
2. $A(1, 0) = 2$.
3. $A(x, 0) = x + 2$ для $x \geq 2$.
4. $A(x + 1, y + 1) = A(A(x, y + 1), y)$ для любых $x \geq 0$ и $y \geq 0$.

Выполните следующее:

а) (*) вычислите $A(2, 1)$;

б) (!) опишите $A(x, 2)$ как функцию от x;

в) (!) вычислите $A(4, 3)$.

9.2.3. Опишите неформально многоленточные машины Тьюринга, которые *перечисляют* следующие множества целых чисел в том смысле, что, начав с пустых лент, они печатают на одной из лент цепочку $10^{i_1}10^{i_2}1\ldots$, представляющую множество $\{i_1, i_2, \ldots\}$:

а) (*) множество квадратов целых чисел $\{1, 4, 9, \ldots\}$;

б) множество простых чисел $\{2, 3, 5, 7, 11, \ldots\}$;

в) (!!) множество всех таких i, для которых M_i допускает w_i. *Указание*. Такие i невозможно генерировать в порядке возрастания. Причина состоит в том, что данный язык, который есть $\overline{L}_d$, является РП, но не рекурсивным. Такие языки по определению можно перечислить, но не в порядке возрастания. "Прием" с их перечислением состоит в том, чтобы смоделировать работу всех M_i со входами w_i. При этом мы не можем позволить какой-либо M_i работать вечно, поскольку это помешает продолжать проверку для других M_j, где $i \neq j$. Поэтому мы вынуждены работать поэтапно, проверяя на k-м этапе лишь конечное множество машин M_i и ограничивая проверку каждой машины конечным числом шагов. Таким образом, каждый этап проверки будет выполнен за конечное время. Поскольку для всякой МТ M_i и всякого числа шагов s найдется такой этап, на котором будет промоделировано не менее s шагов M_i, то в конце концов мы обнаружим все M_i, допускающие w_i, и перечислим номера i.

9.2.4. (*) Пусть $L_1, L_2, \ldots, L_k$ — набор языков в алфавите Σ, для которого верны следующие утверждения.

1. $L_i \cap L_j \neq \varnothing$ для всех $i \neq j$, т.е. никакая цепочка не принадлежит сразу двум языкам.
2. $L_1 \cup L_2 \cup \cdots \cup L_k = \Sigma^*$, т.е. каждая цепочка принадлежит одному из этих языков.
3. Все языки L_i, где $i = 1, 2, \ldots, k$, рекурсивно перечислимы.

Докажите, что тогда каждый из этих языков рекурсивен.

9.2.5. (*!) Пусть L рекурсивно перечислим и $\overline{L}$ не является РП. Рассмотрим язык

$L' = \{0w \mid w \text{ принадлежит } L\} \cup \{1w \mid w \text{ не принадлежит } L\}$.

Можете ли вы наверняка сказать, что язык L' или его дополнение являются рекурсивными, РП или не-РП? Ответ обоснуйте.

9.2.6. (!) За исключением операции дополнения в разделе 9.2.2, свойства замкнутости рекурсивных и РП-языков пока не обсуждались. Выясните, замкнуты ли рекурсивные и/или РП-языки относительно перечисленных ниже операций:

а) (*) объединение;

б) пересечение;

в) конкатенация;

г) замыкание Клини (звездочка);

д) (*) гомоморфизм;

е) обратный гомоморфизм.

Обосновывая замкнутость, можно использовать неформальные, но достаточно понятные конструкции.

9.3. Неразрешимые проблемы, связанные с машинами Тьюринга

Статус языков L_u и L_d относительно неразрешимости и рекурсивной перечислимости нам уже известен. Используем их для демонстрации других неразрешимых или не РП-языков. В доказательствах используется техника сведения. Все неразрешимые проблемы, которые рассматриваются вначале, связаны с машинами Тьюринга. Кульминацией данного раздела является доказательство "теоремы Райса". В ней утверждается, что любое нетривиальное свойство МТ, зависящее лишь от языка, допускаемого машиной Тьюринга, должно быть неразрешимым. Материал раздела 9.4 позволяет исследовать некоторые неразрешимые проблемы, не связанные ни с машинами Тьюринга, ни с их языками.

9.3.1. Сведения

Понятие сведения было введено в разделе 8.1.3. В общем случае, если у нас есть алгоритм, преобразующий экземпляры проблемы P_1 в экземпляры проблемы P_2, которые имеют тот же ответ (да/нет), то говорят, что P_1 *сводится к* P_2. Доказательство такой сводимости используется, чтобы показать, что P_2 не менее трудна, чем P_1. Таким образом, если P_1 не является рекурсивной, то и P_2 не может быть рекурсивной. Если P_1 не является РП, то и P_2 не может быть РП. Как уже упоминалось в разделе 8.1.3, чтобы доказать, что проблема P_2 не менее трудна, чем некоторая известная P_1, нужно свести P_1 к P_2, а не наоборот.

Как показано на рис. 9.7, сведение должно переводить всякий экземпляр P_1 с ответом "да" (позитивный) в экземпляр P_2 с ответом "да", и всякий экземпляр P_1 с ответом "нет" (негативный) — в экземпляр P_2 с ответом "нет". Отметим, что неважно, является ли каж-

дый экземпляр P_2 образом одного или нескольких экземпляров P_1. В действительности, обычно лишь небольшая часть экземпляров P_2 образуется в результате сведения.

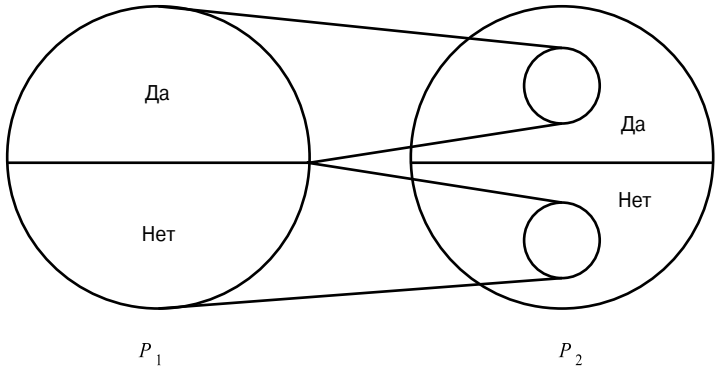


Рис. 9.7. Сведение переводит позитивные экземпляры в позитивные, а негативные — в негативные

Формально сведение P_1 к P_2 является машиной Тьюринга, которая начинает работу с экземпляром проблемы P_1 на ленте и останавливается, имея на ленте экземпляр проблемы P_2. Как правило, сведения описываются так, как если бы они были компьютерными программами, которые на вход получают экземпляры P_1 и выдают экземпляры P_2. Эквивалентность машин Тьюринга и компьютерных программ позволяет описывать сведение в терминах как тех, так и других. Значимость процедуры сведения подчеркивается следующей широко используемой теоремой.

Теорема 9.7. Если P_1 можно свести к P_2, то:

а) если P_1 неразрешима, то и P_2 неразрешима;

б) если P_1 не-РП, то P_2 также не-РП.

Доказательство. Предположим сначала, что проблема P_1 неразрешима. Если решить P_2 возможно, то, объединяя сведение P_1 к P_2 и алгоритм решения последней, можно построить алгоритм, решающий P_1. Эта идея была проиллюстрирована на рис. 8.7. Поясним подробнее. Пусть дан экземпляр w проблемы P_1. Применим к нему алгоритм, который переводит w в экземпляр x проблемы P_2. Затем применим к x алгоритм, решающий P_2. Если алгоритм выдает ответ “да”, то x принадлежит P_2. Поскольку P_1 была сведена к P_2, то известен ответ “да” для P_1 на w, т.е. w принадлежит P_1. Точно так же, если x не принадлежит P_2, то и w не принадлежит P_1. Итак, ответ на вопрос “x принадлежит P_2?” является также правильным ответом на вопрос “w принадлежит P_1?”.

Таким образом, получено противоречие с предположением, что проблема P_1 неразрешима. Делаем вывод, что если P_1 неразрешима, то и P_2 неразрешима.

Рассмотрим теперь часть *б*. Предположим, что P_2 (в отличие от P_1) является РП. В данном случае у нас есть алгоритм сведения P_1 к P_2, но для P_2 существует лишь процедура распознавания, т.е. МТ, которая дает ответ “да”, когда ее вход принадлежит P_2, и мо-

жет работать бесконечно, когда вход не принадлежит P_2. Как и в части *a*, с помощью алгоритма сведения преобразуем экземпляр *w* проблемы P_1 в экземпляр *x* проблемы P_2. Затем применим к *x* МТ для P_2. Если *x* допускается, то допустим и *w*.

Эта процедура описывает МТ (возможно, работающую бесконечно), языком которой является P_1. Если *w* принадлежит P_1, то *x* принадлежит P_2, и поэтому данная МТ допускает *w*. Если же *w* не принадлежит P_1, то *x* не принадлежит P_2. Тогда МТ может или остановиться, или работать бесконечно, но в обоих случаях она не допустит *w*. Поскольку по предположению МТ, распознающей P_1, не существует, то полученное противоречие доказывает, что МТ для P_2 также не существует. Таким образом, если P_1 не-РП, то и P_2 не-РП. □

9.3.2. Машины Тьюринга, допускающие пустой язык

В качестве примера сведения, связанного с машинами Тьюринга, исследуем языки, известные как L_e и L_{ne}. Оба они состоят из двоичных цепочек. Если *w* — двоичная цепочка, то она представляет некоторую МТ M_i из перечисления, описанного в разделе 9.1.2.

Если $M_i = \varnothing$, т.е. M_i не допускает никакого входа, то M_i принадлежит L_e. Таким образом, L_e состоит из кодов всех МТ, языки которых пусты. С другой стороны, если $L(M_i)$ не пуст, то *w* принадлежит L_{ne}. Таким образом, L_{ne} состоит из кодов всех машин Тьюринга, которые допускают хотя бы одну цепочку.

В дальнейшем нам будет удобно рассматривать цепочки как машины Тьюринга, представленные этими цепочками. Итак, упомянутые языки определяются следующим образом.

- $L_e = \{M \mid L(M) = \varnothing\}$
- $L_{ne} = \{M \mid L(M) \neq \varnothing\}$

Отметим, что L_e и L_{ne} — языки над алфавитом $\{0, 1\}$, дополняющие друг друга. Как будет видно, L_{ne} является “более легким”. Он РП, но не рекурсивный, в то время как L_e — не-РП.

Теорема 9.8. *Язык* L_{ne} рекурсивно перечислим.

Доказательство. Достаточно предъявить МТ, допускающую L_{ne}. Проще всего это сделать, описав недетерминированную МТ *M*, которая схематично изображена на рис. 9.8. Согласно теореме 8.11 *M* можно преобразовать в детерминированную МТ.

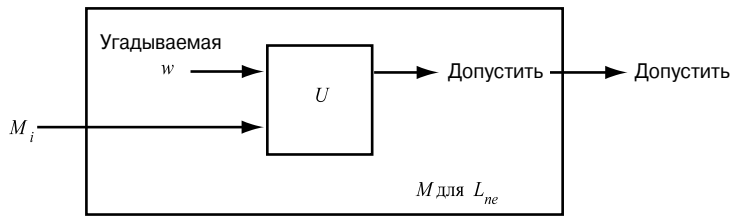


Рис. 9.8. Конструкция НМТ, допускающей язык L_{ne}

Работа M заключается в следующем.

1. На вход M подается код МТ M_i.
2. Используя недетерминизм, M угадывает вход w, который, возможно, допускается M_i.
3. M проверяет, допускает ли M_i свой вход w. В этой части M может моделировать работу универсальной МТ U, допускающей L_u.
4. Если M_i допускает w, то и M допускает свой вход, т.е. M_i.

Таким образом, если M_i допускает хотя бы одну цепочку, то M угадает ее (среди прочих, конечно) и допустит M_i. Если же $L(M_i) = \varnothing$, то ни одна из угаданных w не допускается M_i, и M не допустит M_i. Таким образом, $L(M) = L_{ne}$. □

Следующим шагом будет доказательство, что язык L_{ne} не является рекурсивным. Для этого сведем к L_{ne} язык L_u, т.е. опишем алгоритм, который преобразует вход (M, w) в выход M'— код еще одной машины Тьюринга, для которой w принадлежит $L(M)$ тогда и только тогда, когда язык $L(M')$ не пуст. Таким образом, M допускает w тогда и только тогда, когда M' допускает хотя бы одну цепочку. Прием состоит в том, что M' игнорирует свой вход, а вместо этого моделирует работу M на w. Если M допускает w, то M' допускает свой собственный вход. Таким образом, $L(M')$ не пуст тогда и только тогда, когда M допускает w. Если бы L_{ne} был рекурсивным, то у нас был бы алгоритм выяснения, допускает ли M вход w: построить M' и проверить, будет ли $L(M') = \varnothing$.

Теорема 9.9. Язык L_{ne} не является рекурсивным.

Доказательство. Уточним доказательство, кратко описанное выше. Нужно построить алгоритм, который преобразует свой вход — двоичный код пары (M, w) — в МТ M', для которой $L(M') \neq \varnothing$ тогда и только тогда, когда M допускает вход w. Данная конструкция схематично представлена на рис. 9.9. Как будет показано, если M не допускает w, то M' не допускает никакого входа, т.е. $L(M') = \varnothing$. Но если M допускает w, то M' допускает любой вход, и поэтому, конечно же, $L(M') \neq \varnothing$.

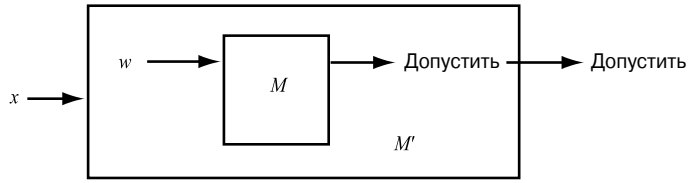


Рис. 9.9. Схема МТ M′, *построенной по (*M, w*) в теореме 9.9;* M′ *допускает произвольный вход тогда и только тогда, когда* M *допускает* w

M' по построению выполняет следующие действия.

1. M' игнорирует собственный вход. Она заменяет его цепочкой, представляющей МТ M и ее вход w. Поскольку M' строится для конкретной пары (M, w), имеющей неко-

торую длину n, можно построить M' с состояниями $q_0, q_1, \ldots, q_n$ (q_0 — начальное состояние), причем:

а) в состоянии q_i, $i = 0, 1, \ldots, n-1$, M' записывает $(i+1)$-й бит кода (M, w), переходит в состояние q_{i+1} и сдвигается вправо;

б) в состоянии q_n в случае необходимости M' сдвигается вправо и заполняет все непустые клетки (содержащие "хвост" цепочки x, если ее длина больше n) пробелами.

2. Когда M' достигает пробела в состоянии q_n, она, используя похожий набор состояний, перемещает головку в левый конец ленты.

3. M', используя дополнительные состояния, моделирует универсальную МТ U на полученной ленте.

4. Если U допускает, то и M' допускает. Если U никогда не допускает, то и M' никогда не допускает.

Приведенное описание M' показывает возможность построения машины Тьюринга, которая переводит код M и цепочки w в код M'. Таким образом, существует алгоритм сведения L_u к L_{ne}. Кроме того, если M допускает w, то M' допускает любой вход x, который первоначально содержался на ее ленте. То, что вначале вход x был проигнорирован, не имеет никакого значения, поскольку по определению МТ допускает то, что было на ее ленте до начала операций. Таким образом, если M допускает w, то код M' принадлежит L_{ne}.

Наоборот, если M не допускает w, то M' никогда не допустит свой вход, каким бы он ни был. Следовательно, в этом случае код M' не принадлежит L_{ne}. Таким образом, L_u сводится к L_{ne} с помощью алгоритма, который строит M' по данным M и w. Поскольку L_u не является рекурсивным, то и L_{ne} также не рекурсивен. Того, что это сведение существует, вполне достаточно для завершения доказательства. Однако для того, чтобы проиллюстрировать значимость сведения, продолжим наши рассуждения. Если бы L_{ne} был рекурсивным, то можно было бы построить алгоритм для L_u следующим образом.

1. Преобразовать (M, w) в МТ M' описанным выше способом.

2. Выяснить с помощью гипотетического алгоритма для L_{ne}, верно ли $L(M') = \varnothing$. Если это так, то сказать, что M не допускает w, если же $L(M') \neq \varnothing$, то сказать, что M допускает w.

Поскольку из теоремы 9.6 известно, что такого алгоритма для L_u нет, приходим к противоречию с тем, что L_{ne} является рекурсивным, и делаем вывод, что L_{ne} — не рекурсивный. □

Теперь известен и статус языка L_e. Если бы L_e был РП, то по теореме 9.4 и он, и L_{ne} были бы рекурсивными. Но поскольку L_{ne} не является рекурсивным по теореме 9.9, то справедливо следующее утверждение.

Теорема 9.10. *Я*зык L_e не является РП. □

9.3.3. Теорема Райса и свойства РП-языков

Неразрешимость языков, подобных L_e и L_{ne}, в действительности представляет собой частный случай гораздо более общей теоремы: любое нетривиальное свойство РП-языков неразрешимо в том смысле, что с помощью машин Тьюринга невозможно распознать цепочки, которые являются кодами МТ, обладающих этим свойством. Примером свойства РП-языков служит выражение "язык является контекстно-свободным". Вопрос о том, допускает ли данная МТ контекстно-свободный язык, является неразрешимым частным случаем общего закона, согласно которому все нетривиальные свойства РП-языков неразрешимы.

Свойство РП-языков представляет собой некоторое множество РП-языков. Таким образом, формально свойство языка быть контекстно-свободным — это множество всех КС-языков. Свойство быть пустым есть множество $\{\varnothing\}$, содержащее только пустой язык.

Почему проблемы и их дополнения различны

Интуиция подсказывает, что в действительности проблема и ее дополнение — одна и та же проблема. Для решения одной можно использовать алгоритм решения другой, и лишь на последнем шаге взять отрицание ответа: вместо "нет" сказать "да" и наоборот. Если проблема и ее дополнение рекурсивны, это действительно верно.

Однако, как обсуждалось в разделе 9.2.2, существуют две другие возможности. Во-первых, может быть, что ни сама проблема, ни ее дополнение не являются даже РП. Тогда ни одна из них вообще не может быть решена никакой МТ. В этом смысле они по-прежнему похожи друг на друга. Существует, однако, еще одна интересная ситуация типа L_e и L_{ne}, когда один из языков является РП, а другой — нет.

Для языка, являющегося РП, можно построить МТ, которая получает на вход цепочку w и исследует, почему данная цепочка принадлежит языку. Так, для L_{ne} и данной в качестве входа МТ M мы заставляем нашу МТ искать цепочки, которые допускает эта МТ, и, обнаружив хотя бы одну такую цепочку, допускаем M. Если M — МТ с пустым языком, то мы никогда не будем знать наверняка, что M не принадлежит L_{ne}, но при этом никогда и не допустим M, и это будет правильным откликом МТ.

С другой стороны, для дополнения этой проблемы — языка L_e, который не является РП, вообще не существует способа, позволяющего допустить все его цепочки. Предположим, что нам дана цепочка M, представляющая МТ с пустым языком. Проверяя различные входы M, мы можем никогда не найти такого, который M допускает, но при этом не сможем и быть уверенными в том, что такого входа нет среди еще непроверенных. Поэтому M никогда не сможет быть допущена, даже если и должна.

Свойство называется *тривиальным*, если оно либо пустое (т.е. никакой язык вообще ему не удовлетворяет), либо содержит все РП-языки. В противном случае свойство называется *нетривиальным*.

- Отметим, что пустое свойство $\varnothing$ и свойство быть пустым языком — не одно и то же.

Мы не можем распознать множество языков так, как сами эти языки. Причина в том, что обычный бесконечный язык нельзя записать в виде цепочки конечной длины, которая может быть входом некоторой МТ. Вместо этого мы должны распознавать машины Тьюринга, которые допускают эти языки. Код самой МТ конечен, даже если язык, который она допускает, бесконечен. Таким образом, если $\mathcal{P}$ — это свойство РП-языков, то $L_{\mathcal{P}}$ — множество кодов машин Тьюринга M_i, языки $L(M_i)$ которых принадлежат $\mathcal{P}$. Говоря о разрешимости свойства $\mathcal{P}$, мы имеем в виду разрешимость языка $L_{\mathcal{P}}$.

Теорема 9.11 (теорема Райса). Всякое нетривиальное свойство РП-языков неразрешимо.

Доказательство. Пусть $\mathcal{P}$ — нетривиальное свойство РП-языков. Вначале допустим, что пустой язык $\varnothing$ не принадлежит $\mathcal{P}$; противоположный случай рассмотрим позже. Поскольку $\mathcal{P}$ — нетривиально, должен существовать непустой язык L, принадлежащий $\mathcal{P}$. Пусть M_L обозначает МТ, допускающую L.

Сведем L_u к $L_{\mathcal{P}}$ и этим докажем, что язык $L_{\mathcal{P}}$ неразрешим, поскольку L_u неразрешим. Алгоритм сведения получает на вход пару (M, w) и выдает некоторую МТ M'. Структура M' представлена на рис. 9.10. Если M не допускает w, то $L(M')$ есть $\varnothing$, и $L(M') = L$, если M допускает w.

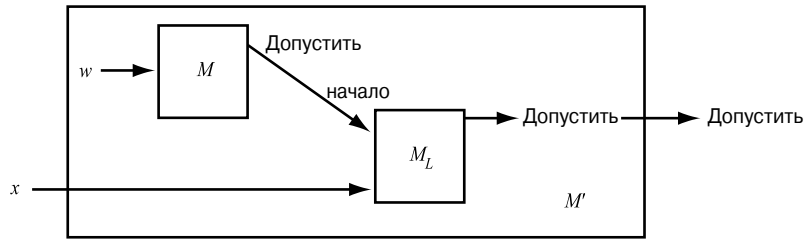


Рис. 9.10. Схема M′ для доказательства теоремы Райса

M' — двухленточная МТ. Одна лента используется для моделирования работы M со входом w. Напомним, что алгоритм сведения получает на вход пару (M, w) и может использовать ее для построения переходов M'. Таким образом, имитация M со входом w "встроена" в M', которой не нужно считывать с ленты переходы M.

Вторая лента M' используется для моделирования работы M_L с цепочкой x, входной для M'. Еще раз отметим, что переходы M_L известны алгоритму сведения и могут быть "встроены" в переходы M'. По построению МТ M' должна выполнять следующие операции.

1. Имитировать работу M со входом w. Отметим, что w не является входом M'. M' записывает M и w на одну из своих лент и имитирует МТ U на этой паре, как в доказательстве теоремы 9.8.

2. Если M не допускает w, то M' больше ничего не делает, т.е. не допускает любой свой вход x, поэтому $L(M') = \varnothing$. Мы предположили, что $\varnothing$ не содержится в свойстве $\mathcal{P}$, поэтому код M' не принадлежит $L_{\mathcal{P}}$.
3. Если M допускает w, то M' начинает имитацию M_L на своем собственном входе x. Таким образом, M' допускает именно язык L. Поскольку L принадлежит $\mathcal{P}$, то код M' принадлежит $L_{\mathcal{P}}$.

Как видим, построение M' по M и w можно провести в соответствии с некоторым алгоритмом. Поскольку этот алгоритм превращает (M, w) в M', которая принадлежит $L_{\mathcal{P}}$ тогда и только тогда, когда (M, w) принадлежит L_u, этот алгоритм представляет собой сведение L_u к $L_{\mathcal{P}}$ и доказывает неразрешимость свойства $\mathcal{P}$.

Для завершения доказательства нужно еще рассмотреть вариант, когда $\varnothing$ *принадлежит* $\mathcal{P}$. Для этого рассмотрим дополнение $\overline{\mathcal{P}}$ свойства $\mathcal{P}$, т.е. множество РП-языков, не обладающих свойством $\mathcal{P}$. Из описанных выше рассуждений следует, что свойство $\overline{\mathcal{P}}$ неразрешимо. Однако поскольку всякая МТ допускает некоторый РП-язык, то $\overline{L_{\mathcal{P}}}$, т.е. множество (кодов) машин Тьюринга, не допускающих языки из $\mathcal{P}$, совпадает с $L_{\overline{\mathcal{P}}}$ — множеством МТ, допускающих языки из $\overline{\mathcal{P}}$. Предположим, что язык $L_{\mathcal{P}}$ разрешим. Тогда $L_{\overline{\mathcal{P}}}$ также должен быть разрешимым, поскольку дополнение рекурсивного языка рекурсивно (теорема 9.3). □

9.3.4. Проблемы, связанные с описаниями языков в виде машин Тьюринга

Согласно теореме 9.11 все проблемы, связанные только с языками машин Тьюринга, неразрешимы. Некоторые из них интересны сами по себе. Например, неразрешимы следующие вопросы.

1. Пуст ли язык, допускаемый МТ (ответ дают теоремы 9.9 и 9.3)?
2. Конечен ли язык МТ?
3. Регулярен ли язык, допускаемый МТ?
4. Является ли язык МТ контекстно-свободным?

Вместе с тем, теорема Райса не означает, что любая проблема, связанная с МТ, неразрешима. Например, вопросы о состояниях МТ, в отличие от вопросов о языке, могут быть разрешимыми.

Пример 9.12. Вопрос о том, имеет ли МТ пять состояний, разрешим. Алгоритм, решающий его, просматривает код МТ и подсчитывает число состояний, встреченных в его переходах.

Еще один пример разрешимого вопроса — существует ли вход, при обработке которого МТ совершает более пяти переходов? Алгоритм решения становится очевидным,

если заметить, что, когда МТ делает пять переходов, она обозревает не более девяти клеток вокруг начальной позиции головки. Поэтому можно проимитировать пять переходов МТ на любом из конечного числа входов, длина которых не более девяти. Если все эти имитации не достигают останова, то делается вывод, что на любом входе данная МТ совершает более пяти переходов. □

9.3.5. Упражнения к разделу 9.3

9.3.1. (*) Покажите, что множество кодов машин Тьюринга, допускающих все входы, которые являются палиндромами (возможно, наряду с другими входами), неразрешимо.

9.3.2. Большая Компьютерная Корпорация для увеличения сократившегося объема продаж решила выпустить высокотехнологичную модель машины Тьюринга под названием МТЗС, оснащенную *звонками* и *свистками*. В основном МТЗС совпадает с обычной МТ, за исключением того, что каждое состояние этой машины отмечено либо как "состояние-звонок", либо как "состояние-свисток". Как только МТЗС попадает в новое состояние, то, в зависимости от его типа, она либо звенит, либо свистит. Докажите, что проблема определения, свистнет ли когда-нибудь данная МТЗС M при данном входе w, неразрешима.

9.3.3. Покажите, что язык кодов МТ, которые, начиная с пустой ленты, в конце концов записывают где-либо на ней символ 1, неразрешим.

9.3.4. (!) В соответствии с теоремой Райса известно, что каждый из следующих вопросов неразрешим:

а) содержит ли $L(M)$ хотя бы две цепочки?

б) бесконечен ли $L(M)$?

в) является ли язык $L(M)$ контекстно-свободным?

г) (*) верно ли, что $L(M) = (L(M))^R$?

Являются ли они рекурсивно перечислимыми или не-РП?

9.3.5. (!) Пусть язык L состоит из троек (M_1, M_2, k), образованных парой кодов МТ и целым числом, для которых $L(M_1) \cap L(M_2)$ содержит не менее, чем k цепочек. Покажите, что L является РП, но не рекурсивным.

9.3.6. Покажите, что следующие вопросы разрешимы:

а) (*) множество кодов МТ M, которые, имея в начальный момент пустую ленту, в конце концов записывают на ней некоторый непустой символ. *Указание*. Если M имеет m состояний, рассмотрите первые $m + 1$ совершаемых ею переходов;

б) (!) множество кодов МТ, которые никогда не совершают сдвиг влево;

в) (!) множество пар (M, w), для которых МТ M при обработке входа w не обозревает никакую клетку на ленте более одного раза.

9.3.7. (!) Покажите, что следующие проблемы не являются рекурсивно-перечислимыми:

а) (*) множество пар (M, w), для которых МТ M при обработке входа w не останавливается;

б) множество пар (M_1, M_2), для которых $L(M_1) \cap L(M_2) = \varnothing$;

в) множество троек (M_1, M_2, M_3), для которых $L(M_1) = L(M_2)\ L(M_3)$, т.е. язык первой машины — это конкатенация языков двух других МТ.

9.3.8. (!!) Ответьте, является ли каждое из следующих множеств рекурсивным, РП, но не рекурсивным, или не-РП:

а) (*) множество кодов всех МТ, останавливающихся на любом входе;

б) множество кодов всех МТ, которые не останавливаются ни на каком входе;

в) множество кодов всех МТ, останавливающихся хотя бы на одном входе;

г) (*) множество кодов всех МТ, которые не останавливаются хотя бы на одном входе.

9.4. Проблема соответствий Поста

В этом разделе начнется сведение неразрешимых вопросов о машинах Тьюринга к неразрешимым вопросам о “реальных вещах”, т.е. обычных предметах, не имеющих отношения к абстрактным машинам Тьюринга. Начнем с проблемы, которая называется “проблемой соответствий Поста” (ПСП). Эта проблема по-прежнему довольно абстрактна, но она связана уже не с машинами Тьюринга, а с цепочками. Наша цель — доказать неразрешимость этой проблемы, чтобы затем доказывать неразрешимость других проблем путем сведения ПСП к ним.

Докажем неразрешимость ПСП сведением L_u к ней. Чтобы облегчить доказательство, рассмотрим вначале “модифицированную” версию ПСП, а потом сведем ее к исходной ПСП. Затем сведем L_u к модифицированной ПСП. Цепь этих сведений представлена на рис. 9.11. Поскольку известно, что исходная проблема L_u неразрешима, можно сделать вывод, что ПСП также неразрешима.

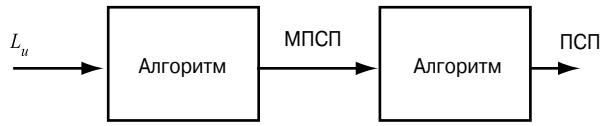


Рис. 9.11. Цепь сведений в доказательстве неразрешимости проблемы соответствий Поста

9.4.1. Определение проблемы соответствий Поста

Экземпляр *проблемы соответствий Поста* (ПСП) состоит из двух списков равной длины в некотором алфавите Σ. Как правило, мы будем называть их списками A и B, и писать $A = w_1, w_2, \ldots, w_k$ и $B = x_1, x_2, \ldots, x_k$ при некотором целом k. Для каждого i пара (w_i, x_i) называется парой *соответствующих цепочек*.

Мы говорим, что экземпляр ПСП *имеет решение*, если существует последовательность из одного или нескольких целых чисел $i_1, i_2, \ldots, i_m$, которая, если считать эти числа индексами цепочек и выбрать соответствующие цепочки из списков A и B, дает одну и ту же цепочку, т.е. $w_{i1} w_{i2} \ldots w_{im} = x_{i1} x_{i2} \ldots x_{im}$. В таком случае последовательность $i_1, i_2, \ldots, i_m$ называется *решающей последовательностью*, или просто *решением*, данного экземпляра ПСП. Проблема соответствий Поста состоит в следующем.

- Выяснить, имеет ли решение данный экземпляр ПСП.

Пример 9.13. Пусть $\Sigma = \{0, 1\}$, A и B — списки (рис. 9.12). Данный экземпляр ПСП имеет решение. Пусть, например, $m = 4$, $i_1 = 2$, $i_2 = 1$, $i_3 = 1$ и $i_4 = 3$, т.е. решающая последовательность имеет вид 2, 1, 1, 3. Для проверки записываются конкатенации соответствующих цепочек каждого из списков в порядке, задаваемом данной последовательностью, т.е. $w_2 w_1 w_1 w_3 = x_2 x_1 x_1 x_3 = 101111110$. Отметим, что это решение — не единственное. Например, 2, 1, 1, 3, 2, 1, 1, 3 также является решением. □

	Список A	Список B
i	w_i	x_i
1	1	111
2	10111	10
3	10	0

Рис. 9.12. Экземпляр ПСП

ПСП как язык

Речь идет о вопросе, существует ли решение у экземпляра ПСП, поэтому данную проблему нужно представить в виде языка. Поскольку экземпляры ПСП могут иметь произвольные алфавиты, язык ПСП на самом деле есть множество цепочек над некоторым фиксированным алфавитом, которыми кодируются экземпляры ПСП. Эти коды во многом напоминают коды машин Тьюринга с произвольными множествами состояний и ленточных символов, рассмотренные в разделе 9.1.2. Например, если экземпляр ПСП имеет алфавит, содержащий до 2^k символов, то для каждого из этих символов можно использовать k-битный двоичный код.

Поскольку каждый экземпляр ПСП имеет конечный алфавит, то некоторое k можно найти для каждого экземпляра. Следовательно, все экземпляры проблемы можно за-

кодировать с помощью алфавита из 3-х символов: 0, 1 и “запятой”, разделяющей цепочки. Код начинается двоичной записью числа k и запятой за ней. Затем записывается каждая из пар цепочек, в которых цепочки разделены запятыми, а символы закодированы k-битными двоичными числами.

Пример 9.14. Пусть, как и раньше, $\Sigma = \{0, 1\}$, но теперь экземпляр проблемы представлен списками, как на рис. 9.13.

	Список A	Список B
i	w_i	x_i
1	10	101
2	011	11
3	101	011

Рис. 9.13. Еще один экземпляр ПСП

В этом примере решения нет. Допустим, что экземпляр ПСП, представленный на рис. 9.13, имеет решение, скажем, $i_1, i_2, \ldots, i_m$ при некотором $m \geq 1$. Утверждаем, что $i_1 = 1$. При $i_1 = 2$ цепочка, начинающаяся с $w_2 = 011$, должна равняться цепочке, которая начинается с $x_2 = 11$. Но это равенство невозможно, поскольку их первые символы — 0 и 1, соответственно. Точно так же невозможно, чтобы $i_1 = 3$, поскольку тогда цепочка, начинающаяся с $w_3 = 101$, равнялась бы цепочке, которая начинается с $x_3 = 011$.

Если $i_1 = 1$, то две соответствующие цепочки из списков A и B должны начинаться так:

A: 10…
B: 101…

Рассмотрим теперь, каким может быть i_2.

1. Вариант $i_2 = 1$ невозможен, поскольку никакая цепочка, начинающаяся с $w_1w_1 = 1010$, не может соответствовать цепочке, которая начинается с $x_1x_1 = 101101$; эти цепочки различаются в четвертой позиции.
2. Вариант $i_2 = 2$ также невозможен, поскольку никакая цепочка, начинающаяся с $w_1w_2 = 10011$, не может соответствовать цепочке, которая начинается с $x_1x_2 = 10111$; они различаются в третьей позиции.
3. Возможен лишь вариант $i_2 = 3$.

При $i_2 = 3$ цепочки, соответствующие списку чисел 1, 3, имеют следующий вид.

A: 10101…
B: 101011…

Пока не видно, что последовательность 1, 3 невозможно продолжить до решения. Однако обосновать это несложно. Действительно, мы находимся в тех же условиях, в которых

были после выбора $i_1 = 1$. Цепочка из списка B отличается от цепочки из списка A лишним символом 1 на конце. Чтобы избежать несовпадения, мы вынуждены выбирать $i_3 = 3$, $i_4 = 3$ и так далее. Таким образом, цепочка из списка A никогда не догонит цепочку из списка B, и решение никогда не будет получено. □

9.4.2. „Модифицированная“ ПСП

Свести L_u к ПСП легче, если рассмотреть вначале промежуточную версию ПСП, которая называется *модифицированной проблемой соответствий Поста*, или МПСП. В модифицированной ПСП на решение накладывается дополнительное требование, чтобы первой парой в решении была пара первых элементов списков A и B. Более формально, экземпляр МПСП состоит из двух списков $A = w_1, w_2, \ldots, w_k$ и $B = x_1, x_2, \ldots, x_k$, и решением является последовательность из 0 или нескольких целых чисел $i_1, i_2, \ldots, i_m$, при которой

$$w_1 w_{i_1} w_{i_2} \ldots w_{i_m} = x_1 x_{i_1} x_{i_2} \ldots x_{i_m}.$$

Отметим, что цепочки обязательно начинаются парой (w_1, x_1), хотя индекс 1 даже не указан в качестве начального элемента решения. Кроме того, в отличие от ПСП, решение которой содержит хотя бы один элемент, решением МПСП может быть и пустая последовательность (когда $w_1 = x_1$). Однако такие экземпляры не представляют никакого интереса и далее не рассматриваются.

Частичные решения

В примере 9.14 для анализа экземпляров ПСП используется широко распространенный метод. Рассматривается, какими могут быть *частичные решения*, т.е. последовательности индексов $i_1, i_2, \ldots, i_r$, для которых одна из цепочек $w_{i_1} w_{i_2} \ldots w_{i_r}$ и $x_{i_1} x_{i_2} \ldots x_{i_r}$ является префиксом другой, хотя сами цепочки не равны. Заметим, что если последовательность целых чисел является решением, то всякий префикс этой последовательности должен быть частичным решением. Поэтому понимание того, как выглядят частичные решения проблемы, позволяет определить, какими могут быть полные решения.

Отметим, однако, что, поскольку ПСП неразрешима, не существует и алгоритма, позволяющего вычислять все частичные решения. Их может быть бесконечно много, а также, что гораздо хуже, если даже частичные решения $w_{i_1} w_{i_2} \ldots w_{i_r}$ и $x_{i_1} x_{i_2} \ldots x_{i_r}$ ведут к решению, разница их длин может быть сколь угодно большой.

Пример 9.15. Списки, представленные на рис. 9.12, можно рассматривать как экземпляр МПСП. Однако этот экземпляр МПСП не имеет решения. Для доказательства заметим, что всякое его частичное решение начинается с индекса 1, поэтому цепочки, образующие решение, должны начинаться следующим образом.

A: 1…
B: 111…

Следующим целым в решении не может быть 2 или 3, поскольку и w_2, и w_3 начинаются с 10, и поэтому при таком выборе возникнет несоответствие в третьей позиции. Таким образом, следующий индекс должен быть 1, и получаются такие цепочки.

A: 11…

B: 111111…

Эти рассуждения можно продолжать бесконечно. Только еще одно число 1 позволяет избежать несоответствия в решении. Но если все время выбирается индекс 1, то цепочка B остается втрое длиннее A, и цепочки никогда не сравняются. □

Важным шагом в доказательстве неразрешимости ПСП является сведение МПСП к ПСП. Далее будет показано, что МПСП неразрешима, путем сведения L_u к МПСП. Тем самым будет доказана неразрешимость ПСП. Действительно, если бы она была разрешима, то можно было бы решить МПСП, а следовательно, и L_u.

По данному экземпляру МПСП с алфавитом Σ соответствующий экземпляр ПСП строится следующим образом. Вначале вводится новый символ $*$, который помещается между символами цепочек экземпляра МПСП. При этом в цепочках списка A он следует за символами алфавита Σ, а в цепочках списка B — предшествует им. Исключением является новая пара, которая строится по первой паре экземпляра МПСП; в этой паре есть дополнительный символ $*$ в начале w_1, поэтому она может служить началом решения ПСП. К экземпляру ПСП добавляется также заключительная пара ($, *$). Она служит последней парой в решении ПСП, имитирующем решение экземпляра МПСП.

Формализуем описанную конструкцию. Пусть дан экземпляр МПСП со списками $A = w_1, w_2, \ldots, w_k$ и $B = x_1, x_2, \ldots, x_k$. Предполагается, что символы $*$ и $ отсутствуют в алфавите Σ данного экземпляра МПСП. Строится экземпляр ПСП со списками $C = y_0, y_1, \ldots, y_{k+1}$ и $D = z_0, z_1, \ldots, z_{k+1}$ следующим образом.

1. Для $i = 1, 2, \ldots, k$ положим y_i равной w_i с символом $*$ после каждого ее символа, а z_i — равной x_i с символом $*$ перед каждым ее символом.
2. $y_0 = *y_1$ и $z_0 = z_1$, т.е. нулевая пара выглядит так же, как первая, с той лишь разницей, что в начале цепочки из первого списка есть еще символ $*$. Отметим, что нулевая пара будет единственной парой экземпляра ПСП, в которой обе цепочки начинаются одним и тем же символом, поэтому всякое решение данного экземпляра ПСП должно начинаться с индекса 0.
3. $y_{k+1} = \$$ и $z_{k+1} = *\$$.

Пример 9.16. Пусть рис. 9.12 изображает экземпляр МПСП. Тогда экземпляр ПСП, построенный с помощью описанных выше действий, представлен на рис. 9.14. □

Теорема 9.17. МПСП сводится к ПСП.

Доказательство. В основе доказательства лежит описанная выше конструкция. Допустим, что $i_1, i_2, \ldots, i_m$ — решение данного экземпляра МПСП со списками A и B; тогда $w_1 w_{i_1} w_{i_2} \ldots w_{i_m} = x_1 x_{i_1} x_{i_2} \ldots x_{i_m}$. Если заменить каждую цепочку w соответствующей y и каж-

дую x — соответствующей z, то получатся две почти одинаковые цепочки $y_1y_{i1}y_{i2}\ldots y_{im}$ и $z_1z_{i1}z_{i2}\ldots z_{im}$. Разница между ними в том, что у первой не хватает * в начале, а у второй — в конце, т.е.

$$*y_1y_{i1}y_{i2}\ldots y_{im} = z_1z_{i1}z_{i2}\ldots z_{im}*.$$

	Список C	Список D
i	y_i	z_i
0	∗1∗	∗1∗1∗1
1	1∗	∗1∗1∗1
2	1∗0∗1∗1∗1∗	∗1∗0
3	1∗0∗	∗0
4	$	∗$

Рис. 9.14. Построение экземпляра ПСП по экземпляру МПСП

Однако $y_0 = *y_1$ и $z_0 = z_1$, поэтому можно “расправиться” с * в начале, изменив первый индекс на 0:

$$y_0y_{i1}y_{i2}\ldots y_{im} = z_0z_{i1}z_{i2}\ldots z_{im}*.$$

Для учета * в конце добавим индекс $k + 1$. Поскольку $y_{k+1} = \$$, а $z_{k+1} = *\$$, получим:

$$y_0y_{i1}y_{i2}\ldots y_{im}y_{k+1} = z_0z_{i1}z_{i2}\ldots z_{im}z_{k+1}.$$

Итак, показано, что последовательность 0, i_1, i_2, …, i_m, $k + 1$ — решение экземпляра ПСП.

Теперь докажем обратное, т.е. если построенный экземпляр ПСП имеет решение, то исходный экземпляр МПСП также имеет решение. Заметим, что решение данного экземпляра ПСП должно начинаться индексом 0 и заканчиваться индексом $k + 1$, поскольку только в нулевой паре цепочки y_0 и z_0 начинаются и только в $(k + 1)$-й паре цепочки y_{k+1} и z_{k+1} оканчиваются одним и тем же символом. Таким образом, решение экземпляра ПСП можно записать в виде 0, i_1, i_2, …, i_m, $k + 1$.

Мы утверждаем, что i_1, i_2, …, i_m есть решение экземпляра МПСП. Действительно, из цепочки $y_0y_{i1}y_{i2}\ldots y_{im}y_{k+1}$ можно удалить все символы * и символ $ в конце. В результате получим цепочку $w_1w_{i1}w_{i2}\ldots w_{im}$. Точно так же, удалив символы * и символ $ из цепочки $z_0z_{i1}z_{i2}\ldots z_{im}z_{k+1}$, получим $x_1x_{i1}x_{i2}\ldots x_{im}$. Поскольку

$$y_0y_{i1}y_{i2}\ldots y_{im}y_{k+1} = z_0z_{i1}z_{i2}\ldots z_{im}z_{k+1},$$

получаем:

$$w_1w_{i1}w_{i2}\ldots w_{im} = x_1x_{i1}x_{i2}\ldots x_{im}.$$

Итак, решение экземпляра ПСП содержит в себе решение экземпляра МПСП.

Теперь ясно, что конструкция, описанная перед данной теоремой, представляет собой алгоритм, который переводит экземпляр МПСП, имеющий решение, в экземпляр ПСП,

также имеющий решение, а экземпляр МПСП, не имеющий решения, — в экземпляр ПСП, не имеющий решения. Таким образом, МПСП сводится к ПСП, а это означает, что из разрешимости ПСП следовала бы разрешимость МПСП. □

9.4.3. Завершение доказательства неразрешимости ПСП

Завершим цепь сведений (см. рис. 9.11), сведя L_u к МПСП. По заданной паре (M, w) строится экземпляр МПСП (A, B), имеющий решение тогда и только тогда, когда МТ M допускает вход w.

Основная идея состоит в том, что частичные решения экземпляра МПСП (A, B) имитируют обработку входа w машиной M. Частичные решения будут состоять из цепочек, которые являются префиксами последовательности МО $M\ \#\alpha_1\#\alpha_2\#\alpha_3\#\ldots$, где α_1 — начальное МО M при входной цепочке w и $\alpha_i \vdash \alpha_{i+1}$ для всех i. Цепочка из списка B всегда на одно МО впереди цепочки из списка A, за исключением ситуации, когда M попадает в допускающее состояние. В этом случае используются специальные пары, позволяющие списку A "догнать" список B и в конце концов выработать решение. Однако если машина не попадает в допускающее состояние, то эти пары не могут быть использованы, и проблема не имеет решения.

Для того чтобы упростить построение экземпляра МПСП, воспользуемся теоремой 8.12, согласно которой можно считать, что МТ никогда не печатает пробел и ее головка не сдвигается левее исходного положения. Тогда МО машины Тьюринга всегда будет цепочкой вида $\alpha q\beta$, где α и β — цепочки непустых символов на ленте, а q — состояние. Однако тогда, когда головка обозревает пробел непосредственно справа от α, позволим цепочке β быть пустой вместо того, чтобы помещать пробел справа от состояния. Таким образом, символы цепочек α и β будут в точности соответствовать содержимому клеток, в которых записан вход, а также всех тех клеток справа, в которых головка уже побывала.

Пусть $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ — машина Тьюринга, удовлетворяющая условиям теоремы 8.12, и w — входная цепочка из Σ^*. По ним строится экземпляр МПСП следующим образом. Чтобы понять, почему пары выбираются пары именно так, а не иначе, нужно помнить, что нам нужно, чтобы первый список всегда на одно МО отставал от второго, если только M не попадает в допускающее состояние.

1. Первая пара имеет следующий вид.

Список A	Список B
#	$\#q_0w\#$

В соответствии с правилами МПСП эта пара — первая в любом решении. С нее начинается имитация M на входе w. Заметим, что в начальный момент список B опережает список A на одно полное МО.

2. Ленточные символы и разделитель # могут быть добавлены в оба списка. Пары

Список A	Список B	
X	X	для каждого X из Γ
#	#	

позволяют "копировать" символы, не обозначающие состояния. Выбирая такие пары, можно одновременно и удлинить цепочку списка A до соответствующей цепочки списка B, и скопировать часть предыдущего МО в конец цепочки списка B. Это поможет нам записать в конец цепочки списка B следующее МО в последовательности переходов M.

3. Для имитации перехода M применяются специальные пары. Для всех q из $Q - F$ (т.е. состояние q не является допускающим), p из Q и X, Y, Z из Γ есть следующие пары.

Список A	Список B	
qX	Yp	если $\delta(q, X) = (p, Y, R)$
ZqX	pZY	если $\delta(q, X) = (p, Y, L)$; Z — любой ленточный символ
$q\#$	$Yp\#$	если $\delta(q, B) = (p, Y, R)$
$Zq\#$	$pZY\#$	если $\delta(q, B) = (p, Y, L)$; Z — любой ленточный символ

Как и в пункте 2, эти пары позволяют добавить к цепочке B следующее МО, дописывая цепочку A таким образом, чтобы она соответствовала цепочке B. Однако эти пары используют состояние для определения, как нужно изменить текущее МО, чтобы получить следующее. Эти изменения — новое состояние, ленточный символ и сдвиг головки — отображаются в МО, которое строится в конце цепочки B.

4. Если МО, которое находится в конце цепочки B, содержит допускающее состояние, нужно сделать частичное решение полным. Для этого добавляются "МО", которые в действительности не являются МО машины M, а отображают ситуацию, при которой в допускающем состоянии разрешается поглощать все ленточные символы по обе стороны от головки. Таким образом, если q — допускающее состояние, то для всех ленточных символов X и Y существуют следующие пары.

Список A	Список B
XqY	q
Xq	q
qY	q

5. Наконец, если допускающее состояние поглотило все ленточные символы, то оно остается в одиночестве как последнее МО в цепочке B. Таким образом, *разность* цепочек (суффикс цепочки B, который нужно дописать к цепочке A для того, чтобы она соответствовала B) есть $q\#$, и для завершения решения используется последняя пара.

Список A	Список B
$q\#\#$	$\#$

В дальнейшем пары этих пяти типов называются парами из правила 1, из правила 2 и так далее.

Пример 9.18. Преобразуем в экземпляр МПСП машину

$$M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_3\})$$

с таблицей δ

q_i	$\delta(q_i, 0)$	$\delta(q_i, 1)$	$\delta(q_i, B)$
q_1	$(q_2, 1, R)$	$(q_2, 0, L)$	$(q_2, 1, L)$
q_2	$(q_3, 0, L)$	$(q_1, 0, R)$	$(q_2, 0, R)$
q_3	—	—	—

и входной цепочкой $w = 01$.

Чтобы упростить построение, заметим, что M никогда не печатает пробел B, и его не будет ни в одном МО. Поэтому все пары с пустым символом B опускаются. Полный список пар с указанием их происхождения представлен на рис. 9.15.

Отметим, что M допускает входную цепочку 01 в результате следующей последовательности переходов.

$$q_101 \vdash 1q_21 \vdash 10q_1 \vdash 1q_201 \vdash q_3101$$

Рассмотрим последовательность частичных решений, которая имитирует эти вычисления M и в конце концов приводит к решению. Начать нужно с первой пары, как того требует всякое решение МПСП.

A: $\#$

B: $\#q_101\#$

Для того чтобы частичное решение можно было продолжить, следующая цепочка из списка A должна быть префиксом разности $q_101\#$. Поэтому следующей парой должна быть $(q_10, 1q_2)$. Это одна из пар, имитирующих переходы, полученная по правилу 3. Итак, получаем следующее частичное решение.

A: $\#q_10$

B: $\#q_101\#1q_2$

Теперь можно продолжить частичное решение, используя “копирующие” пары из правила 2 до тех пор, пока не дойдем до состояния во втором МО. Тогда частичное решение примет такой вид.

A: $\#q_101\#1$

B: $\#q_101\#1q_21\#1$

Для имитации перехода тут снова можно использовать пару из правила 3. Подходящей является пара ($q_2$1, 0q_1), и в результате получается следующее частичное решение.

A: #$q_1$01#1$q_2$1

B: #$q_1$01#1$q_2$1#10q_1

Правило	Список *A*	Список *B*	Источник
(1)	#	#$q_1$01#	
(2)	0 1 #	0 1 #	
(3)	$q_1$0 0$q_1$1# 1$q_1$1# 0q_1# 1q_1# 0$q_2$0 1$q_2$0 $q_2$1 q_2#	1q_2 $q_2$00 $q_2$10 $q_2$01# $q_2$11# $q_3$00# $q_3$10# 0q_1 0q_2#	$\delta(q_1, 0) = (q_2, 1, R)$ $\delta(q_1, 1) = (q_2, 0, L)$ $\delta(q_1, 1) = (q_2, 0, L)$ $\delta(q_1, B) = (q_2, 1, L)$ $\delta(q_1, B) = (q_2, 1, L)$ $\delta(q_2, 0) = (q_3, 0, L)$ $\delta(q_2, 0) = (q_3, 0, L)$ $\delta(q_2, 1) = (q_1, 0, R)$ $\delta(q_2, B) = (q_2, 0, R)$
(4)	0$q_3$0 0$q_3$1 1$q_3$0 1$q_3$1 0q_3 1q_3 $q_3$0 $q_3$1	q_3 q_3 q_3 q_3 q_3 q_3 q_3 q_3	
(5)	q_3##	#	

Рис. 9.15. Экземпляр МПСП, построенный по МТ M *и слову* w *из примера 9.18*

Теперь можно было бы использовать пары из правила 3 и скопировать следующие три символа #, 1 и 0. Однако это решение было бы ошибочным, поскольку следующий переход *M* сдвигает головку влево, и символ 0, стоящий непосредственно перед состоянием, потребуется в следующей паре из правила 3. Поэтому копируются лишь два следующих символа.

A: #$q_1$01#1$q_2$1#1

B: #$q_1$01#1$q_2$1#10q_1#1

Подходящей парой из правила 3 является ($0q_1$#, $q_2$01#), и получается новое частичное решение.

A: #$q_1$01#1$q_2$1#10q_1#
B: #$q_1$01#1$q_2$1#10q_1#1$q_2$01#

Используя теперь еще одну пару ($1q_2$0, $q_3$10) из правила 3, приходим к допусканию.

A: #$q_1$01#1$q_2$1#10q_1#1$q_2$0
B: #$q_1$01#1$q_2$1#10q_1#1$q_2$01#$q_3$10

Теперь, с помощью пар из правила 4 в МО исключаются все символы, кроме q_3. Для правильного копирования символов нужны также пары из правила 2. Частичное решение продолжается до следующего.

A: #$q_1$01#1$q_2$1#10q_1#1$q_2$01#$q_3$101#$q_3$01#$q_3$1#
B: #$q_1$01#1$q_2$1#10q_1#1$q_2$01#$q_3$101#$q_3$01#$q_3$1#q_3#

Теперь в МО находится лишь q_3. Чтобы завершить решение, можно использовать пару (q_3##, #) из правила 5.

A: #$q_1$01#1$q_2$1#10q_1#1$q_2$01#$q_3$101#$q_3$01#$q_3$1#q_3##
B: #$q_1$01#1$q_2$1#10q_1#1$q_2$01#$q_3$101#$q_3$01#$q_3$1#q_3##

□

Теорема 9.19. Проблема соответствий Поста неразрешима.

Доказательство. Цепь сведений, представленная на рис. 9.11, почти завершена. Сведение МПСП к ПСП было показано в теореме 9.17. В данном разделе приведена конструкция, позволяющая свести L_u к МПСП. Таким образом, для завершения доказательства неразрешимости ПСП покажем, что эта конструкция корректна, т.е.

- *M* допускает *w* тогда и только тогда, когда построенный экземпляр МПСП имеет решение.

(*Необходимость*) Основную идею подсказывает пример 9.18. Если *w* принадлежит *L*(*M*), то можно проимитировать работу *M* со входом *w*, начав с пары из правила 1. Для копирования состояния каждого МО и имитации одного перехода *M* используются пары из правила 3, а для копирования ленточных символов и при необходимости маркера # — из 2. Если *M* попадает в допускающее состояние, то с помощью пар из правила 4 и заключительной пары из 5 мы позволяем цепочке *A* догнать цепочку *B*, тем самым формируя решение.

(*Достаточность*) Нужно доказать, что если данный экземпляр МПСП имеет решение, то только потому, что *M* допускает *w*. Отметим, что, поскольку мы имеем дело с МПСП, любое решение должно начинаться с первой пары, так что начало частичного решения имеет следующий вид.

A: #
B: #q_0w#

Поскольку в этом частичном решении нет допускающего состояния, то пары из правил 4 и 5 бесполезны. Состояния и окружающие их один или два символа могут быть обработаны только с помощью пар из правила 3, а все остальные ленточные символы и # должны быть обработаны с помощью пар из правила 2. Поэтому, если M не попала в допускающее состояние, все частичные решения имеют общий вид

A: x

B: xy,

где x — последовательность МО машины M, представляющих вычисления M со входом w, возможно, с символом # и началом α следующего МО в конце. Разность y содержит завершение α, еще один символ # и начало МО, следующего за α, вплоть до точки, на которой заканчивается α в x.

Отметим, что до тех пор, пока M не попадает в допускающее состояние, частичное решение не является решением, поскольку цепочка B длиннее цепочки A. Поэтому, если решение существует, то M должна когда-нибудь попасть в допускающее состояние, т.е. допустить w. □

9.4.4. Упражнения к разделу 9.4

9.4.1. Выясните, имеют ли решение следующие экземпляры ПСП. Каждый из них представлен двумя списками — A и B, и i-е цепочки списков соответствуют друг другу (i = 1, 2, ...).

а) (*) A = (01, 001, 10); B = (011, 10, 00).

б) A = (01, 001, 10); B = (011, 01, 00).

в) A = (ab, a, bc, c); B = (bc, ab, ca, a).

9.4.2. (!) Было доказано, что ПСП неразрешима в предположении, что алфавит Σ может быть произвольным. Покажите, что ПСП неразрешима даже при ограничении Σ = {0, 1}, сведя ПСП к данному частному случаю.

9.4.3. (*!) Допустим, ПСП ограничена односимвольным алфавитом Σ = {0}. Будет ли ПСП и в этом ограниченном случае по-прежнему неразрешимой?

9.4.4. (!) *ТАГ-система Поста* образуется множеством пар цепочек в некотором конечном алфавите Σ и стартовой цепочкой. Если (w, x) — пара и y — произвольная цепочка в Σ, то мы говорим, что $wy \vdash yx$. Таким образом, на одном шаге можно стереть некоторый префикс w "текущей" цепочки wy и вместо него в конце приписать второй компонент пары с w — цепочку x. Определим $\overset{*}{\vdash}$ как нуль или несколько шагов $\vdash$, точно так же, как для порождений в КС-грамматиках. Покажите, что проблема выяснения, верно ли $z \overset{*}{\vdash} \varepsilon$ для данного множества пар P и стартовой цепочки z, неразрешима. *Указание*. Пусть для любых

МТ M и цепочки w стартовая цепочка z является начальным МО со входом w и разделителем # на конце. Выберите пары P так, что всякое МО M должно в конце концов превращаться в МО, в которое M попадает за один переход. Устройте так, что если M попадает в допускающее состояние, то текущая цепочка может быть в конце концов стерта, т.е. сведена к ε.

9.5. Другие неразрешимые проблемы

Здесь рассматривается ряд других неразрешимых проблем. Основным методом доказательства их неразрешимости является свед*е*ние к ним ПСП.

9.5.1. Проблемы, связанные с программами

Прежде всего, отметим, что на любом привычном языке можно написать программу, которая на вход получает экземпляр ПСП и ищет решение по определенной системе, например, в порядке возрастания *длины* (числа пар) потенциальных решений. Поскольку ПСП может иметь произвольный алфавит, нужно закодировать символы ее алфавита с помощью двоичного или другого фиксированного алфавита, о чем говорилось во врезке "ПСП как язык" в разделе 9.4.1.

Наша программа может выполнять любое конкретное действие, например, останавливаться или печатать фразу `hello, world`, если она нашла решение. В противном случае программа никогда не выполняет это конкретное действие. Таким образом, вопрос о том, печатает ли программа `hello, world`, останавливается ли, вызывает ли определенную функцию, заставляет ли консоль звенеть, или выполняет любое другое нетривиальное действие, неразрешим. В действительности для программ справедлив аналог теоремы Райса: всякое нетривиальное свойство программы, связанное с ее действиями (но не лексическое или синтаксическое свойство самой программы), является неразрешимым.

9.5.2. Неразрешимость проблемы неоднозначности КС-грамматик

Программы и машины Тьюринга — это в общем-то одно и то же, так что в замечаниях из раздела 9.5.1 нет ничего удивительного. Теперь мы увидим, как ПСП можно свести к проблеме, которая, на первый взгляд, совсем не похожа на вопрос о компьютерах. Это проблема выяснения, является ли данная КС-грамматика неоднозначной.

Основная идея заключается в рассмотрении цепочек, представляющих обращение последовательности индексов, вместе с соответствующими им цепочками одного из списков экземпляра ПСП. Такие цепочки могут порождаться некоторой грамматикой. Аналогичное множество цепочек для второго списка экземпляра ПСП также может порождаться некоторой грамматикой. Если объединить эти грамматики очевидным способом, то найдется цепочка, которая порождается правилами каждой исходной грамматики тогда и только тогда,

когда данный экземпляр ПСП имеет решение. Таким образом, решение существует тогда и только тогда, когда в грамматике для объединения присутствует неоднозначность.

Уточним эту идею. Пусть экземпляр ПСП состоит из списков $A = w_1, w_2, \ldots, w_k$ и $B = x_1, x_2, \ldots, x_k$. Для списка A построим КС-грамматику с одной переменной A. Терминалами будут все символы алфавита Σ, используемые в данном экземпляре ПСП, плюс не пересекающееся с Σ множество *индексных символов* $a_1, a_2, \ldots, a_k$, которые представляют выборы пар цепочек в решении ПСП. Таким образом, индексный символ a_i представляет выбор w_i из списка A или x_i из списка B. Продукции КС-грамматики для списка A имеют следующий вид.

$$\begin{aligned} A \quad \rightarrow \quad & w_1Aa_1 \mid w_2Aa_2 \mid \ldots \mid w_kAa_k \mid \\ & w_1a_1 \mid w_2a_2 \mid \ldots \mid w_ka_k \end{aligned}$$

Обозначим эту грамматику через G_A, а ее язык — через L_A. Язык типа L_A будет называться в дальнейшем *языком для списка A*.

Заметим, что все терминальные цепочки, порожденные переменной A в G_A, имеют вид $w_{i1}w_{i2}\cdots w_{im}a_{im}\cdots a_{i2}a_{i1}$ при некотором $m \geq 1$, где $i_1, i_2, \ldots, i_m$ — последовательность целых чисел в пределах от 1 до k. Все выводимые цепочки содержат одиночный символ A, отделяющий цепочки w от индексных символов a, до тех пор, пока не используется одно из k правил последней группы, в которых A отсутствует. Поэтому деревья разбора имеют структуру, представленную на рис. 9.16.

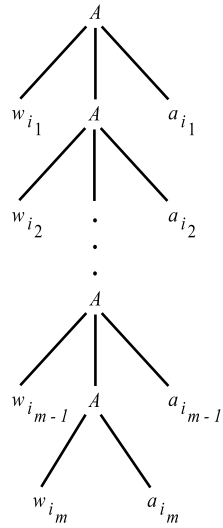


Рис. 9.16. Общий вид деревьев разбора в грамматике G_A

Заметим также, что любая терминальная цепочка порождается в G_A одним-единственным способом. Индексные символы в конце цепочки однозначно определяют, какое именно правило должно использоваться на каждом шаге, поскольку тела лишь двух

продукций оканчиваются данным индексным символом a_i: $A \to w_i A a_i$ и $A \to w_i a_i$. Первую продукцию нужно использовать, когда данный шаг порождения не последний, в противном же случае используется вторая продукция.

Рассмотрим теперь вторую часть экземпляра ПСП — список $B = x_1, x_2, \ldots, x_k$. Этому списку мы поставим в соответствие еще одну грамматику G_B с такими продукциями.

$$\begin{aligned} B \quad \to \quad & x_1 B a_1 \mid x_2 B a_2 \mid \ldots \mid x_k B a_k \mid \\ & x_1 a_1 \mid x_2 a_2 \mid \ldots \mid x_k a_k \end{aligned}$$

Язык этой грамматики обозначим через L_B. Все замечания, касающиеся G_A, справедливы и для G_B. В частности, терминальная цепочка из L_B имеет одно-единственное порождение, определяемое индексными символами в конце этой цепочки.

Наконец, мы объединяем языки и грамматики двух списков, формируя грамматику G_{AB} для всего данного экземпляра ПСП. G_{AB} имеет следующие компоненты.

1. Переменные A, B и S, последняя из которых — стартовый символ.
2. Продукции $S \to A \mid B$.
3. Все продукции грамматики G_A.
4. Все продукции грамматики G_B.

Утверждаем, что грамматика G_{AB} неоднозначна тогда и только тогда, когда экземпляр (A, B) ПСП имеет решение. Это утверждение является основным в следующей теореме.

Теорема 9.20. Вопрос о неоднозначности КС-грамматики неразрешим.

Основная часть сведения ПСП к вопросу о неразрешимости КС-грамматики проведена выше. В силу неразрешимости ПСП это сведение доказывает неразрешимость проблемы неоднозначности КС-грамматики. Остается лишь показать корректность приведенной конструкции, т.е. что

- G_{AB} неоднозначна тогда и только тогда, когда экземпляр (A, B) ПСП имеет решение.

(*Достаточность*) Предположим, что $i_1, i_2, \ldots, i_m$ — решение данного экземпляра ПСП. Рассмотрим два порождения в G_{AB}.

$$S \Rightarrow A \Rightarrow w_{i1} A a_{i1} \Rightarrow w_{i1} w_{i2} A a_{i2} a_{i1} \Rightarrow \cdots \Rightarrow$$
$$w_{i1} w_{i2} \cdots w_{im-1} A a_{im-1} \cdots a_{i2} a_{i1} \Rightarrow w_{i1} w_{i2} \cdots w_{im} a_{im} \cdots a_{i2} a_{i1}$$
$$S \Rightarrow B \Rightarrow x_{i1} B a_{i1} \Rightarrow x_{i1} x_{i2} B a_{i2} a_{i1} \Rightarrow \cdots \Rightarrow$$
$$x_{i1} x_{i2} \cdots x_{im-1} B a_{im-1} \cdots a_{i2} a_{i1} \Rightarrow x_{i1} x_{i2} \cdots x_{im} a_{im} \cdots a_{i2} a_{i1}$$

Поскольку $i_1, i_2, \ldots, i_m$ — решение, то $w_{i1} w_{i2} \cdots w_{im} = x_{i1} x_{i2} \cdots x_{im}$, т.е. эти два порождения представляют собой порождения одной и той же цепочки. А поскольку сами порожде-

ния, очевидно, являются двумя различными левыми порождениями одной и той же терминальной цепочки, делаем вывод, что грамматика G_{AB} неоднозначна.

(*Необходимость*) Как было отмечено ранее, для данной терминальной цепочки существует не более одного порождения в G_A и не более одного порождения в G_B. Поэтому терминальная цепочка может иметь в G_{AB} два левых порождения лишь в том случае, если одно из них начинается с $S \Rightarrow A$ и продолжается порождением в G_A, а второе начинается с $S \Rightarrow B$ и продолжается порождением в G_B.

Цепочка с двумя порождениями имеет окончание из индексов $a_{i_m} \cdots a_{i_2} a_{i_1}$ при некотором $m \geq 1$. Это окончание и должно быть решением данного экземпляра ПСП, поскольку то, что ей предшествует в цепочке с двумя порождениями, есть одновременно и $w_{i_1} w_{i_2} \cdots w_{i_m}$, и $x_{i_1} x_{i_2} \cdots x_{i_m}$. □

9.5.3. Дополнение языка списка

Наличие контекстно-свободных языков, подобных L_A для списка A, позволяет показать неразрешимость ряда проблем, связанных с КС-языками. Еще больше фактов о неразрешимости свойств КС-языков можно получить, рассмотрев дополнение этого языка $\overline{L_A}$. Отметим, что язык $\overline{L_A}$ состоит из всех цепочек в алфавите $\Sigma \cup \{a_1, a_2, \ldots, a_k\}$, не содержащихся в L_A, где Σ — алфавит некоторого экземпляра ПСП, а a_i — символы, которые не принадлежат Σ и представляют индексы пар в этом экземпляре ПСП.

Интересными элементами языка $\overline{L_A}$ являются цепочки, префикс которых принадлежит Σ^*, т.е. является конкатенацией цепочек из списка A, а суффикс состоит из индексных символов, *не соответствующих* цепочкам из префикса. Однако помимо этих элементов в $\overline{L_A}$ содержатся цепочки неправильного вида, не принадлежащие языку регулярного выражения $\Sigma^*(a_1 + a_2 + \ldots + a_k)^*$.

Мы утверждаем, что $\overline{L_A}$ является КС-языком. В отличие от L_A, грамматику для $\overline{L_A}$ построить нелегко, но можно построить МП-автомат, точнее — детерминированный МП-автомат. Конструкция описывается в следующей теореме.

Теорема 9.21. Если L_A — язык для списка A, то $\overline{L_A}$ является контекстно-свободным языком.

Доказательство. Пусть Σ — алфавит цепочек из списка $A = \{w_1, w_2, \ldots, w_k\}$, а $I = \{a_1, a_2, \ldots, a_k\}$ — множество индексных символов. ДМПА P, который по построению должен допускать $\overline{L_A}$, работает следующим образом.

1. Обозревая символ из Σ, P записывает его в магазин. Поскольку все цепочки из Σ^* принадлежат $\overline{L_A}$, P допускает их по мере чтения.
2. При виде индексного символа из I, скажем, a_i, P просматривает магазин, чтобы проверить, образуют ли символы из его верхней части цепочку w_i^R, т.е. обращение соответствующей цепочки:

а) если это не так, то входная цепочка, а также всякое ее продолжение принадлежат $\overline{L_A}$. Поэтому P переходит в допускающее состояние, в котором он дочитывает вход, не изменяя содержимое магазина;

б) если цепочка w_i^R была извлечена из верхней части магазина, но маркер дна магазина не показался на вершине, то P допускает, но запоминает в своем состоянии, что он ищет только символы из I и может еще встретить цепочку из L_A (которую *не допустит*). P повторяет шаг 2 до тех пор, пока вопрос о принадлежности входа языку L_A будет оставаться неразрешенным;

в) если w_i^R была извлечена и показался маркер дна магазина, то P прочитал вход, принадлежащий L_A. P не допускает этот вход. Однако, поскольку всякое продолжение этого входа уже не может содержаться в L_A, P переходит в такое состояние, в котором он допускает все последующие входы, не изменяя магазин.

3. Если после того, как P считал один или несколько символов из I, он встречает еще один символ из Σ, то входная цепочка не может принадлежать L_A, поскольку имеет неправильную форму. Поэтому P переходит в состояние, в котором он допускает этот и все последующие входы, не изменяя магазин.

□

Для получения результатов о неразрешимости КС-языков можно использовать L_A, L_B и их дополнения различными способами. Часть этих фактов собрана в следующей теореме.

Теорема 9.22. Пусть G_1 и G_2 — КС-грамматики, а R — регулярное выражение. Тогда неразрешимы следующие проблемы:

а) $L(G_1) \cap L(G_2) = \varnothing$?

б) $L(G_1) = L(G_2)$?

в) $L(G_1) = L(R)$?

г) верно ли, что $L(G_1) = T^*$ для некоторого алфавита T?

д) $L(G_1) \subseteq L(G_2)$?

е) $L(R) \subseteq L(G_1)$?

Доказательство. Каждое из доказательств представляет собой сведение ПСП. Покажем, как преобразовать экземпляр (A, B) ПСП в вопрос о КС-грамматиках и/или регулярных выражениях, который имеет ответ “да” тогда и только тогда, когда данный экземпляр ПСП имеет решение. В некоторых случаях ПСП сводится к самому вопросу, сформулированному в теореме, в других же случаях — к его дополнению. Это не имеет значения, поскольку, если показано, что дополнение проблемы неразрешимо, то сама

проблема также не может быть разрешимой, так как рекурсивные языки замкнуты относительно дополнения (теорема 9.3).

Алфавит цепочек данного экземпляра ПСП обозначим Σ, а алфавит индексных символов — I. В наших сведениях будем опираться на то, что L_A, L_B, $\overline{L_A}$ и $\overline{L_B}$ имеют КС-грамматики. КС-грамматики либо строятся непосредственно, как в разделе 9.5.2, либо вначале строятся МП-автоматы для языков-дополнений (см. теорему 9.21), а затем с помощью теоремы 6.14 они преобразуются в КС-грамматики.

1. Пусть $L(G_1) = L_A$ и $L(G_2) = L_B$. Тогда $L(G_1) \cap L(G_2)$ — множество решений данного экземпляра ПСП. Пересечение пусто тогда и только тогда, когда решений нет. Отметим, что технически мы свели ПСП к языку, который состоит из пар КС-грамматик с непустым пересечением их языков, т.е. показали, что проблема "является ли пересечение языков двух КС-грамматик непустым?" неразрешима. Однако, как указано во введении к доказательству, доказать неразрешимость дополнения проблемы — это то же самое, что доказать неразрешимость самой проблемы.
2. Поскольку КС-языки замкнуты относительно объединения, то можно построить КС-грамматику G_1 для $\overline{L_A} \cup \overline{L_B}$. Поскольку $(\Sigma \cup I)^*$ — регулярное множество, то для него, конечно же, можно построить КС-грамматику G_2. Далее, $\overline{L_A} \cup \overline{L_B} = \overline{L_A \cap L_B}$. Таким образом, в $L(G_1)$ отсутствуют лишь цепочки, которые представляют решения данного экземпляра ПСП. $L(G_2)$ содержит все цепочки из $(\Sigma \cup I)^*$, так что эти языки совпадают тогда и только тогда, когда данный экземпляр ПСП не имеет решения.
3. Рассуждения точно такие же, как и в п. 2, но полагаем R регулярным выражением $(\Sigma \cup I)^*$.
4. Достаточно рассуждений п. 3, поскольку $\Sigma \cup I$ — единственный алфавит, замыканием которого может быть $\overline{L_A} \cup \overline{L_B}$.
5. Пусть G_1 — КС-грамматика для $(\Sigma \cup I)^*$ и G_2 — КС-грамматика для $\overline{L_A} \cup \overline{L_B}$. Следовательно, $L(G_1) \subseteq L(G_2)$ тогда и только тогда, когда $\overline{L_A} \cup \overline{L_B} = (\Sigma \cup I)^*$, т.е. тогда и только тогда, когда данный экземпляр ПСП не имеет решения.
6. Рассуждения точно такие же, как и в п. 5, но полагаем R регулярным выражением $(\Sigma \cup I)^*$, а $L(G_1)$ — $\overline{L_A} \cup \overline{L_B}$.

□

9.5.4. Упражнения к разделу 9.5

9.5.1. (*) Пусть L — множество (кодов) КС-грамматик G, у которых $L(G)$ содержит хотя бы один палиндром. Покажите неразрешимость L. *Указание*. Сведите ПСП к L путем построения по каждому экземпляру ПСП грамматики, содержащей палиндром тогда и только тогда, когда этот экземпляр ПСП имеет решение.

9.5.2. (!) Покажите, что язык $\overline{L_A} \cup \overline{L_B}$ является регулярным тогда и только тогда, когда он представляет собой множество всех цепочек над своим алфавитом, т.е. экземпляр (A, B) ПСП не имеет решения. Таким образом, докажите неразрешимость вопроса, порождает ли КС-грамматика регулярный язык. *Указание*. Допустим, решение ПСП существует; например, в $\overline{L_A} \cup \overline{L_B}$ нет цепочки wx, где w — цепочка из алфавита экземпляра ПСП, а x — соответствующая цепочка индексных символов, записанная в обратном порядке. Определим гомоморфизм $h(0) = w$ и $h(1) = x$. Тогда, что представляет собой $h^{-1}(\overline{L_A} \cup \overline{L_B})$? В доказательстве того, что язык $\overline{L_A} \cup \overline{L_B}$ нерегулярен, используйте замкнутость регулярных множеств относительно обратного гомоморфизма и дополнения, а также лемму о накачке для регулярных множеств.

9.5.3. (!!) Вопрос о том, является ли дополнение КС-языка также КС-языком, неразрешим. С помощью упражнения 9.5.2 можно показать, что неразрешим вопрос о том, является ли дополнение КС-языка регулярным языком, но это не одно и то же. Чтобы доказать первое утверждение, нужно определить другой язык, который представляет цепочки, не являющиеся решениями экземпляра (A, B) ПСП. Пусть L_{AB} есть множество цепочек вида $w\#x\#y\#z$ со следующими свойствами.

1. w и x — цепочки над алфавитом Σ данного экземпляра ПСП.
2. y и z — цепочки над алфавитом индексов I данного экземпляра.
3. # — символ, не принадлежащий ни Σ, ни I.
4. $w \neq x^{\text{R}}$.
5. $y \neq z^{\text{R}}$.
6. x^{R} *не* совпадает с тем, что порождает цепочка индексов y в соответствии со списком B.
7. w *не* совпадает с тем, что порождает цепочка индексов z^{R} в соответствии со списком A.

Отметим, что L_{AB} состоит из всех цепочек, принадлежащих $\Sigma^*\#\Sigma^*\#I^*\#I^*$, если только экземпляр (A, B) ПСП не имеет решения, но независимо от этого L_{AB} является КС-языком. Докажите, что $\overline{L_{AB}}$ является КС-языком тогда и только тогда, когда решения нет. *Указание*. Как и в упражнении 9.5.2, используйте прием с обратным гомоморфизмом, а для того, чтобы добиться равенства длин тех или иных подцепочек, воспользуйтесь леммой Огдена, как в указании к упражнению 7.2.5, *б*.

9.6. Резюме

- *Рекурсивные и рекурсивно-перечислимые языки*. Языки, допускаемые машинами Тьюринга, называются рекурсивно-перечислимыми (РП), а РП-языки, допускаемые МТ, которые всегда останавливаются, — рекурсивными.
- *Дополнения рекурсивных и РП-языков*. Множество рекурсивных языков замкнуто относительно дополнения, и если язык и его дополнение являются РП, то оба они рекурсивны. Поэтому дополнение языка, являющегося РП, но не рекурсивным, не может быть РП.
- *Разрешимость и неразрешимость*. "Разрешимость" есть синоним "рекурсивности", однако языки чаще называются "рекурсивными", а проблемы (которые представляют собой языки, интерпретируемые как вопросы) — "разрешимыми". Если язык не является рекурсивным, то проблема, которую выражает этот язык, называется "неразрешимой".
- *Язык* L_d. В этот язык входит каждая цепочка в алфавите $\{0, 1\}$, которая, будучи проинтерпретированной как код МТ, *не* принадлежит языку этой МТ. Язык L_d является хорошим примером не РП-языка, т.е. его не допускает ни одна машина Тьюринга.
- *Универсальный язык*. Язык L_u состоит из цепочек, интерпретируемых как код МТ, к которому дописан ее вход. Цепочка принадлежит L_u, если эта МТ допускает данный вход. Язык L_u — это пример РП-языка, который не является рекурсивным.
- *Теорема Райса*. Всякое нетривиальное свойство языков, допускаемых МТ, является неразрешимым. Например, множество кодов машин Тьюринга, допускающих пустой язык, согласно теореме Райса является неразрешимым. В действительности этот язык не является РП, хотя его дополнение — множество кодов МТ, допускающих хотя бы одну цепочку, — является РП, но не рекурсивным.
- *Проблема соответствий Поста*. Заданы два списка, содержащие одинаковое количество цепочек. Спрашивается, можно ли, выбирая последовательности соответствующих цепочек из этих двух списков, построить путем их конкатенации одну и ту же цепочку. ПСП является важным примером неразрешимой проблемы. Сводимость ПСП к ряду других проблем обеспечивает доказательство их неразрешимости.
- *Неразрешимые проблемы, связанные с контекстно-свободными языками*. Посредством сведения ПСП можно доказать неразрешимость многих вопросов о КС-языках или их грамматиках. Так, например, неразрешимы вопросы о неоднозначности КС-грамматики, о включении одного КС-языка в другой или о пустоте пересечения двух КС-языков.

9.7. Литература

Результат о неразрешимости универсального языка фактически взят из работы Тьюринга [9], хотя там он выражен в терминах вычислений арифметических функций и останова, а не в терминах языков и допустимости по заключительному состоянию. Теорема Райса взята из [8].

Неразрешимость проблемы соответствий Поста была доказана в [7], но приведенное здесь доказательство, не вошедшее в печатные работы, принадлежит Р. В. Флойду (R. W. Floyd). Неразрешимость ТАГ-систем Поста (определенных в упражнении 9.4.4) доказывается в [6].

Основополагающими работами о неразрешимости вопросов, связанных с контекстно-свободными языками, являются [1] и [5]. Однако неразрешимость неоднозначных КС-грамматик была установлена независимо друг от друга Кантором [2], Флойдом [4], Хомским и Шютценберже [3].

1. Y. Bar-Hillel, M. Perles, and E. Shamir, “On formal properties of simple phrase-structure grammars”, *Z. Phonetik. Sprachwiss. Kommunikations-forsch.* **14** (1961), pp. 143–172.
2. D. C. Cantor, “On the ambiguity problem in Backus systems”, *J. ACM* **9**:4 (1962), pp. 477–479.
3. N. Chomsky and M. P. Schutzenberger, “The algebraic theory of context-free languages”, *Computer Programming and Formal Systems* (1963), North Holland, Amsterdam, pp. 118–161. (Хомский Н., Шютценберже М. Алгебраическая теория контекстно-свободных языков. — Кибернетический сборник, новая серия, вып. 3. — М.: Мир, 1966, С. 195–242.)
4. R. W. Floyd, “On ambiguity in phrase structure languages”, *Communications of the ACM* **5**:10 (1962), pp. 526–534.
5. S. Ginsburg and G. F. Rose, “Some recursively unsolvable problems in ALGOL-like languages”, *J. ACM* **10**:1 (1963), pp. 29–47.
6. M. L. Minsky, “Recursive unsolvability of Post’s problem of ‘tag’ and other topics in the theory of Turing machines”, *Annals of Mathematics* **74**:3 (1961), pp. 437–455.
7. E. Post, “A variant of a recursively unsolvable problem”, *Bulletin of the AMS* **52** (1946), pp. 264–268.
8. H. G. Rice, “Classes of recursively enumerable sets and their decision problems”, *Transactions of the AMS* **89** (1953), pp. 25–59.
9. A. M. Turing, “On computable numbers with an application to the Entscheidungsproblem”, *Proc. London Math. Society* **2**:42 (1936), pp. 230–265.

ГЛАВА 10

Труднорешаемые проблемы

Обсуждение вычислимости переносится на уровень ее эффективности или неэффективности. Предметом изучения становятся разрешимые проблемы, и нас интересует, какие из них можно решить на машинах Тьюринга за время, полиномиально зависящее от размера входных данных. Напомним два важных факта из раздела 8.6.3.

- Проблемы, разрешимые за полиномиальное время на обычном компьютере, — это именно те проблемы, которые разрешимы за такое же время с помощью машины Тьюринга.
- Практика показывает, что разделение проблем на разрешимые за полиномиальное время и требующие экспоненциального или большего времени, является фундаментальным. Полиномиальное время решения реальных задач, как правило, является приемлемым, тогда как задачи, требующие экспоненциального времени, практически (за приемлемое время) неразрешимы, за исключением небольших экземпляров.

В этой главе изучается теория “труднорешаемости”, т.е. методы доказательства неразрешимости проблем за полиномиальное время. Вначале рассматривается конкретный вопрос *выполнимости* булевой формулы, т.е. является ли она истинной для некоторого набора значений **ИСТИНА** и **ЛОЖЬ** ее переменных. Эта проблема играет в теории сложности такую же роль, как L_u и ПСП для неразрешимых проблем. Вначале мы докажем “теорему Кука”, которая означает, что проблему выполнимости булевых формул нельзя разрешить за полиномиальное время. Затем покажем, как свести эту проблему ко многим другим, доказывая тем самым их труднорешаемость.

При изучении полиномиальной разрешимости проблем изменяется понятие сведения. Теперь уже недостаточно просто наличия алгоритма, который переводит экземпляры одной проблемы в экземпляры другой. Алгоритм перевода сам по себе должен занимать не больше времени, чем полиномиальное, иначе сведение не позволит сделать вывод, что доказываемая проблема труднорешаема, даже если исходная проблема была таковой. Таким образом, в первом разделе вводится понятие “полиномиальной сводимости” (сводимости за полиномиальное время).

Между выводами, которые дают теории неразрешимости и труднорешаемости, существует еще одно принципиальное различие. Доказательства неразрешимости, приведенные в главе 9, неопровержимы. Они основаны только на определении машины Тьюринга и общих математических принципах. В отличие от них, все приво-

димые здесь результаты о труднорешаемости проблем основаны на недоказанном, но безоговорочно принимаемом на веру предположении, которое обычно называется предположением $\mathcal{P} \neq \mathcal{NP}$.

Итак, предполагается, что класс проблем, разрешимых недетерминированными МТ за полиномиальное время, содержит, по крайней мере, несколько проблем, которые нельзя решить за полиномиальное время детерминированными МТ (даже если для последних допускается более высокая степень полинома). Существуют буквально тысячи проблем, принадлежность которых данной категории *практически не вызывает сомнений*, поскольку они легко разрешимы с помощью НМТ с полиномиальным временем, но для их решения не известно ни одной ДМТ (или, что то же самое, компьютерной программы) с полиномиальным временем. Более того, важным следствием теории труднорешаемости является то, что либо все эти проблемы имеют детерминированные решения с полиномиальным временем — решения, ускользавшие от нас в течение целых столетий, либо таких решений не имеет ни одна из них, и они действительно требуют экспоненциального времени.

10.1. Классы $\mathcal{P}$ и $\mathcal{NP}$

В этом разделе вводятся основные понятия теории сложности: классы $\mathcal{P}$ и $\mathcal{NP}$ проблем, разрешимых за полиномиальное время, соответственно, детерминированными и недетерминированными МТ, а также метод полиномиального сведения. Кроме того, определяется понятие “NP-полноты” — свойства, которым обладают некоторые проблемы из $\mathcal{NP}$. Они, как минимум, так же трудны (в пределах полиномиальной зависимости времени), как любая проблема из класса $\mathcal{NP}$.

10.1.1. Проблемы, разрешимые за полиномиальное время

Говорят, что машина Тьюринга M имеет *временную сложность* $T(n)$ (или “время работы $T(n)$”), если, обрабатывая вход w длины n, M делает не более $T(n)$ переходов и останавливается независимо от того, допущен вход или нет. Это определение применимо к любой функции $T(n)$, например, $T(n) = 50n^2$ или $T(n) = 3^n + 5n^4$. Нас будет интересовать, главным образом, случай, когда $T(n)$ является полиномом относительно n. Мы говорим, что язык L принадлежит классу $\mathcal{P}$, если существует полином $T(n)$, при котором $L = L(M)$ для некоторой детерминированной МТ M с временной сложностью $T(n)$.

10.1.2. Пример: алгоритм Крускала

Читатель, вероятно, уже хорошо знаком со многими проблемами, имеющими эффективные решения. Некоторые из них, возможно, изучались в курсе по структурам данных и алгоритмам. Как правило, эти проблемы принадлежат классу $\mathcal{P}$. Рассмотрим одну такую проблему: поиск в графе остовного дерева минимального веса (*ОДМВ*).

Есть ли что-то между полиномами и экспонентами?

В дальнейшем, как и во вступлении, мы часто будем действовать так, как если бы время работы любой программы было либо полиномиальным ($O(n^k)$ для некоторого целого числа k), либо экспоненциальным ($O(2^{cn})$ для некоторой константы $c > 0$) или более того.

Известные из практики алгоритмы решения наиболее распространенных проблем, как правило, относятся к одной из этих категорий. Когда мы говорим об экспоненциальном времени, то всегда имеем в виду "время работы, которое больше любого полинома". Однако существуют времена работы программ, лежащие между полиномиальным и экспоненциальным временем.

Примером функции, находящейся между полиномами и экспонентами, является функция $n^{\log_2 n}$. Эта функция с ростом n увеличивается быстрее любого полинома, поскольку $\log n$ в конце концов (при больших n) превосходит любую константу k. С другой стороны, $n^{\log_2 n} = 2^{(\log_2 n)^2}$ (если это неочевидно, возьмите логарифм обеих частей). Эта функция растет медленнее, чем 2^{cn} при любой константе $c > 0$, поскольку cn в конце концов превышает $(\log_2 n)^2$, какой бы малой ни была c.

Неформально графы представляются как диаграммы, наподобие изображенной на рис. 10.1. У них есть узлы, которые в данном примере графа пронумерованы 1–4, и ребра, соединяющие некоторые пары узлов. Каждое ребро имеет целочисленный *вес*. *Остовное дерево* — это подмножество ребер, с помощью которых соединены все узлы, но циклы отсутствуют. Пример остовного дерева показан на рис. 10.1 — это три ребра, выделенные жирными линиями. *Остовное дерево минимального веса* — это дерево наименьшего общего веса среди всех остовных деревьев.

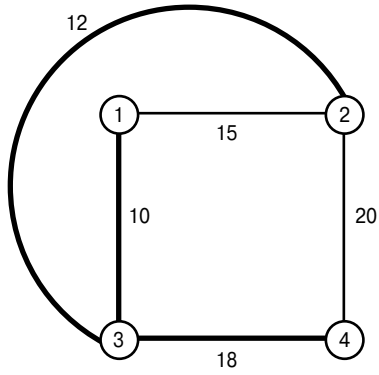


Рис. 10.1. Граф, в котором остовное дерево минимального веса выделено жирными линиями

Для нахождения ОДМВ существует хорошо известный “жадный” алгоритм, который называется *алгоритмом Крускала*[1]. Опишем вкратце его основные идеи.

1. Для каждого узла определяется *компонент связности*, который содержит этот узел и образован с использованием ребер, выбранных до сих пор. Вначале не выбрано ни одно ребро, так что каждый узел образует отдельный компонент связности.
2. Среди еще не выбранных ребер рассмотрим ребро минимального веса. Если оно соединяет два узла, которые в данный момент принадлежат различным компонентам связности, то выполняется следующее:
   а) ребро добавляется в остовное дерево;
   б) связные компоненты сливаются (объединяются) путем замены номера компонента у всех узлов одного из них номером другого.

   Если же выбранное ребро соединяет узлы из одного и того же компонента, то это ребро не принадлежит остовному дереву; его добавление привело бы к образованию цикла.
3. Выбор ребер продолжается до тех пор, пока мы не выберем все ребра, или число ребер, выбранных в остовное дерево, не станет на единицу меньше общего числа узлов. Заметим, что в последнем случае все узлы должны принадлежать одному компоненту связности, и процесс выбора ребер можно прекратить.

Пример 10.1. В графе на рис. 10.1 сначала рассматривается ребро (1, 3), поскольку оно имеет минимальный вес — 10. Так как вначале 1 и 3 принадлежат разным компонентам, это ребро принимается, а узлам 1 и 3 приписывается один и тот же номер компонента, скажем, “компонент 1”. Следующее по порядку возрастания веса ребро — (2, 3) с весом 12. Поскольку 2 и 3 принадлежат разным компонентам, то это ребро принимается, и 2 добавляется в “компонент 1”. Третье ребро — (1, 2) с весом 15. Но узлы 1 и 2 уже находятся в одном компоненте, поэтому данное ребро отбрасывается, и мы переходим к четвертому ребру — (3, 4). Поскольку 4 не содержится в “компоненте 1”, данное ребро принимается. Теперь у нас есть остовное дерево из трех ребер, и можно остановиться. □

Этот алгоритм обработки графа с m узлами и e ребрами можно реализовать (с помощью компьютера, а не машины Тьюринга) за время $O(m + e \log e)$. Более простая реализация имеет e этапов. Номер компонента каждого узла хранится в некоторой таблице. Выбор ребра наименьшего веса среди оставшихся занимает время $O(e)$, а поиск компонентов, в которых находятся узлы, связанные этим ребром, — $O(m)$. Если эти узлы принадлежат различным компонентам, то на просмотр таблицы для объединения всех узлов с соответствующими номерами нужно время $O(m)$. Таким образом, общее время работы этого алгоритма — $O(e(e + m))$. Это время полиномиально зависит от “размера” входных данных, в качестве которого можно неформально выбрать сумму e и m.

---

[1] J. B. Kruskal Jr., “On the shortest spanning subtree of a graph and the traveling salesman problem”, Proc. AMS **7**:1 (1956), pp.48–50.

При перенесении изложенных идей на машины Тьюринга возникают следующие вопросы.

- Выход алгоритмов разрешения различных проблем может иметь много разных форм, например, список ребер ОДМВ. Проблемы же, решаемые машинами Тьюринга, можно интерпретировать только как языки, а их выходом может быть только **ДА** или **НЕТ** (допустить или отвергнуть). Например, проблему поиска ОДМВ можно перефразировать так: "Для данного графа G и предельного числа W выяснить, имеет ли G остовное дерево с весом не более W?". Может показаться, что эту проблему решить легче, чем проблему ОДМВ в знакомой нам формулировке, поскольку не нужно даже искать остовное дерево. Однако в рамках теории труднорешаемости нам, как правило, нужно доказать, что проблема трудна (не легка). А из того, что "да/нет"-версия проблемы трудна, следует, что трудна и ее версия, предполагающая полный ответ.
- Хотя "размер" графа можно неформально представить себе как число его узлов или ребер, входом МТ является цепочка в некотором конечном алфавите. Поэтому такие элементы, фигурирующие в проблеме, как узлы и ребра, должны быть подходящим образом закодированы. Выполняя это требование, получаем в результате цепочки, длина которых несколько превышает предполагаемый "размер" входа. Однако есть две причины проигнорировать эту разницу.

1. Размеры входной цепочки МТ и входа неформальной проблемы всегда отличаются не более, чем малым сомножителем, обычно — логарифмом размера входных данных. Таким образом, все, что делается за полиномиальное время с использованием одной меры, можно сделать за полиномиальное время, используя другую меру.
2. Длина цепочки, представляющей вход, — в действительности более точная мера числа байтов, которые должен прочитать реальный компьютер, обрабатывая свой вход. Например, если узел задается целым числом, то количество байтов, необходимых для представления этого числа, пропорционально его логарифму, а это не "1 байт на каждый узел", как можно было бы предположить при неформальном подсчете размера входа.

Пример 10.2. Рассмотрим код для графов и предельных весов, который может быть входом для проблемы ОДМВ. В коде используются пять символов: 0, 1, левая и правая скобки, а также запятая.

1. Припишем всем узлам целые числа от 1 до m.
2. В начало кода поместим значения m и предельного веса W в двоичной системе счисления, разделенные запятой.
3. Если существует ребро, соединяющее узлы i и j и имеющее вес w, то в код записывается тройка (i, j, w), где целые числа i, j и w кодируются в двоичном виде. Порядок записи узлов ребра и порядок ребер в графе не играют роли.

Таким образом, один из возможных кодов графа на рис. 10.1 с предельным весом $W = 40$ имеет вид

100, 101000(1, 10, 1111)(1, 11, 1010)(10, 11, 1100)(10, 100, 10100)(11, 100, 10010).

□

Если кодировать входы проблемы ОДМВ так, как в примере 10.2, то вход длины n может представлять максимум $O(n/\log n)$ ребер. Если число ребер очень мало, то число узлов m может быть экспоненциальным относительно n. Однако если число ребер e меньше $m - 1$, то граф не может быть связным, и независимо от того, каковы эти ребра, не имеет ОДМВ. Следовательно, если количество узлов превосходит число $n/\log n$, то нет никакой необходимости запускать алгоритм Крускала — мы просто говорим: "нет, остовного дерева с таким весом не существует".

Итак, пусть у нас есть верхняя граница времени работы алгоритма Крускала, выражаемая в виде функции от m и e, например, как найденная выше верхняя граница $O(e(e + m))$. Можно изменить m и e на n и сказать, что время работы выражается функцией от длины входа n, имеющей вид $O(n(n + n))$ или $O(n^2)$. В действительности, более удачная реализация алгоритма Крускала требует времени $O(n \log n)$, но сейчас это не важно.

Представленный алгоритм Крускала предназначен для реализации на языке программирования с такими удобными структурами данных, как массивы и указатели, но в качестве модели вычислений мы используем машины Тьюринга. Тем не менее, описанную выше версию алгоритма можно реализовать за $O(n^2)$ шагов на многоленточной машине Тьюринга. Дополнительные ленты используются для выполнения нескольких вспомогательных задач.

1. На одной из лент можно хранить информацию об узлах и их текущих номерах компонентов. Длина такой таблицы составит $O(n)$.
2. Еще одна лента может применяться в процессе просмотра входной ленты для хранения информации о ребре, имеющем на данный момент наименьший вес среди ребер, не помеченных как "использованные" (выбранные). С помощью второй дорожки входной ленты можно отмечать те ребра, которые были выбраны в качестве ребер наименьшего веса на одном из предыдущих этапов работы алгоритма. Поиск непомеченного ребра наименьшего веса занимает время $O(n)$, поскольку каждое ребро рассматривается лишь один раз, и сравнить вес можно, просматривая двоичные числа линейно, справа налево.
3. Если ребро на некотором этапе выбрано, то соответствующие два узла помещаются на ленту. Чтобы установить компоненты этих двух узлов, нужно просмотреть таблицу узлов и компонентов. Это займет $O(n)$ времени.
4. Еще одна лента может хранить информацию об объединяемых компонентах i и j, когда найденное ребро соединяет различающиеся на данный момент компоненты. В этом

случае просматривается таблица узлов и компонентов, и для каждого узла из компонента i номер компонента меняется на j. Эта процедура также занимает $O(n)$ времени.

Теперь нетрудно самостоятельно завершить доказательство, что на многоленточной МТ каждый этап работы алгоритма может быть выполнен за время $O(n)$. Поскольку число этапов e не превышает n, делаем вывод, что времени $O(n^2)$ достаточно для многоленточной МТ. Теперь вспомним теорему 8.10, утверждавшую, что работу, которую многоленточная МТ выполняет за s шагов, можно выполнить на одноленточной МТ за $O(s^2)$ шагов. Таким образом, если многоленточной МТ требуется сделать $O(n^2)$ шагов, то можно построить одноленточную МТ, которая делает то же самое за $O((n^2)^2) = O(n^4)$ шагов. Следовательно, "да/нет"-версия проблемы ОДМВ ("имеет ли граф G ОДМВ с общим весом не более W?") принадлежит $\mathcal{P}$.

10.1.3. Недетерминированное полиномиальное время

Фундаментальный класс проблем в изучении труднорешаемости образован проблемами, разрешимыми с помощью недетерминированной МТ за полиномиальное время. Формально, мы говорим, что язык L принадлежит классу $\mathcal{NP}$ (недетерминированный полиномиальный), если существуют недетерминированная МТ M и полиномиальная временная сложность $T(n)$, для которых $L = L(M)$, и у M нет последовательностей переходов длиной более $T(n)$ при обработке входа длины n.

Поскольку всякая детерминированная МТ представляет собой недетерминированную МТ, у которой нет возможности выбора переходов, то $\mathcal{P} \subseteq \mathcal{NP}$. Однако оказывается, что в $\mathcal{NP}$ содержится множество проблем, не принадлежащих $\mathcal{P}$. Интуиция подсказывает: причина в том, что НМТ с полиномиальным временем работы имеет возможность угадывать экспоненциальное число решений проблемы и проверять "параллельно" каждое из них за полиномиальное время. И все-таки,

- одним из самых серьезных нерешенных вопросов математики является вопрос о том, верно ли, что $\mathcal{P} = \mathcal{NP}$, т.е. все, что с помощью НМТ делается за полиномиальное время, ДМТ в действительности также может выполнить за полиномиальное время (которое, возможно, выражается полиномом более высокой степени).

10.1.4. Пример из $\mathcal{NP}$: проблема коммивояжера

Для того чтобы осознать, насколько обширен класс $\mathcal{NP}$, рассмотрим пример проблемы, которая принадлежит классу $\mathcal{NP}$, но, предположительно, не принадлежит $\mathcal{P}$, — *проблему коммивояжера* (ПКОМ). Вход ПКОМ (как и у ОДМВ) — это граф, каждое ребро которого имеет целочисленный вес (рис. 10.1), а также предельный вес W. Вопрос состоит в том, есть ли в данном графе "гамильтонов цикл" с общим весом, не превышающим W. *Гамиль-*

тонов цикл — это множество ребер, соединяющих узлы в один цикл, причем каждый узел встречается в нем только один раз. Заметим, что число ребер в гамильтоновом цикле должно равняться числу узлов графа.

Одна из версий недетерминированной допустимости

Заметим, что мы требовали останова нашей НМТ за полиномиальное время на любой из ветвей (ее работы), независимо от того, допускает она или нет. Можно было бы также установить полиномиальное ограничение во времени $T(n)$ лишь для тех ветвей, которые ведут к допусканию, т.е. определить $\mathcal{NP}$ как класс языков, допускаемых НМТ, которые допускают с помощью хотя бы одной последовательности переходов длиной не более $T(n)$, где $T(n)$ — некоторый полином.

Однако в результате мы получили бы тот же самый класс языков, и вот почему. Если нам известно, что M, если вообще допускает, делает это в результате $T(n)$ переходов, то ее можно модифицировать так, чтобы на отдельной дорожке ленты она вела счет до $T(n)$ и останавливалась, не допуская, если счет превысит $T(n)$. Число шагов модифицированной M могло бы достигать $O(T^2(n))$. Но если $T(n)$ — полином, то и $T^2(n)$ — полином.

В действительности, класс $\mathcal{P}$ можно было бы также определить с помощью машин Тьюринга, допускающих за время $T(n)$, где $T(n)$ есть некоторый полином. Эти МТ могли бы не останавливаться, не допуская. Но с помощью такой же конструкции, как и для НМТ, мы могли бы модифицировать ДМТ так, чтобы она считала до $T(n)$ и останавливалась, перейдя эту границу. Время работы такой ДМТ было бы $O(T^2(n))$.

Пример 10.3. Граф, изображенный на рис. 10.1, имеет в действительности лишь один гамильтонов цикл — (1, 2, 4, 3, 1). Его общий вес составляет 15 + 20 + 18 + 10 = 63. Таким образом, если W имеет значение 63 или больше, то ответ — “да”, а если $W < 63$, то ответ — “нет”.

Однако простота ПКОМ для графа с четырьмя узлами обманчива. В данном графе попросту не может быть более двух различных гамильтоновых циклов, учитывая, что один и тот же цикл может иметь начало в разных узлах и два направления обхода. Но в графе с m узлами число различных циклов достигает $O(m!)$, факториала числа m, что превышает 2^{cm} для любой константы c. □

Оказывается, что любой способ решения ПКОМ включает перебор, по сути, всех циклов и подсчет общего веса каждого из них. Можно поступить умнее, отбросив некоторые, очевидно неподходящие, варианты. Однако, по всей видимости, при любых наших усилиях все равно придется рассматривать экспоненциальное число циклов перед тем, как убедиться в том, что ни один из них не имеет вес меньше предельного W, или отыскать такой цикл.

С другой стороны, если бы у нас был недетерминированный компьютер, то можно было бы “угадать” перестановку узлов и вычислить общий вес цикла, образованного узлами в этом порядке. Если бы существовал реальный недетерминированный компьютер, то при обработке входа длины n ни на одной из ветвей ему не пришлось бы сделать более $O(n)$ шагов. На многоленточной НМТ перестановку можно угадать за $O(n^2)$ шагов, и столько же времени понадобится для проверки ее общего веса. Таким образом, одноленточная НМТ может решить ПКОМ за время, не превышающее $O(n^4)$. Делаем вывод, что ПКОМ принадлежит классу $\mathcal{NP}$.

10.1.5. Полиномиальные сведения

Основной метод доказательства того, что проблему P_2 нельзя решить за полиномиальное время (т.е. P_2 не принадлежит $\mathcal{P}$), состоит в сведении к ней проблемы P_1, относительно которой известно, что она не принадлежит $\mathcal{P}$.[2] Данный подход был представлен на рис. 8.7, который воспроизводится здесь в виде рис. 10.2.

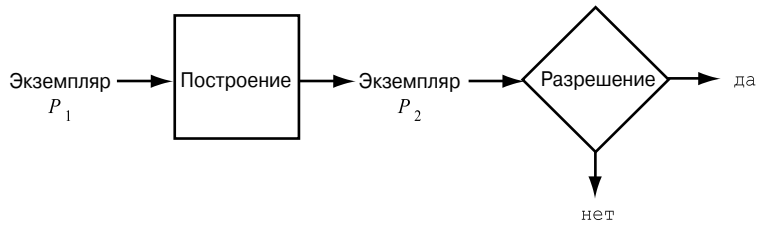


Рис. 10.2. Повторение картины сведения

Допустим, что мы хотим доказать утверждение: “если P_2 принадлежит $\mathcal{P}$, то и P_1 принадлежит $\mathcal{P}$”. Поскольку мы утверждаем, что P_1 *не принадлежит* $\mathcal{P}$, можно будет утверждать, что и P_2 не принадлежит $\mathcal{P}$. Однако одного лишь существования алгоритма, обозначенного на рис. 10.2 как “Построение”, не достаточно для доказательства нужного нам утверждения.

В самом деле, пусть по входному экземпляру проблемы P_1 длиной m алгоритм вырабатывает выходную цепочку длины 2^m, которая подается на вход гипотетическому алгоритму, решающему P_2 за полиномиальное время. Если решающий P_2 алгоритм делает это, скажем, за время $O(n^k)$, то вход длиной 2^m он обработает за время $O(2^{km})$, экспонен-

[2]Это утверждение не совсем верно. В действительности мы лишь *предполагаем*, что P_1 не принадлежит $\mathcal{P}$, на том весьма веском основании, что P_1 является “NP-полной” (понятие “NP-полноты” обсуждается в разделе 10.1.6). Затем мы доказываем, что P_2 также является “NP-полной” и, таким образом, не принадлежит $\mathcal{P}$ на том же основании, что и P_1.

циальное по m. Таким образом, алгоритм, решающий P_1, обрабатывает вход длиной m за время, экспоненциальное по m. Эти факты целиком согласуются с ситуацией, когда P_2 принадлежит $\mathcal{P}$, а P_1 — нет.

Даже если алгоритм, переводящий экземпляр P_1 в экземпляр P_2, всегда вырабатывает экземпляр, длина которого полиномиальна относительно длины входа, то мы все равно можем не добиться желаемого результата. Предположим, что построенный экземпляр P_2 имеет ту же длину m, что и P_1, но сам алгоритм преобразования занимает время, экспоненциальное по m, скажем, $O(2^m)$. Тогда из того, что алгоритм решения P_2 тратит на обработку входа длиной n время $O(n^k)$, следует лишь то, что существует алгоритм решения P_1, которому для обработки входа длиной m нужно время $O(2^m + m^k)$. В этой оценке времени работы учитывается тот факт, что необходимо не только решить итоговый экземпляр P_2, но и получить его. И вновь возможно, что P_2 принадлежит $\mathcal{P}$, а P_1 — нет.

Правильное ограничение, которое необходимо наложить на преобразование P_1 в P_2, состоит в том, что время его выполнения должно полиномиально зависеть от размера входных данных. Отметим, что если при входе длины m преобразование занимает время $O(m^j)$, то длина выходного экземпляра P_2 не может превышать числа совершенных шагов, т.е. она не больше cm^j, где c — некоторая константа. Теперь можно доказать, что если P_2 принадлежит $\mathcal{P}$, то и P_1 принадлежит $\mathcal{P}$.

Для доказательства предположим, что принадлежность цепочки длины n к P_2 можно выяснить за время $O(n^k)$. Тогда вопрос о принадлежности P_1 цепочки длины m можно решить за время $O(m^j + (cm^j)^k)$; слагаемое m^j учитывает время преобразования, а $(cm^j)^k$ — время выяснения, является ли результат экземпляром P_2. Упрощая выражение, видим, что P_1 может быть решена за время $O(m^j + cm^{jk})$. Поскольку c, j и k — константы, это время зависит от m полиномиально, и, следовательно, P_1 принадлежит $\mathcal{P}$.

Итак, в теории труднорешаемости используются только *полиномиальные сведения* (*сведения за полиномиальное время*). Сведение P_1 к P_2 является полиномиальным, если оно занимает время, полиномиальное по длине входного экземпляра P_1. Как следствие, длина выходного экземпляра P_2 будет полиномиальной по длине экземпляра P_1.

10.1.6. NP-полные проблемы

Теперь познакомимся с группой проблем, которые являются наиболее известными кандидатами на то, чтобы принадлежать $\mathcal{NP}$, но не принадлежать $\mathcal{P}$. Пусть L — язык (проблема) из класса $\mathcal{NP}$. Говорят, что L является *NP-полным*, если для него справедливы следующие утверждения.

1. L принадлежит $\mathcal{NP}$.
2. Для всякого языка L' из $\mathcal{NP}$ существует полиномиальное сведение L' к L.

Как мы увидим, примером NP-полной проблемы является проблема коммивояжера (см. раздел 10.1.4). Предполагается, что $\mathcal{P} \neq \mathcal{NP}$, и, в частности, что все NP-полные проблемы содержатся в $\mathcal{NP} - \mathcal{P}$, поэтому доказательство NP-полноты проблемы можно рассматривать как свидетельство того, что она не принадлежит $\mathcal{P}$.

Вначале докажем NP-полноту так называемой проблемы выполнимости булевой формулы (ВЫП), показав, что язык всякой НМТ с полиномиальным временем полиномиально сводится к ВЫП. Имея в распоряжении некоторые NP-полные проблемы, можно доказать NP-полноту еще одной, новой проблемы посредством полиномиального сведения к ней одной из известных проблем. Следующая теорема объясняет, почему такое сведение доказывает, что новая проблема является NP-полной.

Теорема 10.4. Если проблема P_1 является NP-полной, и существует полиномиальное сведение P_1 к P_2, то проблема P_2 также NP-полна.

Доказательство. Нужно показать, что всякий язык L из $\mathcal{NP}$ полиномиально сводится к P_2. Мы знаем, что существует полиномиальное сведение L к P_1; это сведение занимает некоторое полиномиальное время $p(n)$. Поэтому цепочка w из L длиной n преобразуется в цепочку x из P_1, длина которой не превосходит $p(n)$.

Мы также знаем, что существует полиномиальное сведение P_1 к P_2; пусть это сведение занимает полиномиальное время $q(m)$. Тогда это сведение преобразует x в некоторую цепочку y из P_2, за время, не превосходящее $q(p(n))$. Таким образом, преобразование w в y занимает времени не более $p(n) + q(p(n))$, которое является полиномиальным. Следовательно, L полиномиально сводим к P_2. Поскольку L — произвольный язык из $\mathcal{NP}$, то мы показали, что все проблемы класса $\mathcal{NP}$ полиномиально сводятся к P_2, т.е. P_2 является NP-полной. □

NP-трудные проблемы

Некоторые проблемы L трудны настолько, что, хотя и можно доказать, что для них выполняется условие 2 из определения NP-полноты (всякий язык из $\mathcal{NP}$ сводится к L за полиномиальное время), условие 1 — что L принадлежит $\mathcal{NP}$ — мы доказать не можем. Если это так, то L называется NP-трудной. До сих пор в отношении проблем, требующих для решения экспоненциального времени, использовался нестрогий термин “труднорешаемая”. Вообще говоря, термин “труднорешаемая” в значении “NP-трудная” использовать можно, хотя могут существовать проблемы, требующие экспоненциального времени, несмотря на то, что в строгом смысле они не являются NP-трудными.

Для доказательства NP-трудности проблемы L достаточно показать высокую вероятность того, что L требует экспоненциального или большего времени. Однако если L не принадлежит классу $\mathcal{NP}$, то ее очевидная трудность никак не может служить подтверждением того, что все NP-полные проблемы трудны, т.е. может выясниться, что $\mathcal{P} = \mathcal{NP}$, но при этом L все равно требует экспоненциального времени.

Есть еще одна, более важная, теорема, которую нужно доказать для NP-полных проблем: если любая из них принадлежит $\mathcal{P}$, то весь класс $\mathcal{NP}$ содержится в $\mathcal{P}$. Но мы твердо верим, что в $\mathcal{NP}$ содержится много проблем, *не принадлежащих* $\mathcal{P}$. Таким образом, доказательство NP-полноты проблемы мы считаем равносильным доказательству того, что у нее нет алгоритма решения с полиномиальным временем, и поэтому она не имеет хорошего компьютерного решения.

Теорема 10.5. Если некоторая NP-полная проблема P принадлежит $\mathcal{P}$, то $\mathcal{P} = \mathcal{NP}$.

Доказательство. Предположим, что P одновременно и NP-полна, и принадлежит $\mathcal{P}$. Тогда любой язык L из $\mathcal{NP}$ полиномиально сводится к P. Если P принадлежит $\mathcal{P}$, то и L принадлежит $\mathcal{P}$ (см. раздел 10.1.5). □

10.1.7. Упражнения к разделу 10.1

10.1.1. Каким будет ОДМВ для графа на рис 10.1, если вес его ребер изменить следующим образом:

а) (*) вес 10 ребра (1, 3) изменить на 25;

б) изменить вес ребра (2, 4) на 16.

Альтернативные определения NP-полноты

Конечной целью изучения NP-полноты является теорема 10.5, т.е. поиск проблем P, принадлежность которых классу $\mathcal{P}$ влечет равенство $\mathcal{P} = \mathcal{NP}$. Определение "NP-полноты", использованное здесь, часто называется *полнотой по Карпу*, поскольку оно впервые было использовано Р. Карпом в фундаментальной работе на данную тему. Этому определению соответствует любая проблема, предположительно удовлетворяющая условиям теоремы 10.5.

Однако существуют другие, более широкие понятия NP-полноты, для которых теорема 10.5 также справедлива. Например, С. Кук в своей первой работе, посвященной данному предмету, дал следующее определение "NP-полноты" проблемы P. Проблему P он назвал NP-полной, если, имея для проблемы P *оракул* — механизм, который за единицу времени определяет, принадлежит ли данная цепочка P, — можно распознать всякий язык из $\mathcal{NP}$ за полиномиальное время. Этот тип NP-полноты называют *полнотой по Куку*. В некотором смысле, полнота по Карпу есть частный случай этой полноты, когда оракулу задается лишь один вопрос. Однако полнота по Куку допускает отрицание ответа. Можно, например, задать оракулу некоторый вопрос, а в качестве ответа взять противоположное тому, что он ответит. Согласно определению полноты по Куку, дополнение NP-полной проблемы также является NP-полной проблемой. Используя же более узкое определение полноты по Карпу, можно указать важное отличие NP-полных проблем от их дополнений. Это делается в разделе 11.1.

10.1.2. Каким будет гамильтонов цикл минимального веса для графа на рис. 10.1, если в него добавить ребро с весом 19, соединяющее узлы 1 и 4?

10.1.3. (***!**) Предположим, что существует NP-полная проблема с детерминированным решением, занимающим время $O(n^{\log_2 n})$. Заметим, что эта функция лежит между полиномами и экспонентами и не принадлежит ни одному из этих классов функций. Что можно сказать о времени, необходимом для решения произвольной проблемы из $\mathcal{NP}$?

10.1.4. (**!!**) Рассмотрим графы, узлами которых являются узлы целочисленной решетки в n-мерном кубе со стороной длины m, т.е. векторы $(i_1, i_2, \ldots, i_n)$, где каждое i_j находится в диапазоне от 1 до m (или от 0 до $m - 1$). Ребро между двумя узлами существует тогда и только тогда, когда они различаются на единицу ровно по одной координате. Например, вариант $n = 2$ и $m = 2$ представляет собой квадрат, $n = 3$ и $m = 2$ — куб, а $n = 2$ и $m = 3$ — граф, изображенный на рис. 10.3. Некоторые из этих графов имеют гамильтонов цикл, некоторые — нет. Например, квадрат имеет такой цикл. Куб — тоже, хотя это и не очевидно; примером является цикл (0, 0, 0), (0, 0, 1), (0, 1, 1), (0, 1, 0), (1, 1, 0), (1, 1, 1), (1, 0, 1), (1, 0, 0) и снова (0, 0, 0). Граф на рис. 10.3 гамильтонова цикла не имеет.

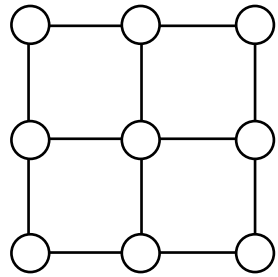


Рис. 10.3. Граф, соответствующий n = *2 и* m = *3*

а) Докажите, что граф на рис. 10.3 не имеет гамильтонова цикла. *Указание.* Нужно рассмотреть ситуацию, когда гипотетический гамильтонов цикл проходит через центральный узел. Откуда он может исходить и куда он может вести, не отсекая при этом части графа от гамильтонова цикла?

б) Для каких значений n и m гамильтонов цикл существует?

10.1.5. (**!**) Предположим, что у нас есть способ кодировки контекстно-свободных грамматик с помощью некоторого конечного алфавита. Рассмотрим следующие два языка.

1. $L_1 = \{(G, A, B) \mid G$ — (закодированная) КС-грамматика, A и B — (закодированные) переменные G, причем множества терминальных цепочек, выводимых из A и B, совпадают$\}$.

2. $L_2 = \{(G_1, G_2) \mid G_1$ и G_2 — (закодированные) КС-грамматики, и $L(G_1) = L(G_2)\}$.

Выполните следующие задания:

а) (*****) покажите, что L_1 полиномиально сводится к L_2;

б) покажите, что L_2 полиномиально сводится к L_1;

в) (*****) что можно сказать об NP-полноте L_1 и L_2 на основании *а* и *б*?

10.1.6. $\mathcal{P}$ и $\mathcal{NP}$, как классы языков, обладают определенными свойствами замкнутости. Покажите, что класс $\mathcal{P}$ замкнут относительно следующих операций:

а) обращение;

б) (*****) объединение;

в) (***!**) конкатенация;

г) (**!**) замыкание (звездочка);

д) обратный гомоморфизм;

е) (*****) дополнение.

10.1.7. Класс $\mathcal{NP}$ также замкнут относительно каждой из операций, перечисленных в упражнении 10.1.6, за (предполагаемым) исключением операции дополнения (пункт *е*). Замкнут ли класс $\mathcal{NP}$ относительно дополнения — неизвестно; этот вопрос обсуждается в разделе 11.1. Докажите, что $\mathcal{NP}$ замкнут относительно операций из пунктов *а–д* упражнения 10.1.6.

10.2. Первая NP-полная проблема

Теперь познакомимся с первой NP-полной проблемой — проблемой выполнимости булевых формул. Ее NP-полнота доказывается путем непосредственного сведения к ней языка любой недетерминированной МТ с полиномиальным временем.

10.2.1. Проблема выполнимости

Булевы формулы (*выражения*) строятся из следующих элементов.

1. Булевы переменные, принимающие значения 1 (истина) или 0 (ложь).
2. Бинарные операторы $\wedge$ и $\vee$, обозначающие логические связки **И** и **ИЛИ** двух формул.
3. Унарный оператор $\neg$, который обозначает логическое отрицание.
4. Скобки для группирования операторов и операндов, если необходимо изменить порядок старшинства (приоритетов) операторов, принятый по умолчанию (вначале применяется $\neg$, затем $\wedge$ и, наконец, $\vee$).

Пример 10.6. Примером булевой формулы является $x \wedge \neg(y \vee z)$. Подформула $y \vee z$ имеет значение “истина”, если истинна хотя бы одна из переменных y или z, и “ложь”, если

обе они ложны. Бо́льшая подформула $\neg(y \vee z)$ истинна лишь тогда, когда $y \vee z$ ложно, т.е. когда обе переменные ложны. Если хотя бы одна из y или z истинна, то $\neg(y \vee z)$ ложно.

Рассмотрим, наконец, всю формулу. Поскольку она представляет собой логическое **И** двух подформул, то она истинна только тогда, когда истинны обе подформулы, т.е. $x \wedge \neg(y \vee z)$ истинна тогда и только тогда, когда x истинна, а y и z ложны. □

Выбрать *подстановку* (*набор значений переменных*) для данной булевой формулы E — значит приписать значение "истина" или "ложь" каждой из переменных, фигурирующих в E. *Значение* формулы E при данной подстановке T, обозначаемое как $E(T)$, — это результат вычисления E, в которой каждая из ее переменных x заменена значением $T(x)$ ("истина" или "ложь"), которое соответствует x в T.

Формула E *истинна* при подстановке T, или подстановка T *удовлетворяет* формуле E, если $E(T) = 1$, т.е. данная подстановка T делает формулу E истинной. Булева формула E называется *выполнимой*, если существует хотя бы одна подстановка T, для которой E истинна.

Пример 10.7. Формула $x \wedge \neg(y \vee z)$ из примера 10.6 выполнима. Мы убедились, что эта формула истинна при подстановке T, определенной равенствами $T(x) = 1$, $T(y) = 0$ и $T(z) = 0$, поскольку принимает значение 1. Мы также заметили, что T — *единственная* подстановка, удовлетворяющая этой формуле, поскольку для остальных семи комбинаций значений трех переменных формула ложна (принимает значение 0).

В качестве еще одного примера рассмотрим формулу $E = x \wedge (\neg x \vee y) \wedge \neg y$. Утверждаем, что E невыполнима. Поскольку переменных в ней всего две, то число возможных подстановок есть $2^2 = 4$, так что нетрудно проверить их и убедиться, что при всех формула E принимает значение 0. Однако это можно обосновать и по-другому. E истинна только тогда, когда все три члена, связанных $\wedge$, истинны. Это значит, что x должно быть истинным (из-за первого члена), а y — ложным (из-за третьего члена). Но для такой подстановки значение среднего члена $\neg x \vee y$ ложно. Таким образом, значение E не может быть истинным, и E действительно невыполнима.

В одном из рассмотренных примеров у формулы была лишь одна удовлетворяющая подстановка, а в другом их вообще не было. Существует также множество примеров, в которых у формулы есть более одной удовлетворяющей подстановки. Например, рассмотрим $F = x \vee \neg y$. Значение F есть 1 для трех следующих подстановок.

1. $T_1(x) = 1$; $T_1(y) = 1$.
2. $T_2(x) = 1$; $T_2(y) = 0$.
3. $T_3(x) = 0$; $T_3(y) = 0$.

F принимает значение 0 лишь для четвертой подстановки, где $x = 0$ и $y = 1$. Таким образом, F выполнима. □

Проблема выполнимости состоит в следующем.

- Выяснить, выполнима ли данная булева формула.

Проблема выполнимости обычно обозначается как ВЫП. Если рассматривать ее как язык, то проблема ВЫП есть множество (закодированных) выполнимых булевых формул. Цепочки, которые либо не образуют правильные коды булевых формул, либо являются кодами невыполнимых булевых формул, не принадлежат ВЫП.

10.2.2. Представление экземпляров ВЫП

Символами в булевых формулах являются $\wedge$, $\vee$, $\neg$, левые и правые скобки, а также символы, представляющие переменные. Выполнимость формулы зависит не от имен переменных, а от того, являются ли два вхождения переменных одной и той же переменной или двумя разными. Таким образом, можно считать, что переменными являются x_1, x_2, …, хотя в примерах в качестве имен переменных по-прежнему используется не только x, но и имена вроде y или z. Считается также, что переменные переименованы так, что используются наименьшие из возможных индексов. Например, переменная x_5 не используется, если в той же формуле не использованы переменные x_1–x_4.

Поскольку число переменных, встречающихся в булевых формулах, может быть бесконечным, то мы сталкиваемся с уже знакомой проблемой разработки кода, имеющего фиксированный конечный алфавит, для представления формул с произвольным, сколь угодно большим, числом переменных. Только имея такой код, можно говорить о ВЫП как о “проблеме”, т.е. языке с фиксированным алфавитом, состоящем из кодов выполнимых булевых формул. Будем использовать следующий код.

1. Символы $\wedge$, $\vee$, $\neg$, (и) представляют самих себя.
2. Переменная x_i представляется символом x с дописанной к нему последовательностью нулей и единиц — двоичной записью числа i.

Таким образом, алфавит проблемы/языка ВЫП содержит всего восемь символов. Все экземпляры ВЫП являются цепочками символов в этом фиксированном конечном алфавите.

Пример 10.8. Рассмотрим формулу $x \wedge \neg(y \vee z)$ из примера 10.6. Первое, что нужно сделать для ее кодирования, — заменить переменные индексированными символами x. Поскольку переменных три, следует использовать x_1, x_2 и x_3. Выбор x_i для замены переменных x, y и z зависит от нас. Положим для определенности $x = x_1$, $y = x_2$ и $z = x_3$. Тогда формула принимает вид $x_1 \wedge \neg(x_2 \vee x_3)$, и ее кодом является $x1 \wedge \neg(x10 \vee x11)$. □

Отметим, что длина закодированной булевой формулы примерно равна числу позиций в этой формуле, если считать каждое вхождение переменной за 1. Разница возникает из-за того, что если формула имеет m позиций, то она может содержать $O(m)$ переменных, и для кодирования каждой из них может понадобиться $O(\log m)$ символов. Таким образом, формула длиной m позиций может иметь код длиной $n = O(m \log m)$ символов.

Однако разница между m и $m \log m$, очевидно, ограничена полиномом. Поэтому, если нас интересует лишь, можно ли решить проблему за время, полиномиально зависящее от размера входных данных, то длину кода формулы и число позиций в ней различать не обязательно.

10.2.3. NP-полнота проблемы ВЫП

Докажем “теорему Кука”, утверждающую, что ВЫП NP-полна. Для доказательства, что некоторая проблема является NP-полной, сначала нужно показать, что она принадлежит классу $\mathcal{NP}$, а затем — что к ней сводится любой язык из $\mathcal{NP}$. Как правило, для доказательства второй части к нашей проблеме полиномиально сводится какая-либо другая NP-полная проблема, а затем применяется теорема 10.5. Но пока у нас нет ни одной NP-полной проблемы, которую можно было бы свести к ВЫП. Поэтому нам остается только сводить каждую без исключения проблему из $\mathcal{NP}$ к ВЫП.

Теорема 10.9 (теорема Кука). Проблема ВЫП NP-полна.

Доказательство. В первой части доказывается, что ВЫП принадлежит $\mathcal{NP}$. Сделать это довольно легко.

1. С помощью свойства недетерминированности НМТ для данной формулы E угадываем подстановку T. Если код E имеет длину n, то многоленточной НМТ для этого достаточно времени $O(n)$. Заметим, что у НМТ есть много возможностей выбора переходов, и по окончании процесса угадывания она может иметь 2^n различных МО, где каждая ветка представляет угадывание отдельной подстановки.
2. Находим значение E для подстановки T. Если $E(T) = 1$, то допускаем. Отметим, что эта часть детерминирована. Тот факт, что другие ветки могут не приводить к допусканию, никак не влияет на результат, поскольку НМТ допускает даже в том случае, если найдена всего одна удовлетворяющая подстановка.

Вычислить значение формулы с помощью многоленточной НМТ легко за время $O(n^2)$. Таким образом, весь процесс распознавания ВЫП многоленточной НМТ занимает время $O(n^2)$. Преобразование в одноленточную НМТ может возвести время в квадрат, так что одноленточной НМТ будет достаточно времени $O(n^4)$.

Теперь нужно доказать трудную часть — что для произвольного языка L из $\mathcal{NP}$ существует полиномиальное сведение L к ВЫП. Можно предположить, что существует некоторая одноленточная НМТ M и полином $p(n)$, для которого M обрабатывает вход длиной n не более, чем за $p(n)$ шагов вдоль любой ветки. Далее, ограничения, доказанные в теореме 8.12 для ДМТ, можно аналогично доказать и для НМТ. Поэтому можно предположить, что M никогда не печатает пробела и никогда не сдвигает головку левее ее исходной позиции.

Итак, если M допускает вход w, и $|w| = n$, то существует последовательность переходов M со следующими свойствами.

1. α_0 — исходное МО M со входом w.
2. $\alpha_0 \vdash \alpha_{01} \vdash \cdots \vdash \alpha_k$, где $k \le p(n)$.
3. α_k — МО с допускающим состоянием.

4. Каждое α_i не содержит пробелов (за исключением тех α_i, которые заканчиваются состоянием и пробелом) и располагается вправо от исходной позиции головки — крайнего слева входного символа.

Стратегия построения формулы вкратце такова.

1. Каждое α_i может быть записано как последовательность символов $X_{i0}X_{i1}\ldots X_{i,p(n)}$ длиной $p(n) + 1$. Один из них — состояние, а остальные $p(n)$ — ленточные символы. Как обычно, предположим, что множества состояний и ленточных символов не пересекаются, так что можно отличить, какое из X_{ij} является состоянием, и указать, где находится головка. Отметим, что нет надобности представлять символы, расположенные справа от первых $p(n)$ символов на ленте, поскольку, если M гарантированно останавливается, сделав не более $p(n)$ переходов, то на переходы M эти символы справа не влияют.
2. Для описания последовательности МО в терминах булевых переменных введем переменные y_{ijA}, которые представляют утверждения вида $X_{ij} = A$. Здесь i и j — целые из интервала от 0 до $p(n)$, а A — либо ленточный символ, либо состояние.
3. Условие того, что последовательность МО представляет допускание, записывается в виде булевой формулы, выполнимой тогда и только тогда, когда M допускает w в результате последовательности не более чем $p(n)$ переходов. Удовлетворяющая подстановка "характеризует" МО, т.е. y_{ijA} истинна тогда и только тогда, когда $X_{ij} = A$. Для гарантии того, что полиномиальное сведение $L(M)$ к ВЫП корректно, эта формула записывается так, чтобы отражать следующие свойства вычисления.
    i. *Правильный старт* — исходное МО есть q_0w с последующими пробелами.
    ii. *Правильные переходы* (т.е. согласующиеся с правилами МТ) — каждое последующее МО получается из предыдущего путем выполнения одного из возможных законных переходов M.
    iii. *Правильный финиш* — существует МО с допускающим состоянием.

Прежде, чем точно описать построение нашей булевой формулы, рассмотрим несколько деталей.

- Во-первых, по построению МО заканчивается там, где начинается бесконечный "хвост" из пробелов. Однако, имитируя вычисление за полиномиальное время, удобнее считать, что все МО имеют одну и ту же длину $p(n) + 1$. Поэтому в МО может присутствовать хвост из пробелов.
- Во-вторых, удобно считать, что все вычисления происходят в точности за $p(n)$ переходов (и, следовательно, имеют $p(n) + 1$ МО), даже если допускание происходит раньше. Следовательно, всякому МО с допускающим состоянием позволено быть собственным преемником, т.е., когда α содержит допускающее состояние, разрешен "переход" $\alpha \vdash \alpha$. Таким образом, можно предполагать, что если

существует допускающее вычисление, то $\alpha_{p(n)}$ имеет допускающее состояние, и это все, что нужно проверить в условии (iii).

На рис. 10.4 представлен общий вид полиномиального вычисления M. Строки соответствуют последовательности МО, а столбцы — это клетки на ленте, которые могут использоваться при вычислении. Заметим, что число ячеек на рис. 10.4 равно $(p(n) + 1)^2$. Кроме того, число различных значений переменных, представляющих ячейки, конечно и зависит только от M; оно равно сумме числа состояний и числа ленточных символов M.

МО	0	1	...					...	$p(n)$
α_0	X_{00}	X_{01}							$X_{0,p(n)}$
α_1	X_{10}	X_{11}							$X_{1,p(n)}$
α_i				$X_{i,j-1}$	$X_{i,j}$	$X_{i,j+1}$			
α_{i+1}				$X_{i+1,j-1}$	$X_{i+1,j}$	$X_{i+1,j+1}$			
$\alpha_{p(n)}$	$X_{p(n),0}$	$X_{p(n),1}$							$X_{p(n),p(n)}$

Рис. 10.4. Построение массива данных о последовательности МО

Приведем теперь алгоритм построения булевой формулы $E_{M,w}$ по M и w. Общий вид $E_{M,w}$ — $S \wedge N \wedge F$, где формулы S, N и F говорят о том, что M совершает правильный старт, переходы и финиш.

Правильный старт

Символ X_{00} должен быть начальным состоянием q_0 машины M, символы с X_{01} по X_{0n} — образовывать цепочку w (n — ее длина), а оставшиеся X_{0j} — быть пробелами B, т.е. если $w = a_1a_2\ldots a_n$, то

$$S = y_{00q0} \wedge y_{01a1} \wedge y_{02a2} \wedge \cdots \wedge y_{0na_n} \wedge y_{0,n+1,B} \wedge y_{0,n+2,B} \wedge \cdots \wedge y_{0,p(n),B}.$$

Безусловно, по данным M и w можно записать S на второй ленте многоленточной МТ за время $O(p(n))$.

Правильный финиш

Поскольку предполагается, что допускающее МО повторяется до бесконечности, то допускание M эквивалентно присутствию допускающего состояния в $\alpha_{p(n)}$. Напомним, что по предположению M является такой НМТ, которая, если допускает, то делает это в пределах

$p(n)$ шагов. Таким образом, F есть логическое **ИЛИ** формул F_j для $j = 0, 1, \ldots, p(n)$, где F_j утверждает, что $X_{p(n),j}$ — допускающее состояние, т.е. F_j есть $y_{p(n),j,a1} \vee y_{p(n),j,a2} \vee \cdots \vee y_{p(n),j,ak}$, где $a_1, a_2, \ldots, a_k$ — все допускающие состояния M. Тогда

$$F = F_0 \vee F_1 \vee \ldots \vee F_{p(n)}.$$

Отметим, что в каждом из F_i используется постоянное число символов, которое зависит от M, а не от длины n входа w. Поэтому F имеет длину $O(p(n))$. Более важно то, что время, необходимое для записи F по данным коду M и цепочке w, полиномиально зависит от n. В действительности, формулу F можно записать на многоленточной МТ за время $O(p(n))$.

Правильные переходы

Намного сложнее убедиться, что M правильно выполняет переходы. Формула N будет логическим **И** формул N_i, где $i = 0, 1, \ldots, p(n) - 1$, и каждое N_i строится так, чтобы α_{i+1} было одним из МО, в которые можно перейти из α_i по правилам M. Чтобы понять, как следует записать N_i, рассмотрим символ $X_{i+1,j}$ на рис. 10.4. Символ $X_{i+1,j}$ всегда можно определить, зная:

а) три символа над ним, т.е. $X_{i,j-1}$, $X_{i,j}$ и $X_{i,j+1}$;

б) выбор перехода НМТ M, если один из этих символов есть состояние в α_i.

Запишем N_i как логическое **И** формул $A_{ij} \vee B_{ij}$, где $j = 0, 1, \ldots, p(n)$.

- Формула A_{ij} говорит о том, что:

  а) состояние МО α_i находится в позиции j, т.е. X_{ij} — состояние;

  б) если X_{ij} — состояние и $X_{i,j+1}$ — обозреваемый символ, то существует выбор такого перехода M, который преобразует последовательность символов $X_{i,j-1}X_{ij}X_{i,j+1}$ в $X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1}$. Заметим, что если X_{ij} — допускающее состояние, то возможен только "выбор", при котором переход вообще не совершается, поэтому все последующие МО совпадают с первым, приведшим к допусканию.

- Формула B_{ij} говорит о том, что:

  а) состояние МО α_i достаточно далеко от X_{ij}, так что оно не может влиять на $X_{i+1,j}$ (ни один из символов $X_{i,j-1}$, X_{ij}, $X_{i,j+1}$ не является состоянием);

  б) $X_{i+1,j} = X_{ij}$.

Формулу B_{ij} записать легче, чем A_{ij}. Пусть $q_1, q_2, \ldots, q_m$ — состояния, а $Z_1, Z_2, \ldots, Z_r$ — ленточные символы. Тогда

$$B_{ij} = (y_{i,j-1,Z1} \vee y_{i,j-1,Z2} \vee \cdots \vee y_{i,j-1,Zr}) \wedge$$
$$(y_{i,j,Z1} \vee y_{i,j,Z2} \vee \cdots \vee y_{i,j,Zr}) \wedge$$
$$(y_{i,j+1,Z1} \vee y_{i,j+1,Z2} \vee \cdots \vee y_{i,j+1,Zr}) \wedge$$
$$(y_{i,j,Z1} \wedge y_{i+1,j,Z1}) \vee (y_{i,j,Z2} \wedge y_{i+1,j,Z2}) \vee \cdots \vee (y_{i,j,Zr} \wedge y_{i+1,j,Zr}).$$

Первая, вторая и третья строчки в B_{ij} соответственно выражают, что $X_{i,j-1}$, X_{ij} и $X_{i,j+1}$ — ленточные символы. Последняя строчка выражает равенство $X_{i+1,j} = X_{ij}$; в ней перечисляются все возможные ленточные символы Z, и говорится, что обе переменные — это либо Z_1, либо Z_2 и т.д.

Существует два важных частных случая: $j = 0$ и $j = p(n)$. В первом нет переменных $y_{i,j-1,Z}$, а во втором — переменных $y_{i,j+1,Z}$. Однако нам известно, что головка никогда не сдвигается влево от своего исходного положения и что ей не хватит времени, чтобы сдвинуться более чем на $p(n)$ клеток вправо от исходной. Поэтому из B_{i0} и $B_{i,p(n)}$ можно выбросить некоторые члены. Выполните это упрощение самостоятельно.

Рассмотрим теперь формулы A_{ij}. Эти формулы отражают все возможные связи между символами $X_{i,j-1}$, X_{ij}, $X_{i,j+1}$, $X_{i+1,j-1}$, $X_{i+1,j}$ и $X_{i+1,j+1}$ в прямоугольниках размером 2×3 из массива (см. рис. 10.4). Подстановка символов в каждую из этих шести переменных является *правильной*, если:

а) X_{ij} есть состояние, а $X_{i,j-1}$ и $X_{i,j+1}$ — ленточные символы;

б) состоянием является только один из $X_{i+1,j-1}$, $X_{i+1,j}$ и $X_{i+1,j+1}$;

в) существует переход M, объясняющий, как $X_{i,j-1}X_{ij}X_{i,j+1}$ превращается в $X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1}$.

Таким образом, для этих шести переменных существует лишь конечное число правильных подстановок символов. Пусть A_{ij} будет логическим **ИЛИ** членов, по одному для каждого множества из шести переменных, образующего правильную подстановку.

Предположим, что переход M обусловлен тем, что $\delta(q, A)$ содержит (p, C, L). Пусть D — некоторый ленточный символ M. Тогда $X_{i,j-1}X_{ij}X_{i,j+1} = DqA$ и $X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1} = pDC$ — одна из правильных подстановок. Заметим, как эта подстановка отражает изменение МО, вызванное данным переходом M. Член, отражающий эту возможность, имеет вид

$$y_{i,j-1,D} \wedge y_{i,j,q} \wedge y_{i,j+1,A} \wedge y_{i+1,j-1,p} \wedge y_{i+1,j,D} \wedge y_{i+1,j+1,C}.$$

Если же $\delta(q, A)$ содержит (p, C, R), т.е. переход такой же, но головка сдвигается вправо, то соответствующая правильная подстановка — это $X_{i,j-1}X_{ij}X_{i,j+1} = DqA$ и $X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1} = DCp$. Соответствующий член имеет вид

$$y_{i,j-1,D} \wedge y_{i,j,q} \wedge y_{i,j+1,A} \wedge y_{i+1,j-1,D} \wedge y_{i+1,j,C} \wedge y_{i+1,j+1,p}.$$

Формула A_{ij} есть логическое **ИЛИ** всех правильных членов. В особых случаях, когда $j = 0$ и $j = p(n)$, нужно внести некоторые изменения, чтобы учесть отсутствие переменных y_{ijZ} при $j < 0$ или $j > p(n)$, как для B_{ij}.

Наконец,

$$N_i = (A_{i0} \vee B_{i0}) \wedge (A_{i1} \vee B_{i1}) \wedge \cdots \wedge (A_{i,p(n)} \vee B_{i,p(n)})$$

и

$$N = N_0 \wedge N_1 \wedge \cdots \wedge N_{p(n)-1}.$$

Несмотря на то что формулы A_{ij} и B_{ij} могут быть очень громоздкими (если M имеет много состояний и/или ленточных символов), их размер представляет собой константу и не зависит от длины входа w. Таким образом, длина N_i есть $O(p(n))$, а длина N — $O(p^2(n))$. Более важно то, что формулу N можно записать на ленту многоленточной МТ за время, пропорциональное ее длине, и это время полиномиально зависит от n — длины цепочки w.

Завершение доказательства теоремы Кука

Конструкция формулы

$$E_{M,w} = S \wedge N \wedge F$$

была описана как функция, зависящая и от M, и от w, но в действительности только часть S — "правильный старт" — зависит от w, причем зависимость эта очень простая (w находится на ленте начального МО). Остальные части, N и F, зависят только от M и n — длины входа w.

Итак, для всякой НМТ M с полиномиальным временем работы $p(n)$ можно построить алгоритм, который, получая на вход цепочку w длины n, выдает $E_{M,w}$. Время работы такого алгоритма на многоленточной недетерминированной МТ есть $O(p^2(n))$, а многоленточную МТ, в свою очередь, можно преобразовать в одноленточную МТ со временем работы $O(p^4(n))$. Выходом данного алгоритма является булева формула $E_{M,w}$, выполнимая тогда и только тогда, когда M допускает w в пределах первых $p(n)$ переходов. □

Для того чтобы подчеркнуть исключительную значимость теоремы Кука, рассмотрим применение к ней теоремы 10.5. Допустим, что экземпляры ВЫП могли бы распознаваться некоторой НМТ за полиномиальное время, скажем, $q(n)$. Тогда всякий язык, допускаемый некоторой НМТ M за полиномиальное время $p(n)$, допускался бы за полиномиальное время ДМТ, структура которой представлена на рис. 10.5. Вход w машины M преобразуется в булеву формулу $E_{M,w}$. Затем эта формула подается на вход машины, распознающей ВЫП, и ответ, который она выдает для входа $E_{M,w}$, будет ответом нашего алгоритма для входа w.

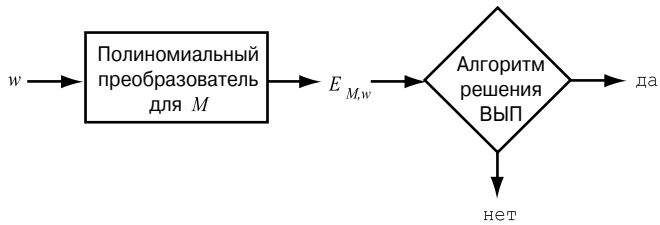


Рис. 10.5. Если ВЫП принадлежит $\mathcal{P}$, то принадлежность $\mathcal{P}$ произвольного языка из $\mathcal{NP}$ может быть доказана с помощью ДМТ указанного здесь вида

10.2.4. Упражнения к разделу 10.2

10.2.1. Сколько удовлетворяющих подстановок имеют следующие формулы? Какие из них принадлежат ВЫП?

а) (*) $x \wedge (y \vee \neg x) \wedge (z \vee \neg y)$;

б) $(x \vee y) \wedge (\neg(x \vee z) \vee (\neg z \wedge \neg y))$.

10.2.2. (!) Рассмотрим граф G с четырьмя узлами: 1, 2, 3 и 4. Пусть x_{ij}, где $1 \leq i < j \leq 4$, — булева переменная, интерпретируемая как высказывание: "существует ребро, соединяющее узлы i и j". Всякий граф, содержащий эти четыре узла, можно представить некоторой подстановкой. Например, граф, изображенный на рис. 10.1, представляется подстановкой, где x_{14} имеет значение "ложь", а остальные переменные — "истина". Всякое свойство графа, касающееся только наличия или отсутствия тех или иных ребер, можно описать с помощью булевой формулы, истинной тогда и только тогда, когда подстановка описывает граф с данным свойством. Запишите формулы для следующих свойств:

а) (*) G имеет гамильтонов цикл;

б) G — связный;

в) G содержит клику размера 3 (треугольник), т.е. множество из трех узлов, каждые два из которых связаны ребром;

г) G содержит хотя бы один изолированный узел (не имеющий ребер).

10.3. Ограниченная проблема выполнимости

Мы планируем доказать NP-полноту целого ряда проблем, таких как ПКОМ, упоминаемая в разделе 10.1.4. Это можно сделать с помощью сведения проблемы ВЫП к каждой интересующей нас проблеме. Однако существует промежуточная проблема, называемая "ЗВЫП", которую намного проще свести к типичным проблемам, чем ВЫП. В проблеме ЗВЫП речь по-прежнему идет о выполнимости булевых формул (выражений), но эти выражения имеют строго определенный вид: они представляют собой логическое **И** "сумм", каждая из которых является логическим **ИЛИ** ровно трех переменных или их отрицаний.

В этом разделе вводится ряд важных терминов, относящихся к булевым формулам. Затем выполнимость формулы произвольного вида сводится к выполнимости выражения в форме, нормальной для проблемы ЗВЫП. Отметим, что всякая булева формула E имеет эквивалентное выражение F в 3-КНФ, но размер F может экспоненциально зависеть от размера E. Поэтому по сравнению с обычными для булевой алгебры преобразованиями полиномиальное сведение ВЫП к проблеме ЗВЫП должно быть более тонким. Всякую

формулу E из ВЫП нужно преобразовать в формулу F, находящуюся в 3-КНФ, но F не обязательно должна быть эквивалентной E. Достаточно убедиться, что F выполнима тогда и только тогда, когда выполнима E.

10.3.1. Нормальные формы булевых выражений

Дадим три основных определения.

- *Литералом* является либо переменная, либо отрицание переменной. Например, x или $\neg y$. Для краткости вместо литерала вида $\neg y$ часто используется переменная с чертой сверху, $\bar{y}$.
- *Дизъюнктом* называется логическое **ИЛИ** одного или нескольких литералов, например x, $x \vee y$, $x \vee \neg y \vee z$.
- Говорят, что формула записана в *конъюнктивной нормальной форме* (КНФ) (здесь замысловатый термин "конъюнкция" обозначает логическое **И**), если представляет собой логическое **И** дизъюнктов.

Для придания записываемым выражениям большей компактности примем альтернативные обозначения. Оператор $\vee$ рассматривается как сложение, и вместо него используется оператор +, а $\wedge$ — как умножение, знак которого обычно опускается. Сомножители записываются рядом, как при конкатенации в регулярных выражениях. Тогда дизъюнкт естественно называть "суммой литералов", а КНФ — "произведением дизъюнктов (сумм)".

Пример 10.10. В сжатых обозначениях формула $(x \vee \neg y) \wedge (\neg x \vee z)$ имеет вид $(x + \bar{y})(\bar{x} + z)$. Она записана в КНФ, так как представляет собой логическое **И** (произведение) сумм $(x + \bar{y})$ и $(\bar{x} + z)$.

Формула $(x + y\bar{z})(x + y + z)(\bar{y} + \bar{z})$ не находится в КНФ. Она представляет собой логическое **И** трех сумм $(x + y\ \bar{z})$, $(x + y + z)$ и $(\bar{y}\ +\ \bar{z})$, однако первая из них — не дизъюнкт, так как является суммой литерала и произведения двух литералов.

Формула xyz находится в КНФ. Напомним, что дизъюнкт может состоять и из одного-единственного литерала. Таким образом, наша формула представляет собой произведение трех дизъюнктов: (x), (y) и (z). □

Говорят, что формула записана в *k-конъюнктивной нормальной форме* (k-КНФ), если представляет собой произведение дизъюнктов, каждый из которых является суммой ровно k различных литералов. Например, формула $(x + \bar{y})(y + \bar{z})(z + \bar{x})$ записана в 2-КНФ, так как каждый из ее дизъюнктов содержит ровно два литерала.

Все эти условия, накладываемые на булевы формулы, приводят к собственным проблемам выполнимости формул, удовлетворяющих этим условиям. Таким образом, мы будем говорить о следующих проблемах.

- Проблема ВКНФ: выполнима ли данная булева формула, записанная в КНФ?
- Проблема k-ВЫП: выполнима ли данная булева формула, находящаяся в k-КНФ?

Обработка плохих входов

Каждая рассмотренная проблема (ВЫП, ВКНФ, 3ВЫП и т.д.) — это язык с фиксированным алфавитом из восьми символов, цепочки которого иногда можно интерпретировать как булевы формулы. Цепочка, которую нельзя интерпретировать как булеву формулу, не может принадлежать языку ВЫП. Аналогично, если рассматриваются формулы ограниченного вида, то цепочка, представляющая собой правильную булеву формулу, но не выражение требуемого вида, не может принадлежать данному языку. Например, алгоритм, решающий проблему ВКНФ, выдаст ответ "нет", если ему на вход подать булеву формулу, которая выполнима, но не находится в КНФ.

Мы увидим, что проблемы ВЫП, 3ВЫП и k-ВЫП при $k > 3$ NP-полны, но для проблем 1ВЫП и 2ВЫП существуют алгоритмы с линейным временем работы.

10.3.2. Преобразование формул в КНФ

Две булевы формулы[3] называются *эквивалентными*, если имеют одно и то же значение при любой подстановке. Если две формулы эквивалентны, то они либо обе выполнимы, либо обе невыполнимы. Поэтому, на первый взгляд, преобразование произвольных формул в эквивалентные формулы, записанные в КНФ, позволило бы разработать метод полиномиального сведения ВЫП к ВКНФ. Это сведение свидетельствовало бы об NP-полноте ВКНФ.

Однако не все так просто. Мы действительно можем преобразовать всякую формулу в КНФ, но время этого преобразования может оказаться больше полиномиального. В частности, при таком преобразовании длина формулы может вырасти экспоненциально, и тогда, безусловно, время порождения выхода также экспоненциально возрастет.

К счастью, приведение произвольной булевой формулы к КНФ — это лишь один из возможных способов сведения ВЫП к ВКНФ, и доказательства, таким образом, NP-полноты ВКНФ. В действительности *достаточно* взять экземпляр ВЫП E и преобразовать его в экземпляр ВКНФ F, выполнимый тогда и только тогда, когда выполним E. Формулы E и F могут быть неэквивалентными. Не требуется даже, чтобы множества переменных E и F совпадали; множество переменных F будет, как правило, надмножеством множества переменных E.

Сведение ВЫП к ВКНФ состоит из двух частей. На первом этапе все отрицания $\neg$ "спускаются" вниз по дереву выражения, так что в формуле присутствуют только отрицания переменных. Булева формула превращается в логическое **И** и **ИЛИ** литералов. Это преобразование дает формулу, эквивалентную исходной, и занимает время, как максимум, квадратичное относительно длины этой формулы. При реализации на обычном компьютере с тщательно выбранной структурой данных это преобразование требует линейного времени.

Второй этап — переписать формулу, которая представляет собой логическое **И** и **ИЛИ** литералов, в виде произведения дизъюнктов, т.е. привести ее к КНФ. Введение новых переменных позволяет провести это преобразование за время, полиномиально зависящее от размера исходной формулы. Новая формула F, вообще говоря, не будет эквивалентна старой формуле E. Но F будет выполнима тогда и только тогда, когда выполнима E. Точнее, если T — некоторая подстановка, для которой E истинна, то существует расширение T, скажем S, для которого истинна F. *Расширение* S подстановки T — это подстановка, приписывающая переменным T те же значения, что и T, и, кроме того, в ней есть переменные, которых не было в T.

На первом этапе нужно спустить операторы $\neg$ ниже операторов $\wedge$ и $\vee$. Для этого используются следующие правила.

1. $\neg(E \wedge F) \Rightarrow \neg(E) \vee \neg(F)$. Это правило — один из *законов Де Моргана* — позволяет спустить оператор $\neg$ под оператор $\wedge$. Отметим, что в виде побочного эффекта $\wedge$ меняется на $\vee$.
2. $\neg(E \vee F) \Rightarrow \neg(E) \wedge \neg(F)$. Другой "закон Де Моргана" позволяет спустить $\neg$ под оператор $\vee$, изменяемый на $\wedge$.
3. $\neg(\neg(E)) \Rightarrow E$. Этот *закон двойного отрицания* уничтожает пару операторов $\neg$, применяемых к одной формуле.

Пример 10.11. Рассмотрим формулу $E = \neg((\neg(x + y))(\bar{x} + y))$. Заметим, что в ней перемешаны оба используемых обозначения. Оператор $\neg$ использован явно, когда формула, отрицание которой берется, состоит более чем из одной переменной. На рис. 10.6 показано, как формула переписывается шаг за шагом до тех пор, пока отрицания $\neg$ не станут составляющими литералов.

Формула	Правило
$\neg((\neg(x + y))(\bar{x} + y))$	Начало
$\neg(\neg(x + y)) + \neg(\bar{x} + y)$	(1)
$x + y + \neg(\bar{x} + y)$	(3)
$x + y + (\neg(\bar{x}))\,\bar{y}$	(2)
$x + y + x\,\bar{y}$	(3)

Рис. 10.6. Спуск отрицаний вниз по дереву формулы так, чтобы они встречались только в литералах

[3] С общим набором переменных. — *Прим. перев.*

Заключительная формула эквивалентна исходной и состоит из логических связок **И** и **ИЛИ** литералов. Ее можно упростить до формулы $x + y$, но это упрощение несущественно для нашего утверждения о том, что формулу можно переписать так, чтобы оператор $\neg$ присутствовал только в литералах. □

Теорема 10.12. Всякая булева формула E эквивалентна формуле F, в которой отрицания присутствуют только в литералах, т.е. применяются непосредственно к переменным. Более того, длина F линейно зависит от количества символов в E, и F можно построить по E за полиномиальное время.

Доказательство. Используем индукцию по числу операторов ($\wedge$, $\vee$ и $\neg$) в E. Покажем, что существует эквивалентная формула F, в которой $\neg$ присутствует только в литералах. Кроме того, если E содержит $n \geq 1$ операторов, то F содержит не более $2n - 1$ операторов.

Поскольку формуле F достаточно одной пары скобок для каждого оператора, а число переменных в ней не может превышать числа операторов больше, чем на единицу, то приходим к выводу, что длина F линейно пропорциональна длине E. Более важно, как мы увидим, что время построения F пропорционально ее длине и, следовательно, — длине E.

Базис. Если E содержит один оператор, то она должна иметь вид $\neg x$, $x \vee y$ или $x \wedge y$, где x и y — переменные. В каждом из этих случаев формула уже имеет требуемый вид, поэтому $F = E$. Отметим, что, поскольку и E, и F содержат по одному оператору, то соотношение "число операторов в F не превышает числа операторов в E, умноженного на два, минус один" выполняется.

Индукция. Предположим, что утверждение справедливо для всех формул, число операторов в которых меньше, чем в E. Если верхний оператор в E — не $\neg$, то формула должна иметь вид $E_1 \vee E_2$ или $E_1 \wedge E_2$. В любом из этих случаев гипотеза индукции применима к формулам E_1 и E_2, и согласно ей существуют эквивалентные формулы — F_1 и F_2, соответственно, — в которых $\neg$ встречается только в литералах. Тогда $F = F_1 \vee F_2$ или $F = (F_1) \wedge (F_2)$ служат подходящими эквивалентами для E. Пусть E_1 и E_2 содержат соответственно a и b операторов. Тогда E содержит $a + b + 1$ операторов. Согласно гипотезе индукции F_1 и F_2 содержат соответственно $2a - 1$ и $2b - 1$ операторов. Таким образом, F содержит не более, чем $2a + 2b - 1$ операторов, что не превосходит $2(a + b + 1) - 1$, или числа операторов в E, умноженного на два, минус 1.

Рассмотрим теперь случай, когда E имеет вид $\neg E_1$. В зависимости от верхнего оператора в формуле E_1 возможны три варианта. Заметим, что E_1 содержит хотя бы один оператор, иначе E — формула из базиса.

1. $E_1 = \neg E_2$. Тогда согласно закону двойного отрицания формула $E = \neg\neg(E_2)$ эквивалентна E_2. Гипотеза индукции применима, так как в E_2 содержится меньше операторов, чем в E. Можно найти формулу F, эквивалентную E_2, в которой отрицания встречаются только в литералах. Формула F годится и для E. Поскольку число опе-

раторов в F не превышает числа операторов в E_2, умноженного на два, минус 1, то оно, конечно же, не превышает числа операторов в E, умноженного на два, минус 1.

2. $E_1 = E_2 \vee E_3$. Согласно закону Де Моргана формула $E = \neg(E_2 \vee E_3)$ эквивалентна $(\neg(E_2)) \wedge (\neg(E_3))$. Обе формулы $\neg(E_2)$ и $\neg(E_3)$ содержат меньше операторов, чем E. Поэтому согласно гипотезе индукции существуют эквивалентные им формулы F_2 и F_3, в которых отрицания встречаются только в литералах. Тогда эквивалентом E служит формула $F = (F_2) \wedge (F_3)$. Кроме того, мы утверждаем, что число операторов в F не слишком велико. Пусть E_2 и E_3 содержат соответственно a и b операторов. Тогда в E содержится $a + b + 2$ операторов. Поскольку в формулах $\neg(E_2)$ и $\neg(E_3)$ соответственно $a + 1$ и $b + 1$ операторов, и формулы F_2 и F_3 построены по этим формулам, то согласно гипотезе индукции в F_2 и F_3 не больше, чем $2(a + 1) - 1$ и $2(b + 1) - 1$ операторов соответственно. Таким образом, F содержит не более $2a + 2b + 3$ операторов, а это и есть число операторов в E, умноженное на два, минус 1.

3. $E_1 = E_2 \wedge E_3$. Обоснование этой части, использующее второй закон Де Моргана, аналогично пункту 2.

□

Описания алгоритмов

Формально время работы алгоритма сведения есть время, необходимое для выполнения его на одноленточной машине Тьюринга, однако такие алгоритмы слишком сложны. Мы знаем, что множества проблем, которые можно решить за полиномиальное время на обычных компьютерах, многоленточных и одноленточных МТ, совпадают, хотя степени полиномов при этом могут различаться. Таким образом, описывая некоторые по-настоящему сложные алгоритмы, требующие сведения одной NP-полной проблемы к другой, примем соглашение, что время сведения будет ограничено временем работы его эффективной реализации на обычном компьютере. Это избавит нас от подробностей обработки лент и позволит выделить по-настоящему важные алгоритмические идеи.

10.3.3. NP-полнота проблемы ВКНФ

Теперь нам необходимо привести к КНФ формулу E, которая состоит из логических **И** и **ИЛИ** литералов. Как уже упоминалось, чтобы за полиномиальное время создать по E формулу F, выполнимую тогда и только тогда, когда выполнима E, нужно отказаться от преобразования, сохраняющего эквивалентность формул, и ввести в F новые переменные, отсутствующие в E. Используем этот прием в доказательстве теоремы об NP-полноте проблемы ВКНФ, а затем приведем пример его применения.

Теорема 10.13. Проблема ВКНФ NP-полна.

Доказательство. Покажем, как свести ВЫП к ВКНФ за полиномиальное время. Вначале с помощью метода, описанного в теореме 10.12, преобразуем данный экземпляр ВЫП в формулу E, содержащую $\neg$ только в литералах. Затем покажем, как за полиномиальное время преобразовать E в КНФ-формулу F, выполнимую тогда и только тогда, когда выполнима E. Формула F строится индуктивно по длине E, а ее конкретные свойства — это даже больше, чем нам нужно. Точнее, индукцией по числу вхождений символов в E ("длине") докажем следующее утверждение.

- Пусть E — булева формула длины n, в которой $\neg$ встречается только в литералах. Тогда существуют константа c и формула F, удовлетворяющие утверждениям:
  а) F находится в КНФ, и содержит не более n дизъюнктов;
  б) F можно построить по E за время, не превышающее $c|E|^2$;
  в) подстановка T для E делает E истинной тогда и только тогда, когда существует расширение S подстановки T, удовлетворяющее формуле F.

Базис. Если E состоит из одного или двух символов, то оно — литерал. Литерал является дизъюнктом, поэтому E уже находится в КНФ.

Индукция. Предположим, что всякую формулу, более короткую, чем E, можно преобразовать в произведение дизъюнктов, и для формулы длиной n это преобразование занимает время не более cn^2. В зависимости от верхнего оператора E возможны два варианта.

Вариант 1. $E = E_1 \wedge E_2$. Согласно гипотезе индукции существуют формулы F_1 и F_2, которые выводятся из E_1 и E_2, соответственно, и находятся в КНФ. Все подстановки, удовлетворяющие E_1, и только они, могут быть расширены до подстановок, удовлетворяющих F_1. То же самое верно для E_2 и F_2. Не теряя общности, можно предполагать, что переменные в F_1 и F_2 различны, за исключением переменных, присутствующих в E, т.е. вводимые в F_1 и/или F_2 переменные выбираются различными.

Рассмотрим $F = F_1 \wedge F_2$. Очевидно, что формула $F_1 \wedge F_2$ находится в КНФ, если в КНФ находятся F_1 и F_2. Нам нужно показать, что подстановку T для формулы E можно расширить до подстановки, удовлетворяющей F, тогда и только тогда, когда T удовлетворяет E.

(*Достаточность*) Пусть T удовлетворяет E. Пусть T_1 и T_2 — сужения подстановки T, применяемые только к переменным формул E_1 и E_2, соответственно. Тогда согласно индуктивной гипотезе T_1 и T_2 могут быть расширены до подстановок S_1 и S_2, удовлетворяющих F_1 и F_2, соответственно. Пусть S — подстановка, согласованная с S_1 и S_2, т.е. приписывает переменным те же значения, что S_1 и S_2. Заметим, что лишь переменные из E присутствуют и в F_1, и в F_2, поэтому подстановки S_1 и S_2 согласованы на переменных, на которых обе они определены, и построить S всегда возможно. Но тогда S есть расширение T, удовлетворяющее F.

(*Необходимость*) Наоборот, пусть подстановка T имеет расширение S, удовлетворяющее F. Пусть T_1 (T_2) — сужение T на переменные E_1 (E_2). Сужение S на переменные F_1 (F_2) обозначим через S_1 (S_2). Тогда S_1 — это расширение T_1, а S_2 — расширение T_2.

Поскольку F есть логическое **И** формул F_1 и F_2, S_1 должна удовлетворять формуле F_1, а S_2 — формуле F_2. Согласно гипотезе индукции T_1 (T_2) должна удовлетворять E_1 (E_2). Поэтому T удовлетворяет E.

Вариант 2. $E = E_1 \vee E_2$. Как и в предыдущем случае, обращаемся к индуктивной гипотезе, утверждающей, что существуют формулы F_1 и F_2, которые обладают следующими свойствами.

1. Подстановка для формулы E_1 (E_2) удовлетворяет E_1 (E_2) тогда и только тогда, когда она может быть расширена до подстановки, удовлетворяющей формуле F_1 (F_2).
2. Переменные в формулах F_1 и F_2 различны, за исключением присутствующих в E.
3. Формулы F_1 и F_2 находятся в КНФ.

Для того чтобы построить искомую формулу F, мы не можем просто объединить F_1 и F_2 логическим **ИЛИ**, так как полученная в результате формула не будет находиться в КНФ. Однако можно использовать более сложную конструкцию, которая учитывает, что важно сохранить выполнимость формул, а не их эквивалентность. Предположим, что $F_1 = g_1 \wedge g_2 \wedge \ldots \wedge g_p$ и $F_2 = h_1 \wedge h_2 \wedge \ldots \wedge h_q$, где символы g и h обозначают дизъюнкты. Введем новую переменную y и определим

$$F = (y + g_1) \wedge (y + g_2) \wedge \ldots \wedge (y + g_p) \wedge (\bar{y} + h_1) \wedge (\bar{y} + h_2) \wedge \ldots \wedge (\bar{y} + h_q).$$

Мы должны доказать, что подстановка T удовлетворяет E тогда и только тогда, когда T может быть расширена до подстановки S, удовлетворяющей F.

(*Необходимость*) Пусть подстановка T удовлетворяет E. Как и в варианте 1, обозначим через T_1 (T_2) сужение T на переменные E_1 (E_2). Поскольку $E = E_1 \vee E_2$, то T удовлетворяет E_1 или E_2. Предположим, что E_1 истинна при T. Тогда подстановку T_1, представляющую собой сужение T на переменные E_1, можно расширить до подстановки S_1, для которой истинна F_1. Расширение S подстановки T, для которого истинна определенная выше формула F, построим следующим образом.

1. $S(x) = S_1(x)$ для всех переменных x из F_1.
2. $S(y) = 0$. Этим выбором всем дизъюнктам F, полученным из F_2, придается значение “истина”.
3. Для всех переменных x из F_2, отсутствующих в F_1, $S(x)$ может принимать значения 0 или 1 произвольно.[4]

По правилу 1 подстановка S делает истинными все дизъюнкты, полученные из дизъюнктов g, а по правилу 2 — все дизъюнкты, полученные из дизъюнктов h. Поэтому подстановка S удовлетворяет формуле F.

[4] В силу п. 2 не обязательно даже, чтобы $S(x) = T(x)$ при тех x, для которых $T(x)$ определено. — *Прим. ред.*

Если подстановка T не удовлетворяет E_1, но удовлетворяет E_2, то расширение строится аналогично, за исключением того, что $S(y) = 1$ в правиле 2. Подстановка $S(x)$ согласуется с $S_2(x)$ на переменных, на которых определена $S_2(x)$, а значения переменных, присутствующих только в S_1, в подстановке S произвольны. Приходим к выводу, что и в этом случае F истинна при S.

(*Достаточность*) Предположим, что подстановка T для E расширена до подстановки S для F, и F истинна при S. В зависимости от значения переменной y возможны два случая. Пусть $S(y) = 0$. Тогда все дизъюнкты, полученные из дизъюнктов h, истинны. Однако y ложно в дизъюнктах вида $(y + g_i)$, получаемых из g_i. Это значит, что S придает значение "истина" самим g_i, так что F_1 истинна при подстановке S.

Более строго, пусть S_1 — сужение S на переменные F_1. Тогда F_1 истинна при S_1. Поскольку S_1 — это расширение подстановки T_1, являющейся сужением T на переменные E_1, то согласно гипотезе индукции E_1 должна быть истинной при подстановке T_1. Но F_1 истинна при T_1, поэтому формула E, представляющая собой $E_1 \vee E_2$, должна быть истинной при T.

Остается рассмотреть случай $S(y) = 1$, аналогичный предыдущему, и это предоставляется читателю. Итак, E истинна при T, если только F истинна при S.

Теперь нужно показать, что время, необходимое для построения F по E, не превышает квадрата n — длины E. Независимо от возможного случая, обе процедуры — разбиение E на E_1 и E_2 и построение формулы F по F_1 и F_2 — занимают время, линейно зависящее от размера E. Пусть dn — верхняя граница времени, необходимого для построения формул E_1 и E_2 по E, вместе со временем, затрачиваемым на построение формулы F по F_1 и F_2, в любом из вариантов 1 и 2. Тогда $T(n)$ — время, необходимое для построения F по E, длина которой n, описывается следующим рекуррентным соотношением.

$T(1) = T(2) \leq e$, где e — некоторая константа.

$T(n) \leq dn + c \max_{0<i<n-1} (T(i) + T(n - 1 - i))$ для $n \geq 3$.

Константу c еще предстоит определить так, чтобы $T(n) \leq cn^2$. Базисное правило для $T(1)$ и $T(2)$ говорит о том, что если E — одиночный символ или пара символов, то рекурсия не нужна, поскольку E может быть только одиночным литералом, и весь процесс занимает некоторое время e. В рекурсивном правиле используется тот факт, что E составлена из подформул E_1 и E_2, связанных оператором $\wedge$ или $\vee$, и, если E_1 имеет длину i, то E_2 имеет длину $n - i - 1$. Более того, весь процесс построения F по E состоит из двух простых шагов — замены E формулами E_1 и E_2 и замены F_1 и F_2 формулой F, — которые, как мы знаем, занимают время, не превышающее dn, плюс два рекурсивных преобразования E_1 в F_1 и E_2 в F_2.

Докажем индукцией по n, что существует такая константа c, при которой $T(n) \leq cn^2$ для всех $n > 0$.

Базис. Для $n = 1$ нам нужно просто выбрать c не меньше, чем e.

Индукция. Допустим, что утверждение справедливо для длин, которые меньше n. Тогда $T(i) \le ci^2$ и $T(n-i-1) \le c(n-i-1)^2$. Таким образом,

$$T(i) + T(n-i-1) \le n^2 - 2i(n-i) - 2(n-i) + 1 \qquad (10.1)$$

Поскольку $n \ge 3$ и $0 < i < n-1$, то $2i(n-i)$ не меньше n, а $2(n-i)$ не меньше 2. Поэтому для любого i в допустимых пределах правая часть (10.1) меньше $n^2 - n$. Тогда $T(n) \le dn + cn^2 - cn$ согласно рекурсивному правилу из определения $T(n)$. Выбирая $c \ge d$, можно сделать вывод, что неравенство $T(n) \le cn^2$ справедливо для n, и завершить индукцию. Таким образом, конструкция F из E занимает время $O(n^2)$. □

Пример 10.14. Покажем, как конструкция теоремы 10.13 применяется к простой формуле $E = x\,\overline{y} + \overline{x}\,(y+z)$. Разбор данной формулы представлен на рис. 10.7. К каждому узлу приписано выражение в КНФ, которое построено по выражению, представленному этим узлом.

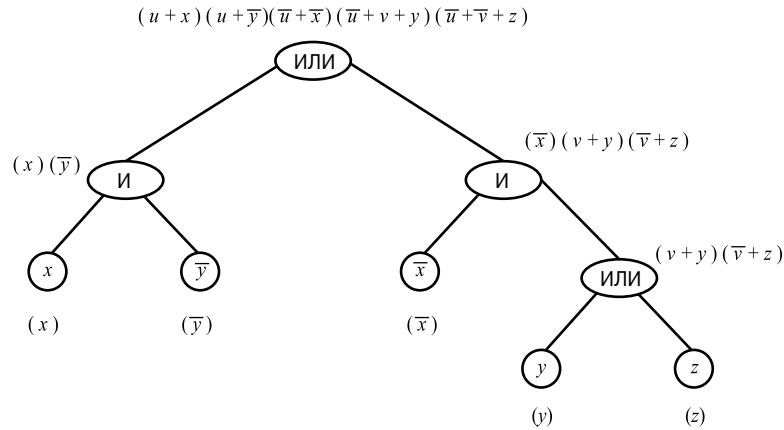


Рис. 10.7. Приведение булевой формулы к КНФ

Листья соответствуют литералам, и КНФ для каждого литерала — это дизъюнкт, состоящий из одного такого литерала. Например, лист с меткой $\overline{y}$ связан с КНФ $(\overline{y})$. Скобки тут не обязательны, но мы ставим их в КНФ, подчеркивая, что речь идет о произведении дизъюнктов.

Для узла с меткой **И** соответствующая КНФ получается как произведение (**И**) всех дизъюнктов двух подформул. Поэтому, например, с узлом, соответствующим выражению $\overline{x}\,(y+z)$, связана КНФ в виде произведения одного дизъюнкта $(\overline{x})$, соответствующего $\overline{x}$, и двух дизъюнктов $(v+y)(\overline{v}+z)$, соответствующих $y+z$.[5]

[5] В данном конкретном случае, когда подформула $y + z$ уже является дизъюнктом, можно не выполнять общее построение дизъюнкта для логического **ИЛИ** формул, а взять $(y + z)$ в качестве произведения дизъюнктов, эквивалентного $y + z$. Однако здесь мы строго придерживаемся общих правил.

Для узла с меткой **ИЛИ** нужно ввести новую переменную. Добавляем ее во все дизъюнкты левого операнда и ее отрицание во все дизъюнкты правого операнда. Рассмотрим, например, корневой узел на рис. 10.7. Он представляет собой логическое **ИЛИ** формул $x\,\bar{y}$ и $\bar{x}\,(y + z)$, для которых соответствующие КНФ были определены как $(x)(\bar{y})$ и $(\bar{x})(v + y)(\bar{v} + z)$, соответственно. Вводим новую переменную u, добавляя ее в дизъюнкты первой группы, а ее отрицание — в дизъюнкты второй группы. В результате получаем формулу

$$F = (u + x)(u + \bar{y})(\bar{u} + \bar{x})(\bar{u} + v + y)(\bar{u} + \bar{v} + z)$$

В теореме 10.13 говорится, что всякую подстановку T, удовлетворяющую E, можно расширить до подстановки S, удовлетворяющей F. Например, E истинна при подстановке $T(x) = 0$, $T(y) = 1$ и $T(z) = 1$. Можно расширить T до подстановки S, добавляя значения $S(u) = 1$ и $S(v) = 0$ к требуемым $S(x) = 0$, $S(y) = 1$ и $S(z) = 1$, которые берутся из T. Нетрудно убедиться, что F истинна при S.

Заметим, что, выбирая S, мы были вынуждены выбрать $S(u) = 1$, так как подстановка T делает истинной только вторую часть E — $\bar{x}\,(y + z)$. Поэтому для того, чтобы истинными были дизъюнкты $(u + x)(u + \bar{y})$ из первой части E, необходимо $S(u) = 1$. Но значение для v можно выбрать любым, так как в соответствии с T в подформуле $y + z$ истинны оба операнда логического **ИЛИ**. □

10.3.4. NP-полнота проблемы 3-выполнимости

Покажем, что проблема выполнимости NP-полна даже для более узкого класса булевых формул. Напомним, что проблема ЗВЫП (3-выполнимости) состоит в следующем.

- Дана булева формула E, представляющая собой произведение дизъюнктов, каждый из которых есть сумма трех различных литералов. Выполнима ли E?

Несмотря на то что формулы вида 3-КНФ — лишь небольшая часть КНФ-формул, их сложности достаточно, чтобы проверка их выполнимости была NP-полной проблемой. Это показывает следующая теорема.

Теорема 10.15. Проблема ЗВЫП NP-полна.

Доказательство. Очевидно, что ЗВЫП принадлежит $\mathcal{NP}$, поскольку ВЫП принадлежит $\mathcal{NP}$. Для доказательства NP-полноты сведем ВКНФ к ЗВЫП следующим образом. В данной КНФ $E = e_1 \wedge e_2 \wedge \cdots \wedge e_k$ каждый дизъюнкт e_i заменяется, как описано ниже, и создается новая формула F. Время, необходимое для построения F, линейно зависит от длины E, и, как мы увидим, подстановка удовлетворяет формуле E тогда и только тогда, когда ее можно расширить до подстановки, удовлетворяющей F.

1. Если e_i — одиночный литерал, скажем, (x)[6], то вводятся две новые переменные u и v, и (x) заменяется произведением четырех дизъюнктов

   $$(x + u + v)(x + u + \bar{v})(x + \bar{u} + v)(x + \bar{u} + \bar{v}).$$

   Поскольку здесь присутствуют все возможные комбинации из u и v, то все четыре дизъюнкта истинны только тогда, когда x истинна. Таким образом, все подстановки, удовлетворяющие E, и только они, могут быть расширены до подстановки, удовлетворяющей F.

2. Предположим, что e_i есть сумма двух литералов — $(x + y)$. Вводится новая переменная z, и e_i заменяется произведением двух дизъюнктов $(x + y + z)(x + y + \bar{z})$. Как и в случае 1, оба дизъюнкта истинны одновременно только тогда, когда истинна $(x + y)$.

3. Если дизъюнкт e_i есть сумма трех литералов, то он уже имеет вид, требуемый 3-КНФ, и остается в создаваемой формуле F.

4. Предположим, что $e_i = (x_1 + x_2 + \cdots + x_m)$ при некотором $m \geq 4$. Вводятся новые переменные $y_1, y_2, \ldots, y_{m-3}$, и e_i заменяется произведением дизъюнктов

   $$(x_1 + x_2 + y_1)(x_3 + \bar{y}_1 + y_2)(x_4 + \bar{y}_2 + y_3)\cdots$$
   $$(x_{m-2} + \bar{y}_{m-4} + y_{m-3})(x_{m-1} + x_m + \bar{y}_{m-3}). \qquad (10.2)$$

   Подстановка T, удовлетворяющая E, должна делать истинным хотя бы один из литералов в e_i, скажем x_j (напомним, что x_j может быть либо переменной, либо ее отрицанием). Тогда, если придать переменным $y_1, y_2, \ldots, y_{j-1}$ значение “ложь”, а $y_j, y_{j+1}, \ldots, y_{m-3}$ — “истина”, то все дизъюнкты в (10.2) будут истинными. Таким образом, T можно расширить до подстановки, удовлетворяющей всем этим дизъюнктам. Наоборот, если при подстановке T все x имеют значение “ложь”, то T невозможно расширить так, чтобы формула (10.2) была истинной. Причина в том, что дизъюнктов $m - 2$, а каждый y, которых всего $m - 3$, может сделать истинным лишь один дизъюнкт, независимо от того, имеет ли он значение “истина” или “ложь”.

Таким образом, мы показали, как свести любой экземпляр E проблемы ВКНФ к экземпляру F проблемы ЗВЫП, выполнимому тогда и только тогда, когда выполнима формула E. Построение, очевидно, требует времени, которое линейно зависит от длины E, так как ни в одном из рассмотренных выше четырех случаев длина дизъюнкта не увеличивается более, чем в 32/3 раза (соотношение числа символов в случае 1). Кроме того, символы, необходимые для построения формулы F, легко найти за время, пропорциональное числу этих символов. Поскольку проблема ВКНФ NP-полна, то и проблема ЗВЫП также NP-полна. □

---

[6] Для удобства, говоря о литералах, считаем их переменными без отрицания, например x. Но все наши построения остаются в силе и в том случае, когда некоторые или все литералы являются отрицаниями переменных, например $\bar{x}$.

10.3.5. Упражнения к разделу 10.3

10.3.1. Приведите следующие формулы к 3-КНФ:

а) (*) $xy + \bar{x}\,z$;

б) $wxyz + u + v$;

в) $wxy + \bar{x}\,uv$.

10.3.2. Проблема 4П-ВЫП определяется следующим образом: по данной булевой формуле E выяснить, есть ли у нее хотя бы четыре удовлетворяющие подстановки. Покажите, что проблема 4П-ВЫП NP-полна.

10.3.3. В этом упражнении определяется семейство 3-КНФ-формул. Формула E_n имеет n переменных $x_1, x_2, \ldots, x_n$. Для всякого множества из трех различных целых чисел от 1 до n формула E_n содержит дизъюнкты $(x_1 + x_2 + x_3)$ и $(\bar{x}_1 + \bar{x}_2 + \bar{x}_3)$. Выполнима ли E_n для:

а) (*!) $n = 4$?

б) (!!) $n = 5$?

10.3.4. (!) Постройте алгоритм с полиномиальным временем работы для решения проблемы 2ВЫП (2-выполнимости), т.е. проблемы выполнимости булевой формулы в КНФ, каждый дизъюнкт которой содержит ровно два литерала. *Указание*. Если один из двух литералов в дизъюнкте ложен, то второй обязательно истинен. Сделайте вначале предположение относительно истинности одной из переменных, а затем постарайтесь извлечь все возможные следствия для оставшихся переменных.

10.4. Еще несколько NP-полных проблем

Приведем несколько примеров того, как с помощью одной NP-полной проблемы можно доказать NP-полноту целого ряда других. Этот процесс получения новых NP-полных проблем имеет два важных следствия.

- Обнаружив новую NP-полную проблему, мы не получаем практически никаких шансов на отыскание эффективного алгоритма ее решения. Мы стремимся найти эвристики, частные решения, аппроксимации, применяем какие-либо иные методы, лишь бы избежать решения “в лоб”. Более того, мы можем делать все это, будучи уверенными, что не просто “не замечаем метод”.
- Всякое добавление новой проблемы P в список NP-полных проблем дает еще одно подтверждение гипотезы, что *все* они требуют экспоненциального времени. Усилия, затраченные на поиски полиномиального алгоритма решения проблемы P, мы, сами того не зная, посвятили обоснованию равенства $\mathcal{P} = \mathcal{NP}$.

Именно безуспешные попытки многих первоклассных математиков и других ученых доказать нечто эквивалентное утверждению $\mathcal{P} = \mathcal{NP}$ в конечном счете убеждают в том, что это равенство невероятно и, наоборот, *все* NP-полные проблемы требуют экспоненциального времени.

В этом разделе мы встретим несколько NP-полных проблем, связанных с графами. Эти проблемы чаще других используются при решении практических вопросов. Мы поговорим о проблеме коммивояжера (ПКОМ), с которой уже встречались в разделе 10.1.4. Покажем, что более простая, но также важная версия этой проблемы, называемая проблемой гамильтонова цикла (ГЦ), NP-полна. Тем самым покажем, что NP-полной является и более общая проблема ПКОМ. Представим несколько проблем, касающихся "покрытия" графов, таких, как "проблема узельного покрытия". В ней требуется отыскать наименьшее множество узлов, "покрывающих" все ребра так, чтобы хотя бы один конец каждого ребра принадлежал выбранному множеству.

10.4.1. Описание NP-полных проблем

Вводя новую NP-полную проблему, будем использовать следующую стилизованную форму ее определения.

1. *Название* проблемы и, как правило, его аббревиатура, например, ВЫП или ПКОМ.
2. *Вход* проблемы: что и каким образом представляют входные данные.
3. Искомый *выход*: при каких условиях выходом будет "да".
4. Проблема, сведение которой к данной доказывает NP-полноту последней.

Пример 10.16. Вот как могут выглядеть описание проблемы 3-выполнимости и доказательство ее NP-полноты.

Проблема. Выполнимость формул, находящихся в 3-КНФ (ЗВЫП).

Вход. Булева формула в 3-КНФ.

Выход. Ответ "да" тогда и только тогда, когда формула выполнима.

Проблема, сводящаяся к данной. ВКНФ. □

10.4.2. Проблема независимого множества

Пусть G — неориентированный граф. Подмножество I узлов G называется *независимым множеством*, если никакие два узла из I не соединены между собой ребром из G. Независимое множество является *максимальным*, если оно не меньше (содержит не меньше узлов), чем любое независимое множество этого графа.

Пример 10.17. Для графа, изображенного на рис. 10.1 (см. раздел 10.1.2), максимальным независимым множеством является {1, 4}. Это единственное множество размера два, которое является независимым, поскольку для любых других двух узлов существует соединяющее их ребро. Таким образом, никакое множество размера три и более не является независимым, например, множество {1, 2, 4} (есть ребро между узлами 1 и 2). Итак,

{1, 4} — максимальное независимое множество, причем единственное для данного графа, хотя граф может иметь много максимальных независимых множеств. Например, множество {1} также независимо для данного графа, но не максимально. □

В комбинаторной оптимизации проблема максимального независимого множества обычно формулируется так: для данного графа найти максимальное независимое множество. Но, как и любую проблему в теории сложности, мы должны сформулировать ее в терминах ответа "да" или "нет". Поэтому в формулировку проблемы нам придется ввести нижнюю границу и сформулировать проблему как вопрос о том, существует ли для данного графа независимое множество размера, не меньшего этой нижней границы. Формальное определение проблемы максимального независимого множества имеет следующий вид.

Проблема. Независимое множество (НМ).

Вход. Граф G и нижняя граница k, значение которой заключено между 1 и числом узлов G.

Выход. Ответ "да" тогда и только тогда, когда G имеет независимое множество из k узлов.

Проблема, сводящаяся к данной. Проблема ЗВЫП.

Мы должны, как и пообещали, доказать NP-полноту проблемы НМ, сведя к ней проблему ЗВЫП.

Теорема 10.18. Проблема независимого множества NP-полна.

Доказательство. Легко видеть, что проблема НМ принадлежит классу $\mathcal{NP}$. Для данного графа G нужно угадать набор из k его узлов и проверить их независимость.

Теперь опишем сведение ЗВЫП к НМ. Пусть $E = (e_1)(e_2) \ldots (e_m)$ — формула в 3-КНФ. По E строится граф G, содержащий $3m$ узлов, которые обозначим как $[i, j]$, где $1 \le i \le m$, а $j = 1$, 2 или 3. Узел $[i, j]$ представляет j-й литерал в дизъюнкте e_i. На рис.10.8 приведен пример графа G, построенного по 3-КНФ

$$(x_1 + x_2 + x_3)(\bar{x}_1 + x_2 + x_4)(\bar{x}_2 + x_3 + x_5)(\bar{x}_3 + \bar{x}_4 + \bar{x}_5).$$

Дизъюнкты формулы представлены столбцами. Поясним вкратце, почему узлы имеют именно такой вид.

"Фокус" построения G состоит в том, чтобы каждое независимое множество из m узлов представляло один из способов выполнить формулу E. Ключевых идей тут две.

1. Мы хотим гарантировать, что можно выбрать только один узел, соответствующий данному дизъюнкту. Для этого помещаем ребра между каждой парой узлов из одного столбца, т.е. создаем ребра ($[i, 1]$, $[i, 2]$), ($[i, 1]$, $[i, 3]$) и ($[i, 2]$, $[i, 3]$) для всех i (см. рис. 10.8).
2. Нам необходимо предотвратить выбор в качестве элементов независимого множества таких узлов, которые представляют литералы, дополняющие друг друга. Поэтому, если есть два узла $[i_1, j_1]$ и $[i_2, j_2]$, один из которых представляет переменную x, а

второй — $\overline{x}$, то соединяем их ребром. Таким образом, их нельзя одновременно выбрать в качестве элементов независимого множества.

Граница k для графа G, построенного по этим двум правилам, равняется m.

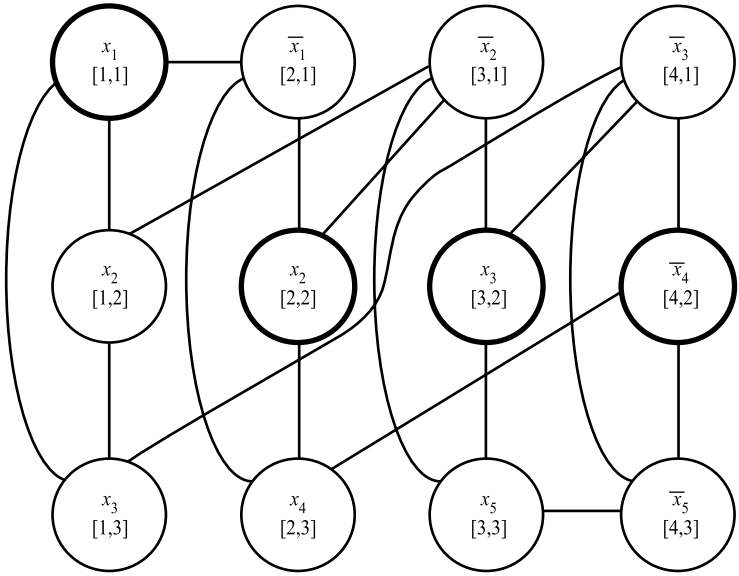


Рис. 10.8. Построение независимого множества по выполнимой булевой формуле, находящейся в 3-КНФ

Легче ли проблемы, требующие ответа “да” или “нет”?

Может показаться, что “да/нет”-версия проблемы легче, чем исходная проблема оптимизации. Например, найти наибольшее независимое множество сложно, в то время как для данного небольшого значения границы k легко проверить, что существует независимое множество размера k. Однако, *возможно*, нам дана константа k, представляющая собой максимальный размер, для которого существует независимое множество. Тогда для решения “да/нет”-версии проблемы необходимо найти максимальное независимое множество.

В действительности, все обычные NP-полные проблемы оптимизации и их “да/нет”-версии имеют эквивалентную сложность, по крайней мере, в пределах полиномиальной зависимости. Например, если бы у нас был полиномиальный алгоритм, позволяющий *найти* максимальное независимое множество, то можно было бы решить “да/нет”-проблему, найдя максимальное независимое множество и проверив, что его размер не меньше предельного значения k. Поскольку, как мы покажем, “да/нет”-версия NP-полна, то и версия оптимизации должна быть трудно разрешимой.

Сравнение можно провести и по-другому. Предположим, что у нас есть полиномиальный алгоритм решения "да/нет"-проблемы НМ. Если граф имеет n узлов, то размер максимального независимого множества заключен между 1 и n. Прогоняя алгоритм решения НМ для всех возможных значений границы от 1 до n, мы, безусловно, найдем размер максимального независимого множества (но не обязательно само это множество) за время, необходимое для однократного решения НМ и умноженное на n. В действительности, с использованием бинарного поиска нам будет достаточно времени однократного решения, умноженного всего лишь на $\log_2 n$.

Граф G и границу k можно построить по формуле E за время, пропорциональное квадрату длины E, поэтому преобразование E в G представляет собой полиномиальное сведение. Мы должны показать, что оно корректно сводит ЗВЫП к проблеме НМ, т.е.

- E выполнима тогда и только тогда, когда G имеет независимое множество размера m.

(*Достаточность*) Во-первых, заметим, что независимое множество не может содержать два узла из одного и того же дизъюнкта, $[i, j_1]$ и $[i, j_2]$, где $j_1 \neq j_2$, поскольку все такие узлы попарно соединены ребрами, как в столбцах на рис. 10.8. Поэтому, если существует независимое множество размера m, то оно содержит только по одному узлу из каждого дизъюнкта.

Более того, независимое множество не может одновременно содержать узлы, которые соответствуют некоторой переменной x и ее отрицанию $\bar{x}$, поскольку такие узлы попарно соединены ребрами. Таким образом, независимое множество I размера m приводит к следующей подстановке T, удовлетворяющей формуле E. Если множеству I принадлежит узел, соответствующий переменной x, то $T(x) = 1$, а если узел, соответствующий отрицанию $\bar{x}$, то $T(x) = 0$. Если I не содержит узла, который соответствовал бы x или $\bar{x}$, то значение $T(x)$ выбирается произвольно. Отметим, что пункт 2 приведенных выше правил объясняет, почему невозможно противоречие, когда узлы, соответствующие x и $\bar{x}$, одновременно принадлежат I.

Утверждаем, что E истинна при T. Для каждого дизъюнкта формулы E существует узел в I, соответствующий одному из ее литералов, и подстановка T выбрана так, что для нее этот литерал имеет значение "истина". Поэтому, если независимое множество размера m существует, то E выполнима.

(*Необходимость*) Допустим теперь, что E истинна при некоторой подстановке T. Поскольку при T каждый дизъюнкт E имеет значение "истина", можно выбрать из каждого дизъюнкта по одному литералу, истинному при подстановке T. В некоторых дизъюнктах таких литералов может быть два или три, и тогда один из них выбирается произвольно. Строим множество I, состоящее из m узлов, соответствующих литералам, которые были выбраны из каждого дизъюнкта.

Утверждаем, что I является независимым множеством. Ребра между узлами из одного и того же дизъюнкта (столбцы на рис. 10.8) не могут иметь оба конца в I, так как из каждого дизъюнкта выбирается только по одному узлу. Оба конца ребра, соединяющего переменную и ее отрицание, также не могут одновременно находиться в I, так как в I выбраны только узлы, которые соответствуют литералам, истинным при подстановке T. Безусловно, для T либо x, либо $\bar{x}$ будет истинным, но не одновременно. Отсюда следует, что если E выполнима, то G имеет независимое множество размера m.

Таким образом, существует полиномиальное сведение проблемы ЗВЫП к проблеме НМ. Поскольку известно, что проблема ЗВЫП NP-полна, то согласно теореме 10.5 проблема НМ также NP-полна. □

Пример 10.19. Посмотрим, как конструкция теоремы 10.18 применяется к формуле

$$E = (x_1 + x_2 + x_3)(\bar{x}_1 + x_2 + x_4)(\bar{x}_2 + x_3 + x_5)(\bar{x}_3 + \bar{x}_4 + \bar{x}_5).$$

Мы уже видели граф, полученный по данной формуле (см. рис. 10.8). Узлы расположены в четырех столбцах, соответствующих четырем дизъюнктам. Кроме обозначений узлов (пары целых чисел), указаны также соответствующие им литералы. Отметим, что все узлы одного столбца, соответствующие литералам из одного дизъюнкта, попарно соединены ребрами. Кроме того, ребрами соединены узлы, соответствующие переменной и ее отрицанию. Так, узел [3, 1], соответствующий $\bar{x}_2$, соединен с узлами [1, 2] и [2, 2], соответствующими вхождениям переменной x_2.

Жирными кружками выделено множество I из четырех узлов, по одному из каждого столбца. Они формируют независимое множество. Поскольку им соответствуют четыре литерала x_1, x_2, x_3 и $\bar{x}_4$, то по ним можно построить подстановку T, в которой $T(x_1) = 1$, $T(x_2) = 1$, $T(x_3) = 1$ и $T(x_4) = 0$. Нужно также приписать значение переменной x_5, но его можно выбрать произвольным образом, скажем, $T(x_5) = 0$. Итак, формула E истинна при T, и множество узлов I указывает по одному литералу, истинному при T, в каждом дизъюнкте. □

Для чего используются независимые множества

В цели данной книги не входит описание приложений тех проблем, NP-полнота которых доказывается. Однако проблемы, рассмотренные в разделе 10.4, взяты из фундаментальной статьи Р. Карпа об NP-полноте, в которой он описал наиболее важные проблемы в области исследования операций и показал, что многие из них являются NP-полными. Таким образом, существует великое множество “реальных” проблем, решаемых с помощью абстрактных.

В качестве примера рассмотрим, как с помощью алгоритма поиска максимального независимого множества можно составить расписания экзаменов. Пусть узлы графа соответствуют различным предметам, и два узла соединяются ребром, если один или несколько студентов изучают оба эти предмета, и экзамены по этим предметам не мо-

гут быть назначены на одно и то же время. Если мы найдем максимальное независимое множество, то сможем составить расписание так, чтобы экзамены по всем предметам из этого множества проходили одновременно, и ни у одного студента не было накладок.

10.4.3. Проблема узельного покрытия

Еще один важный класс проблем комбинаторной оптимизации касается "покрытия" графа. К примеру, *реберное покрытие* есть множество ребер, для которого каждый узел графа является концом хотя бы одного ребра из этого множества. Реберное покрытие является *минимальным*, если оно содержит не больше ребер, чем любое реберное покрытие этого графа. Проблема, состоящая в выяснении того, имеет ли граф реберное покрытие из k ребер, здесь не рассматривается.

Докажем NP-полноту проблемы узельного покрытия. *Узельное покрытие* графа — это множество узлов, для которого хотя бы один конец любого ребра является узлом из этого множества. Узельное покрытие является *минимальным*, если оно содержит не больше узлов, чем любое узельное покрытие данного графа.

Узельные покрытия и независимые множества тесно связаны, поскольку дополнение независимого множества является узельным покрытием, и наоборот. Поэтому проблему НМ легко свести к проблеме узельного покрытия (УП), сформулировав последнюю должным образом в виде "да/нет"-проблемы.

Проблема. Проблема узельного покрытия (УП).

Вход. Граф G и верхняя граница k, значение которой заключено между 0 и числом, которое на 1 меньше числа узлов G.

Выход. Ответ "да" тогда и только тогда, когда G имеет узельное покрытие из k или меньшего числа узлов.

Проблема, сводящаяся к данной. Проблема независимого множества.

Теорема 10.20. Проблема узельного покрытия NP-полна.

Доказательство. Проблема УП, очевидно, принадлежит $\mathcal{NP}$. Нужно угадать множество из k узлов, и проверить для каждого ребра G, принадлежит ли хотя бы один его конец этому множеству.

Для завершения доказательства сведем проблему НМ к проблеме УП. Идея, которую подсказывает рис. 10.8, состоит в том, что дополнение независимого множества есть узельное покрытие. К примеру, узлы на рис. 10.8, которые *не выделены* жирным, образуют узельное покрытие. Поскольку выделенные узлы образуют максимальное независимое множество, то минимальное узельное покрытие образовано остальными узлами.

Сведение заключается в следующем. Пусть граф G с нижней границей k — экземпляр проблемы независимого множества. Если G имеет n узлов, то пусть G с верхней грани-

цей $n - k$ — тот экземпляр проблемы узельного покрытия, который мы строим. Это преобразование, очевидно, может быть произведено за линейное время. Утверждаем, что

- G имеет независимое множество размера k тогда и только тогда, когда в G есть узельное покрытие из $n - k$ узлов.

(*Достаточность*) Пусть N — множество узлов графа G, и пусть C — его узельное покрытие размера $n - k$. Утверждаем, что $N - C$ есть независимое множество. Предположим противное, т.е. что в $N - C$ существует пара узлов v и w, соединенная ребром в G. Тогда, поскольку ни v, ни w не принадлежат C, ребро (v, w), принадлежащее G, не покрывается предполагаемым узельным покрытием C. Таким образом, от противного доказано, что $N - C$ является независимым множеством. Очевидно, это множество содержит k узлов, и в эту сторону утверждение доказано.

(*Необходимость*) Предположим, I — независимое множество, состоящее из k узлов. Утверждаем, что $N - I$ — узельное покрытие графа G, состоящее из $n - k$ узлов. Снова используем доказательство от противного. Если существует некоторое ребро (v, w), не покрываемое множеством $N - I$, то и v, и w принадлежат I, но соединены ребром, а это противоречит определению независимого множества. □

10.4.4. Проблема ориентированного гамильтонова цикла

Мы хотим показать NP-полноту проблемы коммивояжера (ПКОМ), так как она представляет большой интерес с точки зрения комбинаторики. Наиболее известное доказательство ее NP-полноты в действительности является доказательством NP-полноты более простой проблемы, называемой "проблемой гамильтонова цикла" (ГЦ). *Проблема гамильтонова цикла* описывается следующим образом.

Проблема. Проблема гамильтонова цикла.

Вход. Неориентированный граф G.

Выход. Ответ "да" тогда и только тогда, когда G имеет гамильтонов цикл, т.е. цикл, проходящий через каждый узел G только один раз.

Отметим, что проблема ГЦ — это частный случай проблемы ПКОМ, при котором вес каждого ребра имеет значение 1. Поэтому полиномиальное сведение ГЦ к ПКОМ устроено очень просто: нужно всего лишь уточнить, что каждое ребро графа имеет единичный вес.

Доказать NP-полноту проблемы ГЦ достаточно трудно. Вначале рассмотрим ограниченную версию проблемы ГЦ, в которой ребра имеют направления (т.е. являются направленными ребрами, или дугами), а гамильтонов цикл обходит граф в направлении, указываемом дугами. Сведем проблему ЗВЫП к этой ограниченной версии проблемы ГЦ, а затем сведем последнюю к обычной "неориентированной" версии ГЦ.

Проблема. Проблема ориентированного гамильтонова цикла (ОГЦ).

Вход. Ориентированный граф G.

Выход. Ответ “да” тогда и только тогда, когда в G есть ориентированный цикл, проходящий через каждый узел ровно один раз.

Проблема, сводящаяся к данной. Проблема ЗВЫП.

Теорема 10.21. Проблема ориентированного гамильтонова цикла NP-полна.

Доказательство. Доказать, что ОГЦ принадлежит классу $\mathcal{NP}$, легко — нужно угадать цикл и проверить наличие его дуг в данном графе. Сведем проблему ЗВЫП к ОГЦ. Для этого потребуется построить довольно сложный граф, в котором подграфы специального вида представляют переменные и дизъюнкты данного экземпляра проблемы ЗВЫП.

Начнем построение экземпляра ОГЦ по булевой формуле в ЗКНФ. Пусть формула имеет вид $E = e_1 \wedge e_2 \wedge \cdots \wedge e_k$, где каждое e_i — дизъюнкт, представляющий собой сумму трех литералов, скажем, $e_i = (\alpha_{i1} + \alpha_{i2} + \alpha_{i3})$. Пусть $x_1, x_2, \ldots, x_n$ — переменные в формуле E. Для всех дизъюнктов и переменных строятся подграфы, как показано на рис. 10.9.

Для каждой переменной x_i строится подграф H_i, структура которого показана на рис. 10.9, *а*. Здесь m_i — большее из чисел вхождений x_i и $\bar{x}_i$ в E. Узлы c_{ij} и b_{ij}, расположенные в двух столбцах, соединены между собой дугами в обоих направлениях. Кроме того, каждое b имеет дугу, ведущую в c, расположенное на ступеньку ниже, т.е., если $j < m_i$, то b_{ij} имеет дугу, ведущую в $c_{i,j+1}$. Аналогично, если $j < m_i$, то c_{ij} имеет дугу, ведущую в $b_{i,j+1}$. Наконец, есть верхний узел a_i, из которого дуги ведут в b_{i0} и c_{i0}, и нижний узел d_i, в который ведут дуги из b_{im_i} и c_{im_i}.

На рис. 10.9, *б* показана структура графа в целом. Каждый шестиугольник представляет один подграф, построенный для переменной (его структура показана на рис. 10.9, *а*). Шестиугольники расположены циклически, и из нижнего узла каждого подграфа дуга ведет в верхний узел следующего.

Допустим, граф на рис. 10.9, *б* имеет ориентированный гамильтонов цикл. Не ограничивая общности, можно считать, что этот цикл начинается в a_1. Если затем он переходит в b_{10}, то на следующем шаге он обязательно перейдет в c_{10} (иначе c_{10} не появится в цикле). В самом деле, если цикл переходит из a_1 в b_{10}, а затем — в c_{11}, то c_{10} никогда не появится в цикле, поскольку оба его предшественника (a_0 и b_{10}) уже содержатся в нем.

Таким образом, если начало цикла имеет вид a_1, b_{10}, то далее он должен спускаться “лесенкой”, переходя из стороны в сторону:

$$a_1, b_{10}, c_{10}, b_{11}, c_{11}, \ldots, b_{1m_1}, c_{1m_1}, d_1.$$

Если начало цикла имеет вид a_1, c_{10}, то в лесенке меняется порядок предшествования c и b:

$$a_1, c_{10}, b_{10}, c_{11}, b_{11}, \ldots, c_{1m_1}, b_{1m_1}, d_1.$$

Решающим пунктом в доказательстве является то, что порядок, при котором спуск совершается от c к b, можно трактовать как приписывание переменной, соответствующей данному подграфу, значения “истина”, а порядок, при котором спуск совершается от b к c, соответствует приписыванию этой переменной значения “ложь”.

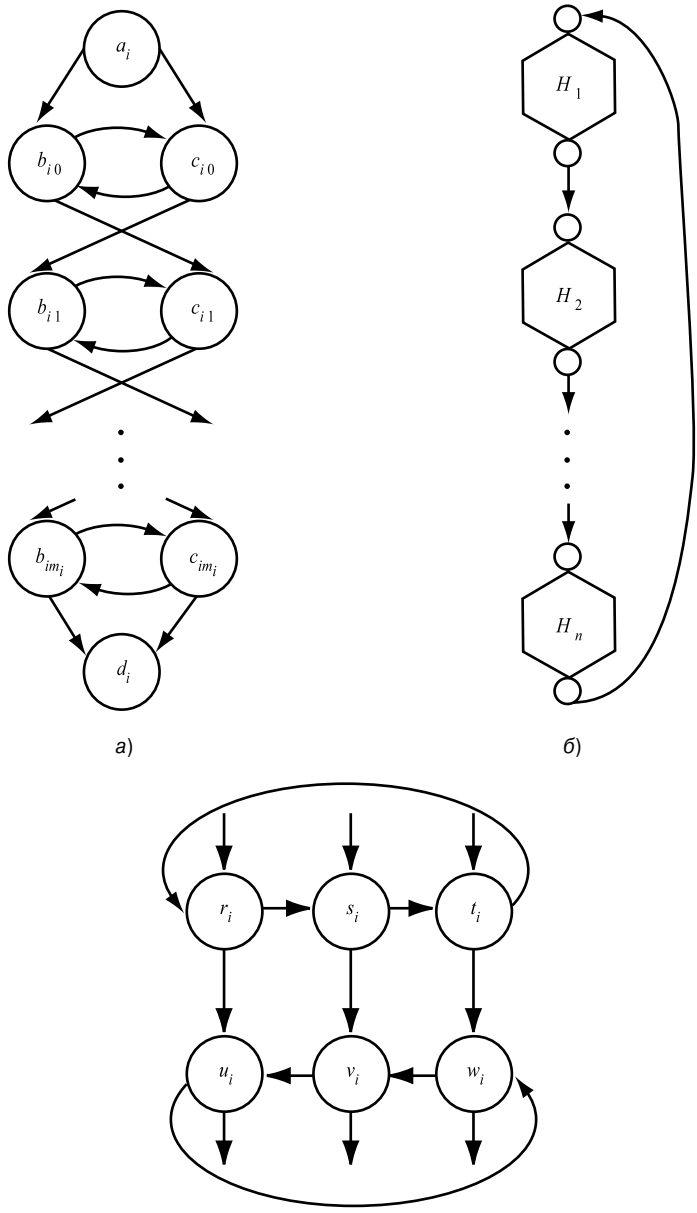


Рис. 10.9. Построение, используемое в доказательстве NP-полноты проблемы гамильтонова цикла

Закончив обход подграфа H_1, цикл должен перейти в a_2, где снова возникает выбор следующего перехода — в b_{20} или в c_{20}. Однако в силу тех же аргументов, которые приведены для H_1, после того, как сделан выбор направления вправо или влево от a_2, путь

обхода H_2 уже зафиксирован. Вообще, при входе в каждый H_i есть выбор перехода влево или вправо, но никакого другого. Иначе некоторый узел обречен быть *недоступным*, т.е. он не сможет появиться в ориентированном гамильтоновом цикле, поскольку все его предшественники появились в нем ранее.

В дальнейшем это позволит нам считать, что выбор перехода из a_i в b_{i0} означает приписывание переменной x_i значения "истина", а перехода из a_i в c_{i0} — значения "ложь". Поэтому граф на рис. 10.9, *б* имеет 2^n ориентированных гамильтоновых циклов, соответствующих 2^n возможным подстановкам для n переменных.

Однако на рис. 10.9, *б* изображен лишь скелет графа, порождаемого по формуле E, находящейся в 3-КНФ. Каждому дизъюнкту e_i ставится в соответствие подграф I_j (рис. 10.9, *в*). Он обладает тем свойством, что если цикл входит в r_j, то должен выходить из u_j, а если входит в s_j, то выходит из v_j, и если входит в t_j, то выходит из w_j. Действительно, если, достигнув I_j, цикл выходит из узла, расположенного не под входным узлом, то один или несколько узлов оказываются недоступными — они никак не могут появиться в цикле. В силу симметрии достаточно рассмотреть ситуацию, когда r_j — первый узел из I_j в цикле. Возможны три варианта.

1. Следующие два узла в цикле — s_j и t_j. Если затем цикл переходит в w_j и выходит, то узел v_j оказывается недоступным. Если цикл переходит в w_j и v_j, а затем выходит, то u_j оказывается недоступным. Таким образом, цикл должен выходить из u_j, обойдя предварительно все шесть узлов данного подграфа.
2. Следующие после r_j узлы — s_j и v_j. Если затем цикл переходит в u_j, то узел w_j оказывается недоступным. Если после u_j цикл переходит в w_j, то t_j никогда не попадет в цикл по причине, "обратной" причине недоступности. В этой ситуации в t_j можно попасть извне, но если t_j войдет в цикл позже, то цикл нельзя будет продолжить, так как оба потомка t_j уже появлялись в цикле ранее. Таким образом, и в этом случае цикл выходит из u_j. Отметим, однако, что t_j и w_j непроходимы слева; они должны появиться в цикле позже, что возможно.
3. Цикл переходит из r_j прямо в u_j. Если цикл переходит затем в w_j, то t_j не сможет появиться в цикле, поскольку оба его потомка уже там есть, о чем говорилось при варианте 2. Таким образом, цикл должен сразу выходить из u_j. Оставшиеся четыре узла должны войти в цикл позже.

В завершение построения графа G для формулы E соединяем подграфы I и H следующим образом. Допустим, у дизъюнкта e_i первым литералом является x_i, переменная без отрицания. Выберем некоторый узел c_{ip}, где p от 0 до $m_i - 1$, ранее не использованный для соединения с подграфами I. Введем дуги, ведущие из c_{ip} в r_j и из u_j в $b_{i,p+1}$. Если же первым литералом дизъюнкта e_j является отрицание $\bar{x}_i$, то нужно отыскать неиспользованный узел b_{ip}, а затем соединить b_{ip} с r_j и u_j с $c_{i,p+1}$.

Для второго и третьего литералов e_j граф дополняется точно так же, за одним исключением. Для второго литерала используются узлы s_j и v_j, а для третьего — t_j и w_j. Таким образом, каждый I_j имеет три соединения с подграфами типа H, которые представляют переменные, присутствующие в дизъюнкте e_j. Если литерал не содержит отрицания, то соединение выходит из c-узла и входит в b-узел, расположенный ниже, а если содержит — соединение выходит из b-узла, возвращаясь в расположенный ниже c-узел. Мы утверждаем, что

- построенный таким образом граф G имеет ориентированный гамильтонов цикл тогда и только тогда, когда формула E выполнима.

(*Достаточность*) Предположим, существует подстановка T, удовлетворяющая формуле E. Построим ориентированный гамильтонов цикл следующим образом.

1. Вначале выберем путь, обходящий только подграфы H (т.е. граф, изображенный на рис. 10.9, *б*) в соответствии с подстановкой T. Таким образом, если $T(x_i) = 1$, то цикл переходит из a_i в b_{i0}, а если $T(x_i) = 0$, то он переходит из a_i в c_{i0}.
2. Однако цикл, построенный к данному моменту, может содержать дугу из b_{ip} в $c_{i,p+1}$, причем у b_{ip} есть еще одна дуга в один из подграфов I_j, который пока не включен в цикл. Тогда к циклу добавляется "крюк", который начинается в b_{ip}, обходит все шесть узлов подграфа I_j и возвращается в $c_{i,p+1}$. Дуга $b_{ip} \to c_{i,p+1}$ исключается из цикла, но узлы на ее концах остаются в нем.
3. Аналогично, если в цикле есть дуга из c_{ip} в $b_{i,p+1}$, и у c_{ip} есть еще одна дуга в один из I_j, пока не включенных в цикл, то к циклу добавляется "крюк", проходящий через все шесть узлов I_j.

Тот факт, что T удовлетворяет формуле E, гарантирует, что исходный путь, построенный на шаге 1, будет содержать, по крайней мере, одну дугу, которая на шаге 2 или 3 позволит включить в цикл подграф I_j для каждого дизъюнкта e_i. Таким образом, цикл включает в себя все подграфы I_j и является ориентированным гамильтоновым.

(*Необходимость*) Предположим, что граф G имеет ориентированный гамильтонов цикл, и покажем, что формула E выполнима. Напомним два важных пункта из предыдущего анализа.

1. Если гамильтонов цикл входит в некоторый I_j в узле r_j, s_j или t_j, то он должен выходить из него в узле u_j, v_j или w_j, соответственно.
2. Таким образом, рассматривая данный гамильтонов цикл как обход подграфов типа H, (см. рис. 10.9, *б*), можно характеризовать "экскурсию", совершаемую в некоторое I_j, как переход цикла по дуге, "параллельной" одной из дуг $b_{ip} \to c_{i,p+1}$ или $c_{ip} \to b_{i,p+1}$.

Если игнорировать экскурсии в подграфы I_j, то гамильтонов цикл должен быть одним из 2^n циклов, которые возможны с использованием только подграфов H_i и соответствуют выборам переходов из a_i либо в b_{i0}, либо в c_{i0}. Каждый из этих выборов соответствует приписыванию значений переменным из E. Если один из них дает га-

мильтонов цикл, включающий подграфы I_j, то подстановка, соответствующая этому выбору, должна удовлетворять формуле E.

Причина в том, что если цикл переходит из a_i в b_{i0}, то экскурсия в I_j может быть совершена только тогда, когда j-й дизъюнкт содержит x_i в качестве одного из литералов. Если цикл переходит из a_i в c_{i0}, то экскурсия в I_j может быть совершена только тогда, когда $\bar{x}_i$ является литералом в j-м дизъюнкте. Таким образом, из того, что все подграфы I_j могут быть включены в граф, следует, что при данной подстановке хотя бы один из литералов в каждом дизъюнкте истинен, т.е. формула E выполнима. □

Пример 10.22. Приведем очень простой пример конструкции из теоремы 10.21, основанный на формуле в 3-КНФ $E = (x_1 + x_2 + x_3)(\bar{x}_1 + \bar{x}_2 + x_3)$. Граф, который при этом строится, изображен на рис. 10.10. Дуги, соединяющие подграфы H-типа с подграфами I-типа, показаны пунктиром исключительно для удобочитаемости, в остальном они ничем не отличаются от сплошных дуг.

Слева вверху виден подграф, соответствующий x_1. Поскольку x_1 встречается по одному разу в чистом виде и с отрицанием, достаточно одной ступеньки "лесенки". Поэтому здесь присутствуют две строки из узлов b и c. Слева внизу виден подграф, соответствующий x_3, которая дважды встречается в чистом виде и ни разу с отрицанием. Поэтому необходимы две различные дуги $c_{3p} \to b_{3,p+1}$, с помощью которых можно присоединить подграфы I_1 и I_2 для дизъюнктов e_1 и e_2, чтобы представить присутствие x_3 в этих дизъюнктах. Поэтому в подграфе для x_3 нужны три строки из узлов b и c.

Рассмотрим подграф I_2, соответствующий дизъюнкту $(\bar{x}_1 + \bar{x}_2 + x_3)$. Для первого литерала $\bar{x}_1$ к b_{10} присоединяется r_2, а к u_2 — c_{11}. Для второго литерала то же самое происходит с b_{20}, s_2, v_2 и c_{21}. Поскольку третий литерал не содержит отрицания, то он прикрепляется к узлам c и b, находящимся ниже, т.е. к c_{31} присоединяется t_2, и к w_2 — b_{32}.

Одна из нескольких удовлетворяющих подстановок имеет вид $x_1 = 1$, $x_2 = 0$ и $x_3 = 0$. При данной подстановке первый дизъюнкт истинен благодаря первому литералу x_1, а второй — благодаря второму литералу $\bar{x}_2$. Для этой подстановки можно построить гамильтонов цикл, в котором присутствуют дуги $a_1 \to b_{10}$, $a_2 \to c_{20}$ и $a_3 \to c_{30}$. Цикл покрывает первый дизъюнкт, совершая "крюк" из H_1 в I_1, т.е. использует дугу $c_{10} \to r_1$, обходит все узлы в I_1 и возвращается в b_{11}. Второй дизъюнкт покрывается "крюком" из H_2 в I_2, который начинается переходом по дуге $b_{20} \to s_2$, затем обходит все узлы I_2 и возвращается в c_{21}. Весь гамильтонов цикл выделен более жирными линиями (сплошными или пунктирными) и крупными стрелками (см. рис. 10.10). □

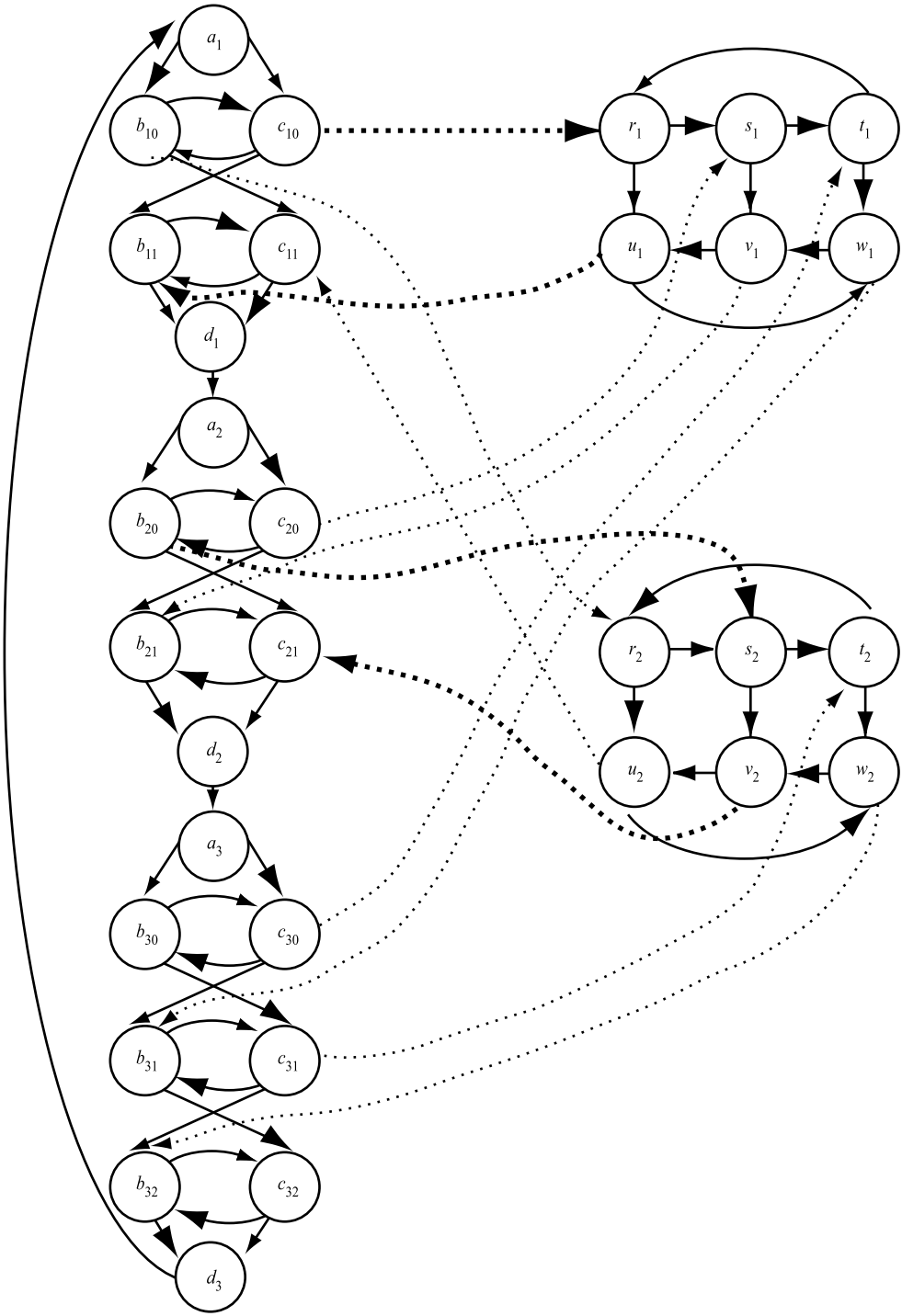


Рис. 10.10. Пример построения гамильтонова цикла

10.4.5. Неориентированные гамильтоновы циклы и ПКОМ

Доказательства NP-полноты проблем неориентированного гамильтонова цикла и коммивояжера относительно просты. В разделе 10.1.4 мы уже убедились, что ПКОМ принадлежит классу $\mathcal{NP}$. Проблема ГЦ представляет собой частный случай ПКОМ, так что она тоже принадлежит $\mathcal{NP}$. Сведем ОГЦ к ГЦ и ГЦ к ПКОМ.

Проблема. Проблема неориентированного гамильтонова цикла.

Вход. Неориентированный граф G.

Выход. Ответ “да” тогда и только тогда, когда G имеет гамильтонов цикл.

Проблема, сводящаяся к данной. ОГЦ.

Теорема 10.23. Проблема ГЦ NP-полна.

Доказательство. ОГЦ сводится к ГЦ следующим образом. Пусть дан ориентированный граф G_d. По нему строится неориентированный граф, который обозначается G_u. Каждому узлу v графа G_d соответствуют три узла графа G_u — v_0, v_1 и v_2. Граф G_u содержит следующие ребра.

1. Каждому узлу v графа G_d в графе G_u соответствуют ребра $(v^{(0)}, v^{(1)})$ и $(v^{(1)}, v^{(2)})$.
2. Если G_d содержит дугу $x \to w$, то G_u содержит ребро $(v^{(2)}, w^{(0)})$.

На рис. 10.11 представлен набор ребер, включая ребро, соответствующее дуге $v \to w$.

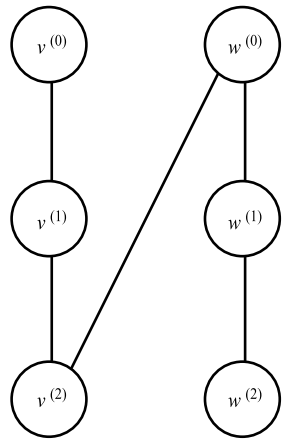


Рис. 10.11. Дуги G_d *заменяются в* G_u *ребрами, которые ведут от узлов с индексом 2 к узлам с индексом 0*

Построение G_u по G_d, очевидно, выполнимо за полиномиальное время. Нужно показать, что

- G_u имеет гамильтонов цикл тогда и только тогда, когда G_d имеет ориентированный гамильтонов цикл.

(*Достаточность*) Пусть $v_1, v_2, \ldots, v_n, v_1$ — ориентированный гамильтонов цикл. Тогда, безусловно,

$v_1^{(0)}, v_1^{(1)}, v_1^{(2)}, v_2^{(0)}, v_2^{(1)}, v_2^{(2)}, v_3^{(0)}, \ldots, v_n^{(0)}, v_n^{(1)}, v_n^{(2)}, v_1^{(0)}$

есть неориентированный гамильтонов цикл в G_u. Таким образом, происходит спуск по каждому столбцу, а затем скачок на вершину следующего столбца, соответствующий дуге в G_d.

(*Необходимость*) Отметим, что каждый узел $v^{(1)}$ в G_u имеет два ребра, и поэтому он должен присутствовать в гамильтоновом цикле вместе с узлами $v^{(0)}$ и $v^{(2)}$ — непосредственными предшественником и потомком. Таким образом, гамильтонов цикл в G_u должен содержать узлы, индексы которых образуют последовательность 0, 1, 2, 0, 1, 2, … или наоборот — 2, 1, 0, 2, 1, 0, … . Поскольку эти последовательности соответствуют обходам цикла в противоположных направлениях, то для определенности можно предполагать, что эта последовательность имеет вид 0, 1, 2, 0, 1, 2, … . Из построения G_u следует, что ребра этого цикла, которые выходят из узлов с индексом 2 и входят в узлы с индексом 0, являются дугами в G_d, и направление обхода таких ребер совпадает с направлением соответствующих дуг. Таким образом, существование неориентированного гамильтонова цикла в G_u влечет существование ориентированного гамильтонова цикла в G_d. □

Проблема. Проблема коммивояжера.

Вход. Неориентированный граф G, ребра которого имеют целочисленный вес, и предельное значение k.

Выход. Ответ “да” тогда и только тогда, когда G имеет гамильтонов цикл, общий вес ребер которого не превышает k.

Проблема, сводящаяся к данной. ГЦ.

Теорема 10.24. Проблема коммивояжера NP-полна.

Доказательство. Проблема ГЦ сводится к ПКОМ следующим образом. По данному графу G строится взвешенный граф G', который имеет те же узлы и ребра, что и G, и каждое ребро имеет вес 1. Предельное значение k берется равным n — числу узлов G. Таким образом, в G' существует гамильтонов цикл с весом n тогда и только тогда, когда существует гамильтонов цикл в G. □

10.4.6. Вывод относительно NP-полных проблем

Все сведения, построенные в данной главе, показаны на рис. 10.12. Заметим, что для всех специфических проблем (например ПКОМ) было представлено их сведение к ВЫП. Язык любой недетерминированной машины Тьюринга с полиномиальным временем в теореме 10.9 был сведен к проблеме ВЫП. Среди этих МТ была хотя бы одна, решающая ПКОМ, хотя бы одна, решающая НМ, и так далее, но они не указывались явно. Таким образом, все NP-полные проблемы полиномиально сводимы друг к другу, и, следовательно, представляют собой разные формы одной и той же проблемы.

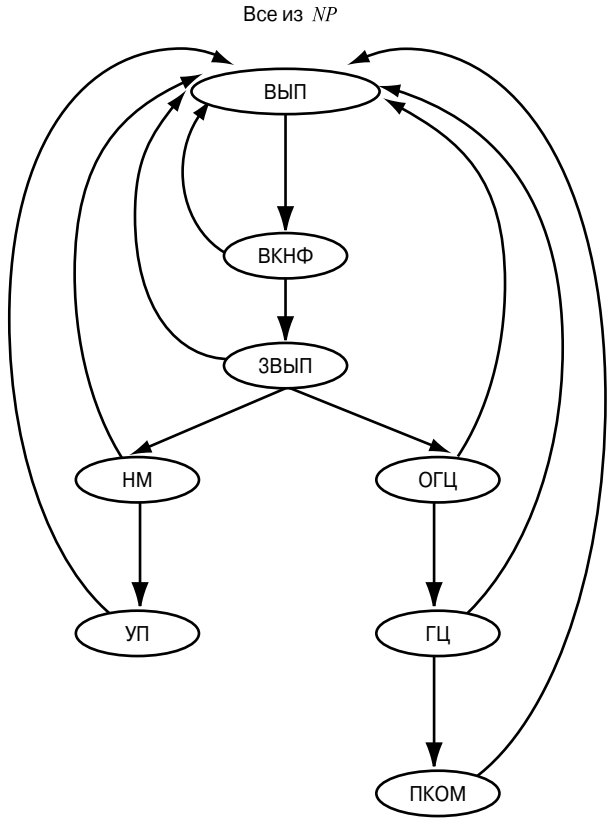


Рис. 10.12. Схема сведений NP-полных проблем

10.4.7. Упражнения к разделу 10.4

10.4.1. (∗) *k*-кликой в графе *G* называется множество из *k* узлов графа *G*, каждые два узла которого соединены ребром. Таким образом, 2-клика — это просто пара узлов, связанных ребром, а 3-клика — треугольник. Проблема клики (КЛИКА) состоит в том, чтобы: по данному графу *G* и константе *k* выяснить, содержит ли он *k*-клику. Выполните следующее:

а) найдите наибольшее значение *k*, при котором граф на рис. 10.1 принадлежит проблеме клики;

б) укажите зависимость количества ребер *k*-клики от *k*;

в) докажите NP-полноту проблемы клики, сведя к ней проблему узельного покрытия.

10.4.2. (∗!) *Проблема раскраски* состоит в том, чтобы для данного графа *G* и целого числа *k* выяснить, является ли *G* "*k*-раскрашиваемым", т.е. каждому узлу *G* можно приписать один из *k* цветов так, что никакие два узла, связанные ребром, не

будут окрашены в один цвет. Например, граф на рис. 10.1 является 3-раскрашиваемым, поскольку узлам 1 и 4 можно приписать красный цвет, узлу 2 — зеленый, а 3 — синий. Вообще, если граф имеет k-клику, то он не может быть менее, чем k-раскрашиваемым, хотя, возможно, для его раскраски нужно намного больше, чем k цветов.

В этом упражнении приводится часть конструкции, доказывающей NP-полноту проблемы раскраски. Оставшуюся часть восстанови*т*е самостоятельно. К данной проблеме сводится проблема ЗВЫП. Предположим, у нас есть формула в 3-КНФ с n переменными. Свед*е*ние переводит эту формулу в граф, часть которого изображена на рис. 10.13. Как видим, слева находятся $n + 1$ узлов c_0, c_1, …, c_n, образующих $(n + 1)$-клику. Поэтому все эти узлы должны быть раскрашены в разные цвета. Цвет, приписанный узлу c_j, мы будем называть "цветом c_j".

Каждой переменной x_i соответствуют два узла, которые можно обозначить как x_i и $\overline{x}_i$. Они соединены ребром, и поэтому не могут быть окрашены в один и тот же цвет. Кроме того, каждый узел x_i соединен с c_j для всех j, не равных 0 и i. Следовательно, один из узлов x_i и $\overline{x}_i$ должен иметь цвет c_0, а другой — цвет c_i. Будем считать, что литерал в узле цвета c_0 истинен, а во втором узле — ложен. Таким образом, выбранная раскраска отвечает некоторой подстановке.

Для завершения доказательства нужно построить для каждого дизъюнкта формулы соответствующий фрагмент графа. Завершить раскраску, используя только цвета от c_0 до c_n, должно быть возможно тогда и только тогда, когда каждый дизъюнкт формулы имеет значение "истина" при подстановке, соответствующей этому выбору цветов. Таким образом, построенный граф является $(n + 1)$-раскрашиваемым тогда и только тогда, когда данная формула выполнима.

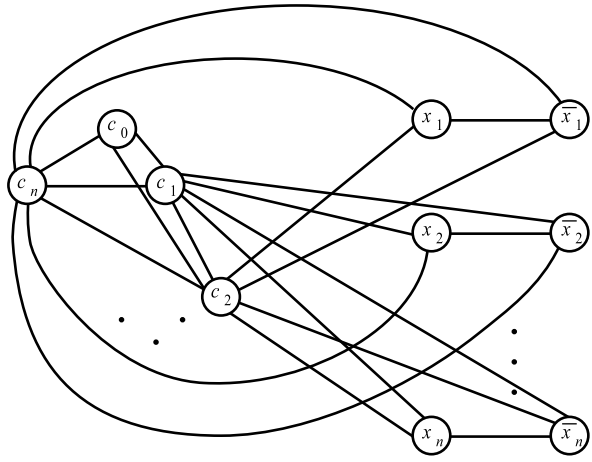


Рис. 10.13. Часть построения, показывающего NP-полноту проблемы раскраски

10.4.3. (!) Даже для относительно небольших графов NP-полные проблемы бывает очень трудно решить вручную. Рассмотрим граф на рис. 10.14:

а) (*) имеет ли этот граф гамильтонов цикл?

б) каково максимальное независимое множество в этом графе?

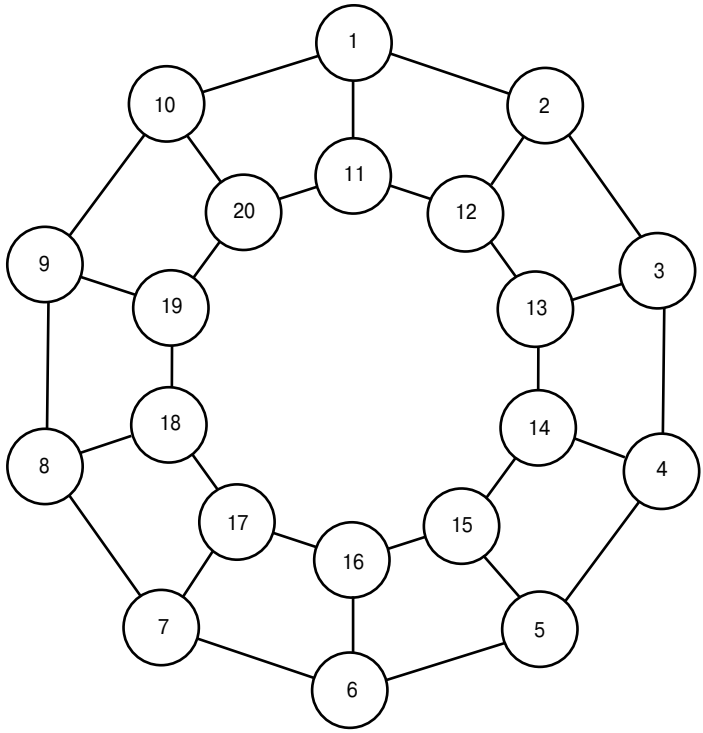


Рис. 10.14. Граф

в) что представляет собой наименьшее узельное покрытие этого графа?

г) каково наименьшее реберное покрытие этого графа (см. упражнение 10.4.4, *в*)?

д) является ли этот граф 2-раскрашиваемым?

10.4.4. Докажите NP-полноту следующих проблем.

а) *Проблема изоморфизма подграфа*. Даны графы G_1 и G_2. Содержит ли G_1 копию G_2 в качестве подграфа, т.е. можно ли найти в G_1 подмножество узлов, которые вместе с ребрами, инцидентными им в G_1, при правильном выборе соответствия между ними и узлами G_2 образуют точную копию графа G_2? *Указание.* Рассмотрите сведение проблемы клики из упражнения 10.4.1 к данной.

б) (!) *Проблема реберного покрытия обратных связей*. Даны граф G и целое число k. Имеет ли G подмножество из k ребер, для которого каждый цикл в G содержит хотя бы одно его ребро?[7]

в) (!) *Задача линейного целочисленного программирования*. Задано множество линейных ограничений-неравенств $\sum_{i=1}^{n} a_i x_i \le c$ или $\sum_{i=1}^{n} a_i x_i \ge c$, где a_i и c — целочисленные константы, а $x_1, x_2, \ldots, x_n$ — переменные. Существует ли набор целых значений этих переменных, при котором верны все ограничения-неравенства?

г) (!) *Проблема доминирующего множества*. Даны граф G и целое число k. Существует ли в G подмножество S, состоящее из k узлов, каждый узел которого либо принадлежит S, либо имеет в S смежный узел?

д) *Проблема пожарных депо*. Даны граф G, расстояние d и некоторое количество f "пожарных депо". Можно ли в G выбрать f узлов так, чтобы расстояние (число ребер, которые нужно пройти, чтобы попасть из одного узла в другой) от любого узла до некоторого пожарного депо не превышало d?

е) (*!) *Проблема половинной клики*. Дан граф G с четным числом узлов. Существует ли в G клика (см. упражнение 10.4.1), содержащая ровно половину узлов G? *Указание.* Сведите проблему клики к проблеме половинной клики. Вам нужно представить, каким образом следует добавлять узлы, чтобы наибольшая клика содержала нужное число узлов.

ж) (!!) *Проблема расписания с единичным временем выполнения*. Даны k "заданий" $T_1, T_2, \ldots, T_k$, число "процессоров" p, предел времени t и "ограничения предшествования" вида $T_i < T_j$ между парами заданий[8]. Существует ли *расписание* выполнения заданий со следующими свойствами?

1. Каждое задание назначено на одну единицу времени между 1 и t.
2. На каждую единицу времени назначено не более p заданий.
3. Учтены ограничения предшествования: если существует ограничение $T_i < T_j$, то задание T_i назначено на более раннюю единицу времени, чем T_j.

з) (!!) *Проблема точного покрытия*. Дано множество S и набор его подмножеств $S_1, S_2, \ldots, S_n$. Можно ли указать набор множеств $T \subseteq \{S_1, S_2, \ldots, S_n\}$ так,

---

[7] В этом месте оригинал книги содержал формулировку проблемы реберного покрытия, которая в действительности полиномиальна. Исправление ошибки было выставлено авторами в Internet (см. предисловие). — *Прим. ред.*

[8] Неявно предполагается, что ограничения предшествования являются отношением частичного порядка. — *Прим. ред.*

чтобы каждый элемент x множества S принадлежал ровно одному из элементов набора T?

и) (**!!**) *Проблема разбиения*. Можно ли разбить данный список из k целых чисел $i_1, i_2, \ldots, i_k$ на две части с одинаковыми суммами элементов? *Указание.* На первый взгляд, эта проблема принадлежит классу $\mathcal{P}$, поскольку можно предположить, что сами целые числа невелики. Действительно, если эти целые числа ограничены полиномом относительно количества чисел k, то существует полиномиальный алгоритм решения. Однако в списке из k целых чисел в двоичной системе, имеющем общую длину n, могут быть элементы, значения которых почти экспоненциальны относительно n.[9]

10.4.5. *Гамильтонов путь* в графе G есть упорядочение всех узлов $n_1, n_2, \ldots, n_k$, при котором для каждого $i = 1, 2, \ldots, k-1$ существует ребро из n_i в n_{i+1}. *Ориентированный гамильтонов путь* — это то же самое для ориентированного графа (должна существовать дуга из n_i в n_{i+1}). Отметим, что условия гамильтонова пути лишь немного слабее условий, налагаемых на гамильтонов цикл. Проблема (ориентированного) гамильтонова пути состоит в следующем: имеет ли данный (ориентированный) граф хотя бы один (ориентированный) гамильтонов путь?

а) (*****) Докажите, что проблема ориентированного гамильтонова пути NP-полна. *Указание*. Сведите проблему ОГЦ к данной. Выберите произвольный узел и разбейте его на два узла так, чтобы они были конечными точками ориентированного гамильтонова пути, и чтобы этот путь существовал тогда и только тогда, когда исходный граф имеет ориентированный гамильтонов цикл.

б) Покажите, что проблема неориентированного гамильтонова пути NP-полна. *Указание*. Используйте конструкцию из теоремы 10.23.

в) (***!**) Покажите NP-полноту следующей проблемы: по данным графу G и целому числу k выяснить, имеет ли G остовное дерево, число листьев которого не более k. *Указание.* Сведите проблему гамильтонова пути к данной.

г) (**!**) Покажите NP-полноту следующей проблемы: по данным графу G и целому числу d выяснить, имеет ли G остовное дерево, в котором степень любого узла не превышает d? (*Степенью* узла n в остовном дереве называется число ребер этого дерева, инцидентных n.)

---

[9] В оригинале данная проблема была названа *"задачей о ранце" (knapsack problem)*, хотя последняя имеет такой вид: "Существует ли для данной последовательности целых чисел $S = i_1, i_2, \ldots, i_n$ и целого числа k подпоследовательность в S, сумма членов которой равна k?". Очевидно, что проблема разбиения является частным случаем задачи о ранце при $\sum_{j=1}^{n} i_j = 2k$. — *Прим. ред.*

10.5. Резюме

- *Классы $\mathcal{P}$ и $\mathcal{NP}$*. Класс $\mathcal{P}$ состоит из всех языков или проблем, допускаемых машинами Тьюринга, время работы которых полиномиально зависит от длины входа. $\mathcal{NP}$ — это класс языков или проблем, допускаемых недетерминированными МТ, у которых время работы при любой последовательности недетерминированных выборов полиномиально ограничено.
- *Вопрос о $\mathcal{P} = \mathcal{NP}$*. Точно не известно, различны ли классы языков $\mathcal{P}$ и $\mathcal{NP}$. Но есть серьезные основания полагать, что в $\mathcal{NP}$ есть языки, не принадлежащие $\mathcal{P}$.
- *Полиномиальные сведения*. Если экземпляры одной проблемы преобразуемы за полиномиальное время в экземпляры другой, дающие тот же ответ ("да" или "нет"), то говорят, что первая проблема полиномиально сводится ко второй.
- *$\mathcal{NP}$-полные проблемы*. Язык является NP-полным, если он принадлежит $\mathcal{NP}$ и всякий язык из $\mathcal{NP}$ можно полиномиально свести к нему. Мы верим в то, что ни одна из NP-полных проблем не принадлежит $\mathcal{P}$. Тот факт, что ни для одной из тысяч известных NP-полных проблем до сих пор не найден полиномиальный алгоритм разрешения, только укрепляет нашу уверенность.
- *NP-полная проблема выполнимости*. В теореме Кука была показана NP-полнота проблемы выполнимости булевой формулы (ВЫП). Доказательство проводилось путем сведения любой проблемы из $\mathcal{NP}$ к проблеме ВЫП. Кроме того, проблема выполнимости остается NP-полной, даже если вид формулы ограничен произведением сомножителей, каждый из которых содержит лишь три литерала (проблема ЗВЫП).
- *Другие NP-полные проблемы*. NP-полнота очень многих проблем доказывается путем сведения к ним других проблем, о которых заранее известно, что они NP-полные. Здесь приведены сведения, доказывающие NP-полноту проблем независимого множества, узельного покрытия, ориентированного и неориентированного гамильтонова цикла, а также коммивояжера.

10.6. Литература

Понятие NP-полноты как свидетельство того, что проблему нельзя решить за полиномиальное время, и доказательство NP-полноты ВЫП, ВКНФ и ЗВЫП впервые были приведены в работе С. Кука [3]. За ней последовала не менее важная статья Р. Карпа [6], в которой было показано, что NP-полнота — это не изолированный феномен; она присуща многим трудным комбинаторным проблемам, изучавшимся долгие годы в исследовании операций и других дисциплинах. Из этой статьи взяты все проблемы, NP-полнота которых доказана в разделе 10.4 — независимое множество, узельное покрытие, гамиль-

тонов цикл и коммивояжер. Кроме того, в ней можно найти решения некоторых проблем, рассмотренных в упражнениях: реберное покрытие обратных связей, разбиение, раскраска и точное покрытие.

В книге Гэри и Джонсона [4] собрано воедино большое число фактов, касающихся NP-полноты различных проблем, а также приведены их частные случаи, разрешимые за полиномиальное время. В [5] содержатся статьи об аппроксимации решения NP-полной проблемы за полиномиальное время.

Необходимо упомянуть еще несколько работ, послуживших серьезным вкладом в теорию NP-полноты. Первые исследования классов языков, определяемых временем работы машин Тьюринга, были предприняты Хартманисом и Стирнзом [8]. Кобхем [2] первым выделил класс $\mathcal{P}$ как понятие, в котором отражено коренное отличие от алгоритмов, имеющих конкретное полиномиальное время работы, например $O(n^2)$. Несколько позже (но независимо) идею NP-полноты исследовал Левин [7].

NP-полнота задачи линейного целочисленного программирования (упражнение 10.4.4, *в*) появилась в работе [1], а также в неопубликованных заметках Дж. Гатена (J. Gathen) и М. Зивекинга (M. Sieveking). NP-полнота проблемы расписания с единичным временем выполнения (упражнение 10.4.4, *ж*) доказана в [9].

1. I. Borosh and L. B. Treybig, “Bounds on positive integral solution of linear Diophantine equations”, *Proceedings of the AMS* **55** (1976), pp. 299–304.
2. A. Cobham, “The intrinsic computational difficulty of functions”, *Proc.1964 Congress for Logic, Mathematics, and the Philosophy of Science*, North Holland, Amsterdam, pp. 24–30.
3. S. C. Cook, “The complexity of theorem-proving procedures”, *Third ACM Symposium on Theory of Computing* (1971), ACM, New York, pp. 151–158. (Кук С. Сложность процедур вывода теорем. — Кибернетический сборник, новая серия, вып. 12. — М.: Мир, 1975. — С. 5–15.)
4. M. R. Garey and D. S. Johnson, *Computers and Intractability: a Guide to the Theory of NP-Completeness*, H. Freeman, New York, 1979. (Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. — М.: Мир, 1982.)
5. D. S. Hochbaum (ed.), *Approximation Algorithms for NP-Hard Problems*, PWS Publishing Co., 1996.
6. R. M. Karp, “Reducibility among combinatorial problems”, in *Complexity of Computer Computations* (R. E. Miller, ed.), Plenum Press, New York, (1972), pp. 85–104. (Карп Р. М. Сводимость комбинаторных проблем. — Кибернетический сборник, новая серия, вып. 12. — М.: Мир, 1975. — С. 16–38.)
7. Л. А. Левин. Универсальные проблемы упорядочения // Проблемы передачи информации. — 1973. — Т. 9, № 3. — С. 265–266.

8. J. Hartmanis and R. E. Stearns, “On the computational complexity of algorithms”, *Transactions of the AMS* **117** (1965), pp. 285–306.
9. J. D. Ullman, “NP-complete scheduling problems”, *J.Computer and System Sciences* **10**:3 (1975), pp. 384–393.

ГЛАВА 11

Дополнительные классы проблем

Труднорешаемые проблемы не ограничиваются классом $\mathcal{NP}$. Существует много других классов труднорешаемых проблем, по разным причинам представляющих интерес. Некоторые вопросы, связанные с этими классами, вроде $\mathcal{P} = \mathcal{NP}$, остаются нерешенными.

Вначале рассматривается класс, тесно связанный с $\mathcal{P}$ и $\mathcal{NP}$, — класс дополнений языков из $\mathcal{NP}$, часто называемый "со-$\mathcal{NP}$". Если $\mathcal{P} = \mathcal{NP}$, то со-$\mathcal{NP}$ равен обоим, поскольку $\mathcal{P}$ замкнут относительно дополнения. Однако более вероятно, что со-$\mathcal{NP}$ отличается от обоих этих классов, и ни одна $\mathcal{NP}$-полная проблема не принадлежит со-$\mathcal{NP}$.

Далее изучается класс $\mathcal{PS}$. Он образован проблемами, которые решаются на машинах Тьюринга с использованием объема ленты, полиномиального относительно длины входа. Этим машинам Тьюринга разрешается использовать экспоненциальное время, но ограниченную часть ленты. В отличие от ситуации с полиномиальным временем, здесь можно доказать, что при таком же ограничении пространства недетерминизм не увеличивает мощности МТ. Однако, несмотря на то, что $\mathcal{PS}$ очевидным образом включает $\mathcal{NP}$, неизвестно, равны ли эти классы. Ожидается, что они не равны, и здесь мы опишем проблему, которая полна для $\mathcal{PS}$ и предположительно не принадлежит $\mathcal{NP}$.

Затем обратимся к рандомизированным алгоритмам и двум классам языков, лежащим между $\mathcal{P}$ и $\mathcal{NP}$. Один из этих классов обозначается $\mathcal{RP}$ ("random polynomial" languages — "случайные полиномиальные" языки). Эти языки имеют алгоритм, который работает полиномиальное время, используя "бросание монеты" или (на практике) генератор случайных чисел. Алгоритм или подтверждает принадлежность входа языку, или отвечает "не знаю". Кроме того, если вход принадлежит языку, то существует некоторая вероятность больше 0, что алгоритм будет "докладывать об успехе", поэтому его повторное применение будет с вероятностью, близкой к 1, подтверждать принадлежность.

Второй класс, называемый $\mathcal{ZPP}$ (zero-error, probabilistic polynomial — безошибочные, вероятностные полиномиальные), также использует рандомизацию, но алгоритмы для языков этого класса отвечают: "да, вход принадлежит языку" или "нет, не принадлежит". Ожидаемое время работы алгоритма полиномиально. Однако возможно выполнение алгоритма, требующее больше времени, чем полиномиальное.

Чтобы увязать приведенные понятия, рассмотрим важный вопрос проверки простоты. Многие современные системы шифрования основаны на следующих свойствах.

1. Способность быстро находить большие простые числа, чтобы защитить от посторонних воздействий канал сообщения между компьютерами.
2. Предположение о том, что разложение на целые сомножители требует экспоненциального времени, измеряемого в виде функции от длины n двоичной записи целого числа.

Будет показано, что проверка простоты чисел принадлежит как $\mathcal{NP}$, та и со-$\mathcal{NP}$, поэтому вряд ли удастся доказать NP-полноту этой проверки. Это плохо, так как доказательство NP-полноты является наиболее действенным доводом в пользу того, что проблема, скорее всего, требует экспоненциального времени. Будет показано также, что проверка простоты принадлежит классу $\mathcal{RP}$. Это и хорошо, и плохо. Хорошо, поскольку системы шифрования, использующие простые числа, применяют для их поиска алгоритм из класса $\mathcal{RP}$. Плохо, так как подтверждается предположение, что доказать NP-полноту проверки простоты так и не удастся.

11.1. Дополнения языков из $\mathcal{NP}$

Класс языков $\mathcal{P}$ замкнут относительно дополнения (см. упражнение 10.1.6). Это легко обосновать. Пусть L принадлежит $\mathcal{P}$, а M — МТ для L. Для допускания $\overline{L}$ изменим M: введем новое допускающее состояние q и новые переходы в q из тех состояний, в которых M останавливается, не допуская. Сделаем исходные допускающие состояния недопускающими. Тогда измененная МТ допускает $\overline{L}$ и работает столько же времени, сколько M, с возможным добавлением одного перехода. Таким образом, $\overline{L}$ принадлежит $\mathcal{P}$, если L принадлежит $\mathcal{P}$.

Неизвестно, замкнут ли класс $\mathcal{NP}$ относительно дополнения, но похоже, что нет. В частности, можно ожидать, что если язык NP-полон, то его дополнение не принадлежит $\mathcal{NP}$.

11.1.1. Класс языков со-$\mathcal{NP}$

Со-$\mathcal{NP}$ — это класс языков, дополнения которых принадлежат $\mathcal{NP}$. Напомним, что дополнение любого языка из $\mathcal{P}$ также принадлежит $\mathcal{P}$ и, следовательно, $\mathcal{NP}$. С другой стороны, мы верим, что дополнения NP-полных проблем не принадлежат $\mathcal{NP}$, поэтому в со-$\mathcal{NP}$ нет ни одной NP-полной проблемы. Аналогично мы верим, что дополнения NP-полных проблем, по определению содержащиеся в со-$\mathcal{NP}$, не находятся в $\mathcal{NP}$. На рис. 11.1 показана предполагаемая взаимосвязь этих классов. Однако не следует забывать, что если $\mathcal{P}$ окажется равным $\mathcal{NP}$, то все три класса совпадут.

Пример 11.1. Рассмотрим дополнение языка ВЫП, которое принадлежит со-$\mathcal{NP}$ и обозначается *НВЫП* (невыполнимая). НВЫП включает все коды невыполнимых булевых формул, а также цепочки, которые не являются кодами допустимых булевых формул. Мы верим, что НВЫП не принадлежит $\mathcal{NP}$, но доказать это не можем.

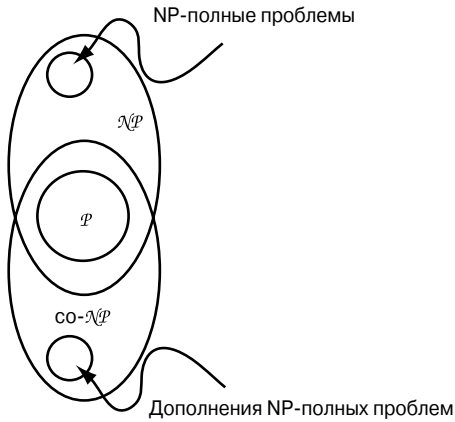


Рис. 11.1. Предполагаемая взаимосвязь между со-$\mathcal{NP}$ и другими классами языков

Еще одним примером проблемы, которая предположительно находится в со-$\mathcal{NP}$, но не в $\mathcal{NP}$, служит язык ТАВТ — множество всех (закодированных) булевых формул, являющихся *тавтологиями*, т.е. истинных при любой подстановке. Заметим, что формула E является тавтологией тогда и только тогда, когда $\neg E$ невыполнима. Таким образом, ТАВТ и НВЫП связаны так, что если булева формула E принадлежит ТАВТ, то $\neg E$ принадлежит НВЫП, и наоборот. Однако НВЫП содержит также цепочки, не представляющие допустимые выражения, тогда как все цепочки в ТАВТ — допустимые. □

11.1.2. NP-полные проблемы и со-$\mathcal{NP}$

Предположим, что $\mathcal{P} \neq \mathcal{NP}$. Возможно, ситуация, связанная с со-$\mathcal{NP}$, отличается от представленной на рис. 11.1, поскольку $\mathcal{NP}$ и со-$\mathcal{NP}$ могут совпадать, но быть больше $\mathcal{P}$. Таким образом, проблемы типа НВЫП или ТАВТ могли бы разрешаться за полиномиальное недетерминированное время, т.е. принадлежать $\mathcal{NP}$, не имея детерминированного полиномиального решения. Однако отсутствие хотя бы одной NP-полной проблемы, имеющей дополнение в $\mathcal{NP}$, обосновывает, что $\mathcal{NP} \neq$ со-$\mathcal{NP}$. Докажем это в следующей теореме.

Теорема 11.2. $\mathcal{NP}$ = со-$\mathcal{NP}$ тогда и только тогда, когда существует NP-полная проблема, дополнение которой принадлежит $\mathcal{NP}$.

Доказательство. (*Необходимость*) Если бы $\mathcal{NP}$ и со-$\mathcal{NP}$ совпадали, то каждая NP-полная проблема L находилась бы как в $\mathcal{NP}$, так и в со-$\mathcal{NP}$. Но дополнение проблемы из со-$\mathcal{NP}$ находится в $\mathcal{NP}$, поэтому дополнение L принадлежало бы $\mathcal{NP}$.

(*Достаточность*) Предположим, что P — NP-полная проблема, дополнение $\overline{P}$ которой принадлежит $\mathcal{NP}$. Тогда для любого языка L из $\mathcal{NP}$ существует полиномиальное све-

дение L к P. Это сведение является одновременно полиномиальным сведением $\overline{L}$ к $\overline{P}$. Докажем равенство классов $\mathcal{NP}$ и со-$\mathcal{NP}$, показав их взаимное включение.

$\mathcal{NP} \subseteq$ со-$\mathcal{NP}$. Пусть L принадлежит $\mathcal{NP}$. Тогда $\overline{L}$ принадлежит со-$\mathcal{NP}$. Объединим полиномиальное сведение $\overline{L}$ к $\overline{P}$ с предполагаемым недетерминированным полиномиальным алгоритмом для $\overline{P}$, чтобы получить такой же алгоритм для $\overline{L}$. Тогда для любого L из $\mathcal{NP}$ его дополнение $\overline{L}$ также принадлежит $\mathcal{NP}$. Следовательно, L, будучи дополнением языка из $\mathcal{NP}$, находится в со-$\mathcal{NP}$. Отсюда $\mathcal{NP} \subseteq$ со-$\mathcal{NP}$.

Со-$\mathcal{NP} \subseteq \mathcal{NP}$. Пусть L принадлежит со-$\mathcal{NP}$. Тогда существует полиномиальное сведение $\overline{L}$ к P, поскольку P является NP-полным, а $\overline{L}$ принадлежит $\mathcal{NP}$. Это сведение является также сведением L к $\overline{P}$. Поскольку $\overline{P}$ принадлежит $\mathcal{NP}$, объединим это сведение с недетерминированным полиномиальным алгоритмом для $\overline{P}$ и убедимся, что L принадлежит $\mathcal{NP}$. □

11.1.3. Упражнения к разделу 11.1

11.1.1. (!) Ниже представлено несколько проблем. Для каждой определите, принадлежит она $\mathcal{NP}$ или со-$\mathcal{NP}$. Опишите дополнение каждой проблемы. Если проблема или ее дополнение являются NP-полными, докажите это.

а) (*) Проблема ИСТ-ВЫП. По данной булевой формуле E, истинной, когда все переменные имеют значение “истина”, определить, существует ли еще одна подстановка, удовлетворяющая E.

б) Проблема ЛОЖЬ-ВЫП. Дана булева формула E, ложная, когда все переменные имеют значение “ложь”. Определить, существует ли еще одна подстановка, при которой E ложна.

в) Проблема ДВЕ-ВЫП. По данной булевой формуле E определить, существуют ли хотя бы две подстановки, удовлетворяющие E.

г) Проблема ПОЧТИ-ТАВТ. Дана булева формула E. Определить, является ли она ложной не более, чем при одной подстановке.

11.1.2. (*!) Предположим, что существует взаимно однозначная функция f, которая отображает одни n-битовые целые числа в другие и обладает следующими свойствами.

1. $f(x)$ можно вычислить за полиномиальное время.
2. $f^{-1}(x)$ нельзя вычислить за полиномиальное время.

Докажите, что в таком случае язык, состоящий из пар целых чисел (x, y), для которых

$f^{-1}(x) < y$,

принадлежит ($\mathcal{NP} \cap$ со-$\mathcal{NP}$) – $\mathcal{P}$.

11.2. Проблемы, разрешимые в полиномиальном пространстве

Рассмотрим класс проблем, включающий $\mathcal{NP}$ и, возможно, еще больше, но уверенности в этом нет. Этот класс определяется машинами Тьюринга, которые могут использовать объем пространства, полиномиальный относительно размера входа; время работы роли не играет. Вначале классы языков, допускаемых детерминированными и недетерминированными МТ с полиномиальным ограничением пространства, различаются, но затем будет показано, что они совпадают.

Для полиномиального пространства существуют полные проблемы P в том смысле, что все проблемы данного класса сводимы за полиномиальное *время* к P. Таким образом, если P принадлежит $\mathcal{P}$ или $\mathcal{NP}$, то все языки МТ с полиномиально ограниченным пространством также принадлежат $\mathcal{P}$ или $\mathcal{NP}$, соответственно. Мы представим пример такой проблемы — "булевы формулы с кванторами".

11.2.1. Машины Тьюринга с полиномиальным пространством

Машина Тьюринга с полиномиальным ограничением пространства представлена на рис. 11.2. Существует некоторый полином $p(n)$, для которого МТ, имея вход w длиной n, не посещает более $p(n)$ клеток ленты. Согласно теореме 8.12 можно считать, что лента является односторонней, а МТ не сдвигается влево от начала входа.

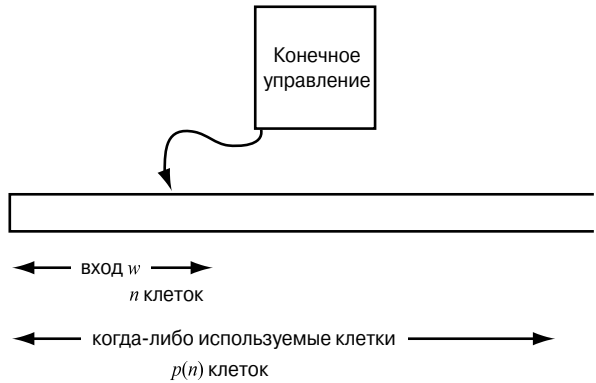


Рис. 11.2. МТ, использующая полиномиальное пространство

Класс языков $\mathcal{PS}$ (*polynomial space* — полиномиальное пространство) определяется как множество языков $L(M)$ детерминированных МТ M с полиномиально ограниченным пространством. Определим также класс $\mathcal{NPS}$ (*nondeterministic polynomial space — недетерминированное полиномиальное пространство*) как множество языков $L(M)$ недетерминированных МТ M с полиномиально ограниченным пространством. Очевидно, $\mathcal{PS} \subseteq \mathcal{NPS}$, по-

скольку каждая детерминированная МТ является недетерминированной. Будет доказан неожиданный результат: $\mathcal{PS} = \mathcal{NPS}$.[1]

11.2.2. Связь $\mathcal{PS}$ и $\mathcal{NPS}$ с определенными ранее классами

Отметим сразу, что включения $\mathcal{P} \subseteq \mathcal{PS}$ и $\mathcal{NP} \subseteq \mathcal{NPS}$ очевидны. Если МТ совершает полиномиальное число переходов, то она использует не более, чем полиномиальное число клеток, точнее, число посещаемых ею клеток не более, чем на 1 превышает число переходов. Доказав, что $\mathcal{PS} = \mathcal{NPS}$, мы убедимся в справедливости цепочки включений $\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{PS}$.

Существенное свойство МТ с полиномиально ограниченным пространством состоит в том, что они могут совершить не более, чем экспоненциальное число переходов перед повторением МО. Этот факт нужен для доказательства других интересных результатов, касающихся $\mathcal{PS}$, а также для того, чтобы показать, что $\mathcal{PS}$ содержит только рекурсивные языки, т.е. языки с алгоритмами. Отметим, что в определении $\mathcal{PS}$ или $\mathcal{NPS}$ останов МТ не упоминается, т.е. МТ может зацикливаться, не выходя за пределы полиномиальной по объему части ленты.

Теорема 11.3. Если M — МТ с полиномиально ограниченным пространством, а $p(n)$ — полиномиальный предел ее пространства, то существует константа c, при которой M, допуская свой вход w длиной n, делает это в пределах $c^{1+p(n)}$ переходов.

Доказательство. Основная идея состоит в том, что M должна повторить МО перед тем, как совершить более $c^{1+p(n)}$ переходов. Если M повторяет МО и затем допускает, то должна существовать более короткая последовательность МО, ведущих к допусканию, т.е., если $\alpha \overset{*}{\vdash} \beta \overset{*}{\vdash} \beta \overset{*}{\vdash} \gamma$, где α — начальное, β — повторяемое, а γ — допускающее МО, то $\alpha \overset{*}{\vdash} \beta \overset{*}{\vdash} \gamma$ — более короткая последовательность МО, приводящая к допусканию.

В обосновании существования c используется то, что у МТ с ограниченным пространством есть лишь ограниченное число МО. Точнее, пусть t — число ленточных символов, а s — состояний МТ M. Тогда число различных МО M, использующей $p(n)$ клеток, не более $sp(n)t^{p(n)}$, т.е. можно выбрать одно из s состояний, поместить ленточную головку в одну из $p(n)$ позиций и заполнить $p(n)$ клеток любой из $t^{p(n)}$ последовательностей ленточных символов.

Выберем $c = s + t$ и раскроем бином $(t + s)^{1+p(n)}$:

$$t^{1+p(n)} + (1 + p(n))st^{p(n)} + \ldots$$

Заметим, что второе слагаемое не меньше $sp(n)t^{p(n)}$; это доказывает, что $c^{1+p(n)}$ не меньше числа возможных конфигураций M. Отметим, что, если M допускает вход w длиной n, то она выполняет последовательность переходов без повторения МО. Следовательно, M допускает, совершив переходов не больше $c^{1+p(n)}$ — количества различных МО. □

---

[1] Во многих работах этот класс обозначается PSPACE. Однако здесь для его обозначения применяется сокращение $\mathcal{PS}$, достаточное ввиду установления равенства $\mathcal{PS} = \mathcal{NPS}$.

Теорему 11.3 можно использовать для преобразования любой МТ с полиномиально ограниченным пространством в эквивалентную машину, которая всегда останавливается, совершив не более экспоненциального числа переходов. Зная, что МТ допускает в пределах экспоненциального числа переходов, можно подсчитать количество совершаемых переходов и остановить работу, не допуская, если сделано достаточно много переходов без допускания.

Теорема 11.4. Если L — язык из $\mathcal{PS}$ ($\mathcal{NPS}$), то L допускается детерминированной (недетерминированной) МТ с полиномиально ограниченным пространством, которая для некоторого полинома $q(n)$ и константы c останавливается после не более чем $c^{q(n)}$ переходов.

Доказательство. Докажем утверждение для детерминированных МТ. Для НМТ доказательство аналогично. Пусть L допускается МТ M_1, имеющей полиномиальное ограничение пространства $p(n)$. Тогда по теореме 11.3, если M_1 допускает w, то делает это в пределах $c^{1+p(|w|)}$ шагов.

Построим новую МТ M_2 с двумя лентами. На первой ленте M_2 имитирует M_1, а на второй ведет счет до $c^{1+p(|w|)}$, используя основание c. Если счет у M_2 достигает этого числа, то она останавливается, не допуская. Таким образом, M_2 использует $1 + p(|w|)$ клеток второй ленты. Поскольку M_1 использует не более $p(|w|)$ клеток своей ленты, M_2 также использует не более $p(|w|)$ клеток первой ленты.

Преобразуя M_2 в одноленточную МТ M_3, можно гарантировать, что M_3 использует не более $1 + p(n)$ клеток ленты при обработке любого входа длиной n. Хотя M_3 может использовать квадрат времени работы M_2, это время не превышает $O(c^{2p(n)})$.[2]

МТ M_3 совершает не более $dc^{2p(n)}$ переходов для некоторой константы d, поэтому можно выбрать $q(n) = 2p(n) + \log_c d$. Тогда M_3 совершает не более $c^{q(n)}$ шагов. M_2 всегда останавливается, поэтому то же делает M_3. M_1 допускает L, поэтому его же допускают M_2 и M_3. Таким образом, M_3 удовлетворяет утверждению теоремы. □

11.2.3. Детерминированное и недетерминированное полиномиальное пространство

Сравнение классов $\mathcal{P}$ и $\mathcal{NP}$ затруднительно, но, на удивление, сравнить классы $\mathcal{PS}$ и $\mathcal{NPS}$ легко — они совпадают. Доказательство основано на имитации недетерминированной МТ, пространство которой ограничено полиномом $p(n)$, с помощью детерминированной МТ, имеющей ограничение пространства $O(p^2(n))$.

---

[2] В действительности, общее правило из теоремы 8,10 не является сильнейшим утверждением, которое можно установить. Поскольку на любой ленте используется только $1 + p(n)$ клеток, имитируемые головки при переходе от многих лент к одной не могут разойтись более, чем на $1 + p(n)$ клеток. Таким образом, $c^{1+p(n)}$ переходов многоленточной МТ M_2 можно проимитировать за $O(p(n)c^{p(n)})$ шагов, что меньше указанного $O(c^{2p(n)})$.

"Сердцем" доказательства является детерминированная рекурсивная проверка, может ли НМТ N перейти от МО I к МО J не более, чем за m переходов. ДМТ D систематически проверяет все промежуточные МО K, чтобы убедиться, может ли N перейти от I к K за $m/2$ переходов, а затем перейти от K к J за $m/2$ переходов. Итак, представим, что существует рекурсивная функция *reach*(I, J, m), решающая, верно ли, что $I \overset{*}{\vdash} J$ не более, чем за $m/2$ переходов.

Рассмотрим ленту машины D как магазин, в который помещаются аргументы рекурсивных вызовов функции *reach*, т.е. один *элемент магазина* хранит [I, J, m]. Алгоритм функции *reach* представлен на рис. 11.3.[3]

```
BOOLEAN FUNCTION reach(I, J, m)
   ID: I,J; INT: m;
   BEGIN
      IF (m == 1) THEN /* базис */ BEGIN
         проверить, что I == J или I может стать J за 1 переход;
         если так, RETURN TRUE, иначе RETURN FALSE;
      END;
      ELSE /* индуктивная часть */ BEGIN
         FOR каждая возможная МО K DO
            IF (reach(I,K,m/2) AND reach(K,J,m/2)) THEN
               RETURN TRUE;
         RETURN FALSE;
      END;
   END;
```

Рис. 11.3. Рекурсивная функция reach *проверяет, можно ли за установленное число переходов перейти от одного МО к другому*

Важно заметить, что, хотя *reach* вызывает саму себя дважды, она делает это последовательно, поэтому в любой момент времени активен лишь один вызов. Таким образом, если начать с элемента магазина [I_1, J_1, m], то в любой момент времени существует только один вызов [I_2, J_2, $m/2$], один вызов [I_3, J_3, $m/4$], еще один [I_4, J_4, $m/8$] и так далее, пока в некоторой точке третий аргумент не станет равным 1. В этот момент *reach* может применить базисный шаг и не нуждается в рекурсивных вызовах. Она проверяет, верно ли, что $I \vdash J$ или $I = J$, и возвращает `true`, если это так, и `false` — если нет. На рис. 11.4 представлен общий вид магазина ДМТ D, когда существует столько активных вызовов *reach*, сколько возможно при начальном количестве переходов m.

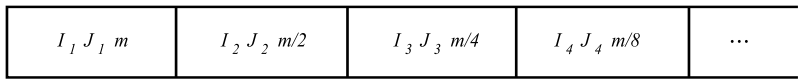


Рис. 11.4. Лента ДМТ, имитирующей НМТ с помощью рекурсивных вызовов reach

[3] ID — MO (Instantaneous Description — мгновенное описание). — *Прим. перев.*

Хотя может показаться, что возможно много вызовов функции *reach*, и лента, изображенная на рис. 11.4, может стать очень длинной, мы докажем, что она не будет "слишком длинной", т.е., если начать с числа переходов m, то в любой момент времени на ленте не может быть более $\log_2 m$ элементов магазина. Поскольку теорема 11.4 гарантирует, что НМТ N не может совершить более $c^{p(n)}$ переходов, начальное значение m также не превышает $c^{p(n)}$. Таким образом, число элементов магазина не превосходит $\log_2 c^{p(n)}$, т.е. $O(p(n))$, и у нас есть все необходимое для доказательства следующей теоремы.

Теорема 11.5 (теорема Сэвича). $\mathcal{PS} = \mathcal{NPS}$.

Доказательство. Очевидно, что $\mathcal{PS} \subseteq \mathcal{NPS}$, поскольку каждая ДМТ является также и НМТ. Таким образом, достаточно доказать, что $\mathcal{NPS} \subseteq \mathcal{PS}$, т.е., если L допускается НМТ N с ограничением пространства $p(n)$, где $p(n)$ — полином, то L также допускается ДМТ D, пространство которой ограничено другим полиномом $q(n)$. В действительности будет показано, что $q(n)$ можно выбрать равным по порядку квадрату $p(n)$.

По теореме 11.3 можно предполагать, что, если N допускает, то делает это в пределах $c^{1+p(n)}$ шагов, где c — некоторая константа. Получив вход w длиной n, D исследует, что делает N со входом w, многократно помещая тройки вида $[I_0, J, m]$ на свою ленту и вызывая *reach* с этими аргументами. Здесь I_0 является начальным МО машины N со входом w, J — некоторое допускающее МО, в котором используется не более $p(n)$ клеток (различные J систематически перечисляются машиной D с помощью рабочей ленты), а $m = c^{1+p(n)}$.

Выше было обосновано, что рекурсивных вызовов, активных одновременно, не может быть более $\log_2 m$, т.е. один с аргументом m, один с аргументом $m/2$, один с аргументом $m/4$ и так далее до 1. Таким образом, существует не более $\log_2 m$ элементов магазина, а $\log_2 m$ есть $O(p(n))$.

Элементы магазина сами по себе занимают пространство $O(p(n))$. Причина в том, что для записи каждого МО нужно $1 + p(n)$ клеток, а для записи m в двоичном виде — $\log_2 c^{1+p(n)}$, т.е. $O(p(n))$ клеток. Таким образом, полный элемент магазина, состоящий из двух МО и целого числа, занимает пространство $O(p(n))$.

Поскольку D может иметь не более $O(p(n))$ элементов магазина, общее количество используемого пространства составляет $O(p^2(n))$. Это количество полиномиально, если $p(n)$ — полином, и мы делаем вывод, что L имеет ДМТ с полиномиально ограниченным пространством. □

В заключение можно уточнить информацию о классах сложности, включая классы с полиномиально ограниченным пространством. Полная диаграмма представлена на рис. 11.5.

11.3. Проблема, полная для $\mathcal{PS}$

В этом разделе представлена проблема, которая называется "булевы формулы с кванторами", и показано, что она полна для $\mathcal{PS}$.

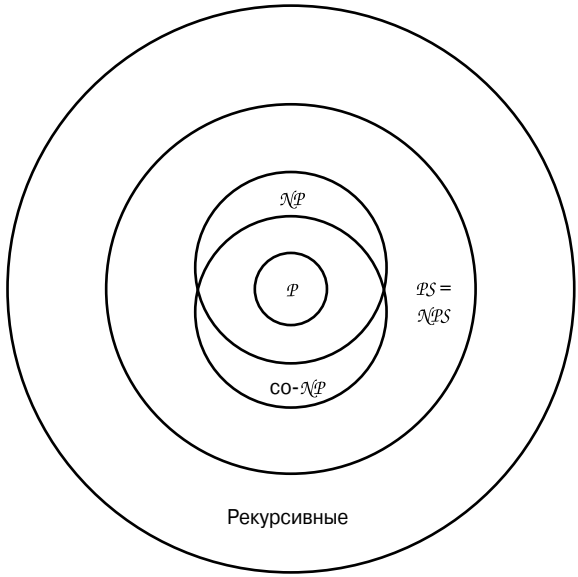


Рис. 11.5. Известные соотношения между классами языков

11.3.1. PS-полнота

Проблема P определяется как *полная для* $\mathcal{PS}$ (PS-полная), если выполняются следующие условия.

1. P принадлежит $\mathcal{PS}$.
2. Все языки L из $\mathcal{PS}$ полиномиально сводимы к P.

Отметим, что условия PS-полноты похожи на условия NP-полноты — сведение должно выполняться за полиномиальное время. Это позволило бы установить, что $\mathcal{P} = \mathcal{PS}$, если бы какая-нибудь PS-полная проблема оказалась в $\mathcal{P}$, и что $\mathcal{NP} = \mathcal{PS}$, если бы она оказалась в $\mathcal{NP}$. Если бы сведения были лишь в полиномиальном пространстве, то размер выхода мог бы быть экспоненциальным относительно размера входа, и невозможно было бы прийти к заключению следующей теоремы. Но, ограничившись сведениями, полиномиальными по времени, получаем желаемые взаимосвязи.

Теорема 11.6. Пусть P — PS-полная проблема. Тогда:

а) если P принадлежит $\mathcal{P}$, то $\mathcal{P} = \mathcal{PS}$;

б) если P принадлежит $\mathcal{NP}$, то $\mathcal{NP} = \mathcal{PS}$.

Доказательство. Докажем вариант *а*. Для любого L из $\mathcal{PS}$ известно, что существует полиномиальное (по времени) сведение L к P. Пусть оно занимает время $q(n)$. Предположим, P принадлежит $\mathcal{P}$ и имеет алгоритм с полиномиальным временем, например $p(n)$.

По данной цепочке w, принадлежность которой языку L мы хотим проверить, можно, используя сведение, получить цепочку x, находящуюся в P тогда и только тогда, когда w

принадлежит L. Поскольку сведение занимает время $q(|w|)$, цепочка x не может быть длиннее, чем $q(|w|)$. Принадлежность x языку P можно проверить за время $p(|x|)$, т.е. $p(q(|w|))$, полиномиальное относительно $|w|$. Приходим к выводу, что для L существует полиномиальный алгоритм.

Следовательно, каждый язык L из $\mathcal{PS}$ принадлежит $\mathcal{P}$. Поскольку включение $\mathcal{P}$ в $\mathcal{PS}$ очевидно, приходим к выводу, что если P принадлежит $\mathcal{P}$, то $\mathcal{P} = \mathcal{PS}$. Доказательство пункта *б*, где P принадлежит $\mathcal{NP}$, аналогично и оставляется читателю.

11.3.2. Булевы формулы с кванторами

Продемонстрируем проблему P, полную для $\mathcal{PS}$. Но сначала нужно изучить термины, в которых формулируется эта проблема, называемая "булевы формулы с кванторами" (quantified boolean formulas), или КБФ.

Грубо говоря, булева формула с кванторами — это булево выражение с добавлением операторов $\forall$ ("для всех") и $\exists$ ("существует"). Выражение $(\forall x)(E)$ означает, что E истинно, если все вхождения x заменить 1 (истина), а также, если все вхождения x заменить 0 (ложь). Выражение $(\exists x)(E)$ означает, что E истинно, если или все вхождения x заменить 1, или все вхождения x заменить 0, или в обоих случаях.

Для простоты описания предположим, что ни одна КБФ не содержит двух и более *квантификаций* ($\forall$ или $\exists$) одной и той же переменной x. Это ограничение не является существенным, и соответствует, грубо говоря, запрещению того, чтобы две различные функции в программе использовали одну и ту же локальную переменную[4]. Формально, *булевы формулы с кванторами* (*КБФ*) определяются следующим образом.

1. 0 (ложь), 1 (истина) или любая переменная являются КБФ.
2. Если E и F — КБФ, то КБФ являются (E), $\neg(E)$, $(E) \wedge (F)$ и $(E) \vee (F)$, представляя E в скобках, отрицание E, логическое **И** E и F и логическое **ИЛИ** E и F, соответственно. Скобки можно опускать, если они избыточны, используя обычные правила приоритета (старшинства): **НЕ**, затем **И**, затем **ИЛИ**. Часто используется "арифметический" стиль представления **И** и **ИЛИ**, когда **И** представлен непосредственным соседством (знак операции отсутствует), а **ИЛИ** — знаком +. Таким образом, вместо $(E) \wedge (F)$ часто используется $(E)(F)$, а вместо $(E) \vee (F)$ — $(E) + (F)$.
3. Если E — КБФ без квантификации переменной x, то $(\forall x)(E)$ и $(\exists x)(E)$ — КБФ. При этом говорят, что *областью действия* переменной x является выражение E. Неформально говоря, x определена только в пределах E, подобно тому, как областью дей-

[4] Два различных использования одного и того же имени переменной всегда можно переименовать, как в программах, так и в булевых формулах с кванторами. В программах нет нужды избегать повторного использования одного и того же локального имени, но в КБФ удобно предполагать, что повторных использований нет.

ствия (определения) переменной в программе является функция, в которой эта переменная определена. Скобки вокруг E (но не вокруг квантификации) можно удалить, если не возникает неоднозначности. Однако во избежание нагромождения вложенных скобок цепочка квантификаций вида

$(\forall x)((\exists y)((\forall z)(E)))$

записывается только с одной парой скобок вокруг E, а не с парой для каждой квантификации в цепочке, т.е. в виде $(\forall x)(\exists y)(\forall z)(E)$.

Пример 11.7. Рассмотрим пример КБФ:

$$(\forall x)((\exists y)(xy) + (\forall z)(\neg x + z)). \qquad (11.1)$$

Сначала переменные x и y связываются операцией **И**, затем применяется квантор $(\exists y)$ для получения подвыражения $(\exists y)(xy)$. Аналогично строится булево выражение $\neg x + z$ и применяется квантор $(\forall z)$ для получения подвыражения $(\forall z)(\neg x + z)$. Далее эти выражения становятся операндами **ИЛИ**; скобки в выражении не нужны, поскольку + (**ИЛИ**) имеет минимальный приоритет. Наконец, к этому выражению применяется квантор $(\forall x)$ и получается вся указанная КБФ. □

11.3.3. Вычисление булевых формул с кванторами

Определим формально значение КБФ. Интуитивную идею для этого дает прочтение $\forall$ как “для всех”, а $\exists$ — как “существует”. КБФ (11.1) утверждает, что для всех x (т.е. $x = 0$ или $x = 1$) существует y, при котором как x, так и y истинны, или же для всех z истинно $\neg x + z$. Это утверждение оказывается истинным. Если $x = 1$, то можно выбрать $y = 1$ и сделать xy истинным. Если же $x = 0$, то $\neg x + z$ истинно для обоих значений z.

Если переменная x находится в области действия некоторой квантификации x, то это вхождение x называется *связанным*, в противном случае — *свободным*.

Пример 11.8. Каждое использование переменной в КБФ (11.1) является связанным, поскольку находится в области действия квантора с этой переменной. Например, область действия переменной y, квантифицированной в $(\exists y)(xy)$, — это выражение xy. Таким образом, данное вхождение y связано. Вхождение x в xy также связано из-за квантора $(\forall x)$, областью действия которого является все выражение. □

Значением КБФ без свободных переменных является или 0, или 1 (ложь или истина, соответственно). Значение такой КБФ можно вычислить с помощью индукции по длине n выражения.

Базис. Если длина выражения 1, то оно может быть только константой 0 или 1 (любая переменная была бы свободной). Значением выражения является оно само.

Индукция. Пусть дана КБФ длиной $n > 1$ без свободных переменных, и можно вычислить любое выражение меньшей длины, если в нем нет свободных переменных. Возможны 6 видов такой КБФ.

1. (E). Тогда E имеет длину $n-2$, и значение E может быть вычислено как 0 или 1. Значение (E) совпадает с ним.
2. $\neg E$. Тогда E имеет длину $n-1$ и его значение можно вычислить. Если $E=1$, то $\neg E=0$, и наоборот.
3. EF. Выражения E и F короче n и могут быть вычислены. Значением EF будет 1, если оба выражения E и F имеют значение 1, и 0, если хотя бы одно из них равно 0.
4. $E+F$. Выражения E и F короче n и могут быть вычислены. Значением $E+F$ будет 1, если хотя бы одно из E и F имеет значение 1, и 0, если оба равны 0.
5. $(\forall x)(E)$. Все вхождения x в E заменяются значением 0 для получения выражения E_0, а также все вхождения x в E заменяются значением 1 для получения E_1. Заметим, что выражения E_0 и E_1:
   а) не имеют свободных переменных, поскольку любое вхождение свободной переменной отличалось бы от x и было бы свободным в E;
   б) имеют длину $n-6$, что меньше n.

   Вычисляются E_0 и E_1. Если у обоих значение 1, то $(\forall x)(E)$ имеет значение 1; в противном случае — 0. Отметим, каким образом это правило отражает интерпретацию $(\forall x)$ с помощью “для всех x”.
6. $(\exists x)(E)$. Как и в п. 5, строятся и вычисляются E_0 и E_1. Если хотя бы у одного из них значение 1, то значением $(\exists x)(E)$ будет 1; в противном случае — 0. Отметим, каким образом это правило отражает интерпретацию $(\exists x)$ с помощью “существует x”.

Пример 11.9. Вычислим КБФ (11.1). Она имеет вид $(\forall x)(E)$, поэтому сначала вычислим E_0:

$$(\exists y)(0y)+(\forall z)(\neg 0+z). \tag{11.2}$$

Значением этого выражения будет 1, если хотя бы один из операндов **ИЛИ** — $(\exists y)(0y)$ и $(\forall z)(\neg 0+z)$ — имеет значение 1. Для вычисления $(\exists y)(0y)$ нужно подставить $y=0$ и $y=1$ в подвыражение $0y$ и проверить, что хотя бы одно из двух получаемых выражений имеет значение 1. Однако и $0\wedge 0$, и $0\wedge 1$ имеют значение 0, поэтому значением $(\exists y)(0y)$ будет 0.[5]

К счастью, значением $(\forall z)(\neg 0+z)$ будет 1 — это видно при подстановке $z=0$ и $z=1$. Поскольку $\neg 0=1$, в этих двух случаях вычисляется $1\vee 0$ и $1\vee 1$, т.е. 1. Поэтому $(\forall z)(\neg 0+z)$ имеет значение 1. Итак, значением E_0, т.е. выражения (11.2), является 1.

Еще нужно также проверить, что выражение E_1 —

$$(\exists y)(1y)+(\forall z)(\neg 1+z), \tag{11.3}$$

[5] Отметим, что используется альтернативная запись **И** и **ИЛИ**, чтобы выражения с 0 и 1 не смотрелись как многоразрядные целые числа или арифметические выражения. Надеемся, читатель воспринимает обе нотации.

получаемое при подстановке $x = 1$ в (11.1), также имеет значение 1. Значением выражения $(\exists y)(1y)$ будет 1, что видно при подстановке $y = 1$. Таким образом, E_1, т.е. выражение (11.3), имеет значение 1, и значением всего выражения (11.1) является 1. □

11.3.4. PS-полнота проблемы КБФ

Теперь определим *проблему формулы с кванторами*: выяснить, имеет ли данная КБФ без свободных переменных значение 1. Эта проблема сокращенно обозначается КБФ, хотя КБФ продолжает применяться и как сокращение для термина "булева формула с кванторами". Контекст всегда позволит избежать двусмысленности.

Будет показано, что проблема КБФ полна для $\mathcal{PS}$. Доказательство сочетает идеи теорем 10.9 и 11.5. Из теоремы 10.9 берется идея представления вычисления МТ с помощью логических переменных, каждая из которых говорит, имеет ли определенная клетка определенное значение в определенный момент времени. Однако в теореме 10.9 речь шла о полиномиальном времени, поэтому там присутствовало полиномиальное количество переменных. Мы были в состоянии за полиномиальное время породить выражение, говорившее, что МТ допускала свой вход. Когда же речь заходит о полиномиальном пространстве, число МО в вычислении может быть экспоненциальным относительно размера входа, поэтому за полиномиальное время записать выражение, говорящее о корректности вычисления, невозможно. К счастью, теперь у нас есть более мощный язык, и возможность квантификации позволяет записать полиномиальную по длине КБФ, которая говорит, что МТ с полиномиально ограниченным пространством допускает свой вход.

Для выражения идеи того, что одно МО превращается в другое за некоторое большое число переходов, из теоремы 11.5 берется принцип "рекурсивного дублирования". Для того чтобы сказать, что МО I превращается в МО J за m переходов, утверждается, что существует МО K, получаемое из I за $m/2$ переходов и приводящее к J еще за $m/2$ переходов. Язык булевых формул с кванторами позволяет выражать такого рода факты в пределах полиномиальной длины, даже если m экспоненциально относительно длины входа.

Перед проведением доказательства, что каждый язык из $\mathcal{PS}$ полиномиально сводим к КБФ, нужно показать, что КБФ принадлежит $\mathcal{PS}$. Эта часть доказательства PS-полноты сама по себе сложна и выделяется в следующую теорему.

Теорема 11.10. КБФ принадлежит $\mathcal{PS}$.

Доказательство. В разделе 11.3.3 был описан рекурсивный процесс вычисления КБФ F. Этот алгоритм можно реализовать с использованием магазина, хранимого на ленте МТ, как в доказательстве теоремы 11.5. Пусть n — длина F. Тогда для F создается запись длиной $O(n)$, включающая саму F и пространство для записи обрабатываемых подвыражений F. Процесс вычисления объясняется для двух из шести возможных вариантов выражения F.

1. Пусть $F = F_1 + F_2$. Тогда выполняем следующее:

а) помещаем F_1 в ее собственную запись справа от записи для F;

б) рекурсивно вычисляем F_1;

в) если значением F_1 является 1, то возвращаем 1 как значение F;

г) если значение F_1 — 0, то ее запись замещаем записью для F_2 и рекурсивно вычисляем F_2;

д) в качестве значения F возвращаем значение F_2.

2. Пусть $F = (\exists x)(E)$. Тогда выполняем следующее:

а) создаем выражение E_0 путем подстановки 0 вместо каждого вхождения x и помещаем E_0 в собственную запись справа от записи для F;

б) рекурсивно вычисляем E_0;

в) если значением E_0 является 1, то возвращаем 1 как значение F;

г) если значение E_0 — 0, то создаем выражение E_1, подставляя 1 вместо x в E;

д) запись для E_0 замещаем записью для E_1 и рекурсивно вычисляем E_1;

е) в качестве значения F возвращаем значение E_1.

Описание подобных шагов вычисления F в ее остальных четырех формах — F_1F_2, $\neg E$, (E), $(\forall x)(E)$ — предоставляется читателю. Базисный случай, когда формула является константой, требует лишь возвращения этой константы без создания записей на ленте.

Заметим, что в любом случае справа от записи для выражения, длина которого m, присутствует запись для выражения меньшей длины. Отметим, что, хотя в случае 1 вычисляются два различных подвыражения F_1 и F_2, это делается последовательно. Таким образом, записи для F_1 и его подвыражений и записи для F_2 и его подвыражений не присутствуют на ленте одновременно. То же верно и для E_0 и E_1 в п. 2.

Следовательно, если мы начинаем с выражения длиной n, в магазине не может быть более n записей. Каждая запись имеет длину $O(n)$. Поэтому размер ленты не превышает $O(n^2)$. Теперь у нас есть конструкция для МТ с полиномиально ограниченным пространством, допускающей КБФ; предел ее пространства является квадратичным. Заметим, что время работы этого алгоритма обычно экспоненциально относительно n, поэтому он *не* полиномиален по времени. □

Обратимся к сведению произвольного языка L из $\mathcal{PS}$ к проблеме КБФ. Нам хотелось бы использовать пропозициональные переменные y_{ijA}, как в теореме 10.9, для утверждения, что в j-й позиции i-го МО находится символ A. Однако, поскольку МО экспоненциальное число, нельзя взять вход длиной n и даже просто выписать эти переменные за время, полиномиальное относительно n. Воспользуемся квантификацией, чтобы с помощью одного и того же множества переменных представлять много различных МО. Эта идея раскрывается в доказательстве следующей теоремы.

Теорема 11.11. Проблема КБФ PS-полна.

Доказательство. Пусть L — язык из $\mathcal{PS}$, допускаемый недетерминированной МТ M, которая при обработке входа длиной n использует не более $p(n)$ клеток. По теореме 11.3 существует константа c, для которой M допускает вход длиной n в пределах $c^{1+p(n)}$ переходов (если допускает). Опишем, как за полиномиальное время по входу w длиной n построить КБФ E без свободных переменных, имеющую значение 1 тогда и только тогда, когда w принадлежит $L(M)$.

При записи E нам понадобится ввести полиномиальное число *переменных МО*, которые представляют собой множества переменных y_{jA}, утверждающих, что j-я позиция представляемого МО содержит символ A (j может изменяться от 0 до $p(n)$). A есть либо ленточный символ, либо состояние M. Таким образом, число пропозициональных переменных в переменном МО полиномиально относительно n. Предположим, что все пропозициональные переменные в разных переменных МО различимы, т.е. ни одна из них не принадлежит двум разным переменным МО. Поскольку существует лишь полиномиальное число переменных МО, общее количество пропозициональных переменных полиномиально.

Удобно ввести нотацию $(\exists I)$, где I — переменное МО. Этот квантор записывается вместо $(\exists x_1)(\exists x_2)\ldots(\exists x_m)$, где $x_1, x_2, \ldots, x_m$ — все пропозициональные переменные в переменном МО I. Аналогично вместо применения квантора $\forall$ ко всем пропозициональным переменным в I записывается $(\forall I)$.

КБФ, которая строится для w, имеет вид

$$(\exists I_0)(\exists I_f)(S \wedge N \wedge F).$$

Подвыражения этой формулы имеют следующий смысл.

1. I_0 и I_1 — переменные МО, представляющие начальное и допускающее МО, соответственно.
2. S — выражение, говорящее о “правильном старте”, т.е. что I_0 действительно является начальным МО M с w на входе.
3. N — выражение, которое говорит о “правильных переходах”, совершаемых M при преобразовании I_0 к I_1.
4. F — выражение, говорящее о “правильном финише”, т.е. что I_f является допускающим МО.

Отметим, что хотя выражение в целом не имеет свободных переменных, переменные из I_0 будут появляться как свободные в S, переменные из I_f — как свободные в F, а обе группы переменных будут свободны в N.

Правильный старт

S является логическим **И** литералов; каждый литерал — это одна из переменных МО I_0. S имеет литерал y_{jA}, если в j-й позиции начального МО со входом w находится символ A, и литерал $\bar{y}_{jA}$, если нет. Таким образом, если $w = a_1a_2\ldots a_n$, то $y_{0q_0}, y_{1a_1}, y_{2a_2}, \ldots, y_{na_n}$ и

все y_{jB} для $j = n + 1, n + 2, \ldots, p(n)$ появляются без отрицания, а все остальные переменные МО I_0 — с отрицаниями. Здесь предполагается, что q_0 — начальное состояние M, а B — пробел.

Правильный финиш

I_f является допускающим МО, если содержит допускающее состояние. Следовательно, F записывается как логическое **ИЛИ** тех переменных y_{jA}, выбранных из пропозициональных переменных МО I_f, для которых A является допускающим состоянием. Позиция j произвольна.

Правильные переходы

Выражение N строится рекурсивно с помощью метода, который позволяет удвоить число рассматриваемых переходов, добавив лишь $O(p(n))$ символов в конструируемое выражение и (что важнее) затратив для написания выражения время $O(p(n))$. Для логического **И** выражений, в которых приравниваются соответствующие переменные МО I и J, полезно использовать сокращение $I = J$. Таким образом, если I состоит из переменных y_{jA} и J состоит из переменных z_{jA}, то $I = J$ — это **И** выражений $(y_{jA} z_{jA} + \bar{y}_{jA}\ \bar{z}_{jA})$, где j изменяется от 0 до $p(n)$, а A — любой ленточный символ или состояние M.

Теперь для обозначения того, что $I \overset{*}{\vdash} J$ за i или менее переходов, построим выражения $N_i(I, J)$, где $i = 1, 2, 4, 8, \ldots$. В этих выражениях свободны только пропозициональные переменные переменных МО I и J; все остальные пропозициональные переменные связаны.

Такое построение N_{2i} не работает

Первым инстинктивным побуждением, связанным с построением N_{2i} по N_i, может быть непосредственное применение подхода “разделяй и властвуй”: если $I \overset{*}{\vdash} J$ за $2i$ или менее переходов, то должно существовать МО K, для которого $I \overset{*}{\vdash} K$ и $K \overset{*}{\vdash} J$ за i или менее переходов. Однако, если записать формулу, которая выражает эту идею, например, $N_{2i}(I, J) = (\exists K)(N_i(I, K) \wedge N_i(K, J))$, то длина выражения удвоится при удвоении i. Чтобы выразить все возможные вычисления M, i должно быть экспоненциальным относительно n, поэтому для написания N будет затрачено слишком много времени, и N будет иметь экспоненциальную длину.

Базис. Для $i = 1$ выражение $N_i(I, J)$ устанавливает, что $I = J$ или $I \vdash J$. Мы только что обсудили, как выразить условие $I = J$. Для условия $I \vdash J$ сошлемся на часть “правильные переходы” из доказательства теоремы 10.9, где также возникала проблема утверждения, что очередное МО следует из предыдущего. Выражение N_1 является логическим **ИЛИ** этих двух выражений. Заметим, что оно записывается за время $O(p(n))$.

Индукция. По N_i построим $N_{2i}(I, J)$. Во врезке “Такое построение N_{2i} не работает” отмечается, что прямой метод построения N_{2i} с помощью двух копий N_i не дает нужного времени и пространства. Корректный способ записи N_{2i} состоит в том, чтобы в выраже-

нии записывать одну копию N_i, подставляя как (I, K), так и (K, J) в одно и то же выражение. Таким образом, в $N_{2i}(I, J)$ используется одно подвыражение $N_i(P, Q)$. $N_{2i}(I, J)$ записывается для утверждения, что существует МО K, при котором для всех МО P и Q выполняется хотя бы одно из следующих условий.

1. $(P, Q) \neq (I, K)$ и $(P, Q) \neq (K, J)$.
2. $N_i(P, Q)$ истинно.

Иными словами, $N_i(I, K)$ и $N_i(K, J)$ истинны, а для других пар МО (P, Q) истинность $N_i(P, Q)$ не имеет значения. Итак, КБФ для $N_{2i}(I, J)$ имеет следующий вид.

$$N_{2i}(I, J) = (\exists K)(\forall P)(\forall Q)(N_i(P, Q) \vee (\neg(I = P \wedge K = Q) \wedge \neg(K = P \wedge J = Q)))$$

Отметим, что на запись N_{2i} уходит время, необходимое для записи N_i, а также $O(p(n))$ для дополнительной работы.

Чтобы завершить построение N, нужно записать N_m для наименьшего m, которое является степенью 2 и не меньше $c^{1+p(n)}$ — максимально возможного числа переходов, совершаемых МТ M перед тем, как допустить вход w длиной n. Количество применений шага индукции, описанного выше, равно $\log_2(c^{1+p(n)})$, или $O(p(n))$. Поскольку каждое использование шага индукции занимает время $O(p(n))$, приходим к выводу, что N можно построить за время $O(p^2(n))$.

Завершение доказательства теоремы 11.11

Выше показано, как преобразовать вход w в КБФ

$$(\exists I_0)(\exists I_f)(S \wedge N \wedge F)$$

за время, полиномиальное относительно $|w|$. Обосновано также, почему выражения S, N и F истинны тогда и только тогда, когда их свободные переменные представляют МО I_0 и I_f, которые являются начальным и заключительным МО в вычислении M со входом w, причем $I_0 \overset{*}{\vdash} I_f$. Таким образом, данная КБФ имеет значение 1 тогда и только тогда, когда M допускает w. □

11.3.5. Упражнения к разделу 11.3

11.3.1. Дополните доказательство теоремы 11.10, рассмотрев варианты:

а) $F = F_1 F_2$;

б) $F = (\forall x)(E)$;

в) $F = \neg(E)$;

г) $F = (E)$.

11.3.2. (***!!**) Докажите, что следующая проблема является PS-полной. По данному регулярному выражению E определить, эквивалентно ли оно Σ^*, где Σ — множество символов, встречающихся в E. *Указание*. Вместо сведения КБФ к данной проблеме можно показать, что любой язык из $\mathcal{PS}$ сводится к ней. Для каждой МТ M

с полиномиально ограниченным пространством покажите, как взять вход w для M и построить за полиномиальное время регулярное выражение, порождающее все цепочки, которые *не* являются последовательностями МО машины M, ведущими к допусканию w.

11.3.3. (!!) *Переключательная игра Шеннона* состоит в следующем. Дается граф G с двумя терминальными узлами s и t. Есть два игрока, называемых SHORT и CUT. По очереди каждый игрок выбирает узел графа G, не равный s и t, который до конца игры будет принадлежать этому игроку. Игру начинает SHORT. Он выигрывает, если выбирает множество узлов, которое вместе с s и t образует путь в графе G из s в t. CUT выигрывает, если все узлы выбраны, но SHORT не выбрал путь в графе G из s в t. Покажите PS-полноту проблемы: по данному графу G определить, может ли SHORT выиграть независимо от ходов CUT.

11.4. Классы языков, основанные на рандомизации

Теперь обратимся к двум классам языков, определяемых машинами Тьюринга, способными при вычислениях использовать случайные числа. Возможно, читатель знаком с алгоритмами на обычных языках программирования, использующими генератор случайных чисел. Функция с названием, подобным `rand()`, возвращающая число, которое кажется "случайным" или непредсказуемым, в действительности выполняет специальный алгоритм. Его можно проимитировать, хотя в порождаемой им последовательности чисел очень трудно увидеть закономерность. Простой пример такой функции (не используемый на практике) — взять предыдущее число последовательности, возвести его в квадрат и взять средние биты этого квадрата. Числа, порождаемые сложным механическим процессом подобного рода, называются *псевдослучайными*.

В этом разделе определяется тип машины Тьюринга, моделирующей генерацию случайных чисел и их использование в алгоритмах. Далее определяются два класса языков, $\mathcal{RP}$ и $\mathcal{ZPP}$, использующих эту случайность и полиномиальное время различными способами. Может показаться, что эти классы содержат совсем немного проблем вне $\mathcal{P}$, однако их отличие от $\mathcal{P}$ весьма важно. В частности, в разделе 11.5 будет показано, почему некоторые наиболее существенные проблемы, связанные с безопасностью компьютеров, в действительности являются вопросами о соотношении этих классов с классами $\mathcal{P}$ и $\mathcal{NP}$.

11.4.1. Быстрая сортировка — пример рандомизированного алгоритма

Возможно, читатель знаком с алгоритмом сортировки, который называется "Быстрая сортировка" ("Quicksort"). Сущность алгоритма такова. Из сортируемого списка элементов $a_1, a_2, \ldots, a_n$ выбирается один, скажем, a_1, и элементы списка делятся на те, которые меньше или равны a_1, и на те, которые больше a_1. Выбираемый элемент называется *ве-*

дущим (*pivot*). Тщательный подбор представления данных позволяет разделить список длиной n на два за время $O(n)$. Далее можно рекурсивно отсортировать по отдельности список нижних (которые меньше или равны ведущему) и список верхних (больше ведущего) элементов и в результате получить отсортированный список из всех n элементов.

Если нам повезет, то ведущий элемент окажется числом в середине сортируемого списка, и оба подсписка будут иметь длину примерно $n/2$. Если нам повезет на каждом рекурсивном шаге, то после примерно $\log_2 n$ уровней рекурсии у нас будут уже отсортированные списки длиной 1. Таким образом, каждый из $O(\log n)$ уровней требует $O(n)$ времени, а вся работа — $O(n \log n)$.

Однако нам может не повезти. Например, если список изначально отсортирован, то выбор первого элемента в каждом списке делит его на нижний подсписок с одним этим элементом и верхний — со всеми остальными. В данном случае быстрая сортировка ведет себя, как сортировка выбором, и при упорядочении n элементов занимает время, пропорциональное n^2.

Таким образом, хорошие реализации быстрой сортировки не выбирают механически никаких определенных позиций в списке для ведущих элементов. Ведущий элемент выбирается в списке случайно, т.е. вероятность выбора каждого из n элементов в качестве ведущего равна $1/n$. Ожидаемое время выполнения быстрой сортировки с использованием такой рандомизации равно $O(n \log n)$, хотя данное утверждение здесь не доказывается.[6] Однако из-за ненулевого шанса того, что каждый ведущий элемент окажется наибольшим или наименьшим, время выполнения быстрой сортировки в худшем случае остается $O(n^2)$. Тем не менее, быстрая сортировка является основным методом сортировки во многих приложениях (например, в UNIX), поскольку ожидаемое время ее выполнения в действительности гораздо меньше, чем у других методов, имеющих $O(n \log n)$ в худшем случае.

11.4.2. Вариант машины Тьюринга с использованием рандомизации

Для того чтобы абстрактно представить способность машин Тьюринга к совершению случайного выбора, похожего на вызов генератора случайных чисел в программе, используем вариант многоленточной МТ, изображенный на рис. 11.6. Первая лента, как обычно для многоленточных машин, содержит вход. Вторая лента также начинается непустыми клетками. В принципе, вся она содержит символы 0 и 1, выбранные с вероятностью 1/2. Вторая лента называется *случайной лентой*. Третья и последующие, если используются, вначале пусты и при необходимости выступают как рабочие. Данный вариант МТ называется *рандомизированной машиной Тьюринга*.

---

[6] Анализ и обоснование ожидаемого времени выполнения быстрой сортировки можно найти в следующих изданиях. D. E. Knuth, *The Art of Computer Programming, Vol. III: Sorting and Searching*, Addison-Wesley, 1973. (Кнут Д. Искусство программирования для ЭВМ. В 3 т. Т. 3: Поиск и сор-

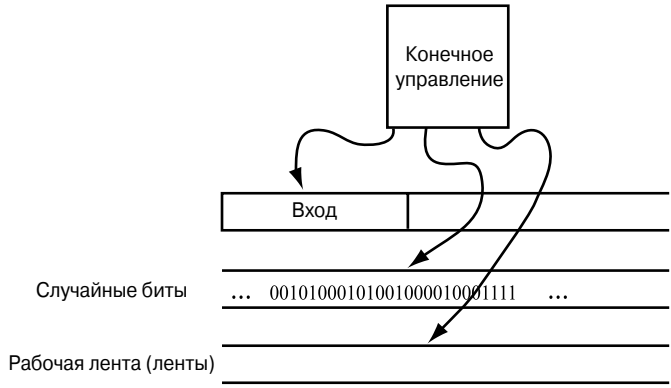


Рис. 11.6. Машина Тьюринга, использующая случайно "генерируемые" числа

Поскольку нереалистично считать, что бесконечную ленту рандомизированной МТ можно вначале случайно заполнить символами 0 и 1, эквивалентный взгляд на такую МТ состоит в том, что ее лента изначально пуста. Однако далее, когда вторая головка обозревает пробел, происходит встроенное "бросание монеты", и рандомизированная МТ немедленно записывает символ 0 или 1 в обозреваемую клетку, не изменяя его в дальнейшем. При таком способе отсутствует бесконечная работа, которую нужно было бы совершить перед началом работы рандомизированной МТ. Тем не менее, вторая лента оказывается покрытой случайными символами 0 и 1, поскольку эти случайные биты появляются везде, где в действительности побывала головка второй ленты МТ.

Пример 11.12. Рандомизированную версию быстрой сортировки можно реализовать с помощью МТ. Важным шагом является следующий рекурсивный процесс. Предположим, что подсписок сохранен в последовательных клетках входной ленты и выделен маркерами с обоих концов. В подсписке выбирается ведущий элемент, и подсписок делится на нижний и верхний подподсписки. Рандомизированная МТ выполняет такие действия.

1. Пусть разделяемый подсписок имеет длину m. Используем до $\log_2 m$ новых случайных битов на второй ленте, чтобы выбрать случайное число между 1 и m; m-й элемент подсписка становится ведущим. Отметим, что, возможно, вероятности выбора чисел от 1 до m не равны, поскольку m может не быть степенью 2. Однако, если взять, например, $\lceil 2 \log_2 m \rceil$ битов с ленты 2, рассмотреть их как число в диапазоне от 0 до m^2, взять остаток от деления на m и прибавить 1, то все числа от 1 до m будут иметь вероятность, достаточно близкую к $1/m$, чтобы быстрая сортировка выполнялась корректно.

тировка. — М.: Мир, 1976. См. также: Кнут Д. Искусство программирования. В 3 т. Т. 3: Поиск и сортировка. — М.: Издательский дом "Вильямс", 2000.)

2. Поместить ведущий элемент на ленту 3.
3. Просмотреть список, выделенный на ленте 1, копируя элементы, которые не больше ведущего, на ленту 4.
4. Снова просмотреть подсписок на ленте 1, копируя элементы, которые больше ведущего, на ленту 5.
5. Скопировать ленты 4 и затем 5 в пространство на ленте 1, ранее занятое выделенным подсписком. Поместить маркер между двумя подподсписками.
6. Если подподсписки (хотя бы один из них) содержат более одного элемента, рекурсивно отсортировать их по этому же алгоритму.

Заметим, что данная реализация быстрой сортировки требует $O(n \log n)$ времени, хотя вычислительным устройством является МТ, а не обычный компьютер. Однако главное в этом примере — не время работы, а использование случайных битов на второй ленте для организации случайного поведения машины Тьюринга. □

11.4.3. Язык рандомизированной машины Тьюринга

Нам привычна ситуация, в которой машина Тьюринга (в частности, КА или МП-автомат) допускает некоторый язык, даже если он пуст или совпадает со всем множеством цепочек во входном алфавите. Имея дело с рандомизированными МТ, нужно быть более аккуратным с тем, что значит допускание входа такой машиной; становится возможным, что МТ вообще не допускает никакого языка. Проблема в том, что при анализе действий рандомизированной МТ M со входом w приходится рассматривать все возможные случайные последовательности на второй ленте. Вполне возможно, что МТ допускает при одних случайных последовательностях, но отвергает при других; в действительности, если рандомизированная МТ должна делать что-то более эффективно, чем детерминированная МТ, то существенно, чтобы различные последовательности на рандомизированной ленте приводили к различному поведению.[7]

Если считать, что рандомизированная МТ допускает, достигая, как обычная МТ, заключительного состояния, то каждый вход w рандомизированной МТ имеет некоторую вероятность допускания, зависящую от содержимого случайной ленты, приводящего к допусканию. Поскольку экземпляров такого содержимого бесконечно много, при вычислении этой вероятности нужно быть осторожным. Вместе с тем, в любой последовательности переходов, приводящей к допусканию, используется лишь конечная часть случайной ленты, поэтому вероятность любой случайной последовательности равна 2^{-m}, если

[7] Подчеркнем, что рандомизированная МТ из примера 11.12 не является распознающей. Она преобразовывает вход, и от того, что было на случайной ленте, зависит время выполнения этого преобразования, а не его результат.

m — число клеток случайной ленты, когда-либо просмотренных и повлиявших на переходы МТ. Следующий пример иллюстрирует вычисления в одном очень простом случае.

Пример 11.13. Функция переходов рандомизированной МТ M представлена на рис. 11.7. M использует только входную и случайную ленты. Она ведет себя очень просто, не изменяя ни одного символа на лентах и сдвигая головки только вправо (направление R) или оставляя на месте (S). Хотя формально запись переходов рандомизированной МТ не была определена, содержимое таблицы на рис. 11.7 должно быть понятно. Каждая строка таблицы соответствует состоянию, а каждая колонка — паре символов XY, где X — символ, обозреваемый на входной ленте, а Y — на случайной. Клетка $qUVDE$ таблицы означает, что МТ переходит в состояние q, записывает U на входной ленте, V — на случайной, сдвигает головку на входной ленте в направлении D, а на случайной — в направлении E.

	00	01	10	11	$B0$	$B1$
$\rightarrow q_0$	$q_1 00RS$	$q_3 01SR$	$q_2 10RS$	$q_3 11SR$		
q_1	$q_1 00RS$				$q_4 B0SS$	
q_2			$q_2 10RS$		$q_4 B0SS$	
q_3	$q_3 00RR$			$q_3 11RR$	$q_4 B0SS$	$q_4 B1SS$
$*q_4$						

Рис. 11.7. Функция переходов рандомизированной машины Тьюринга

Опишем вкратце поведение M при входной цепочке w, состоящей из символов 0 и 1. В начальном состоянии q_0 машина M обозревает первый случайный бит и в зависимости от его значения (0 или 1) выполняет одну из двух проверок, связанных с w.

Если случайный бит равен 0, то M проверяет, состоит ли w только из символов 0 или только из символов 1. В этом случае M больше не смотрит на случайные биты и оставляет вторую ленту без изменений. Если первый бит w равен 0, то M переходит в состояние q_1. В этом состоянии она движется вправо через нули, но останавливается, не допуская, если видит 1. Если в этом состоянии она достигает пробела на входной ленте, то переходит в допускающее состояние q_4. Аналогично, если первый бит w равен 1, и первый случайный бит равен 0, то M переходит в состояние q_2; в нем она проверяет, что все биты w равны 1, и допускает, если это так.

Теперь рассмотрим, что делает M, если первый случайный бит равен 1. Она сравнивает w со вторым и последующими случайными битами, допуская только тогда, когда они совпадают с первым и последующими битами w, соответственно. Таким образом, в состоянии q_0, обозревая 1 на второй ленте, M переходит в состояние q_3. Отметим, что при этом она сдвигает вправо головку на случайной ленте, оставляя на месте головку на входной. Далее в состоянии q_3 она проверяет совпадение содержимого двух лент, сдви-

гая обе головки вправо. Если в некоторой позиции она находит несовпадение, то останавливается без допускания, а если достигает пробела на входной ленте, то допускает.

Вычислим вероятность допускания определенных входов. Сначала рассмотрим однородный вход, в котором встречается только один символ, например, 0^i, где $i \geq 1$. С вероятностью 1/2 первый случайный бит равен 0, и если так, то дальнейшая проверка однородности будет успешной, и 0^i допускается. Однако с той же вероятностью 1/2 первый бит равен 1. В этом случае 0^i допускается тогда и только тогда, когда все случайные биты со второго по $(n + 1)$-й равны 0. Это возможно с вероятностью 2^{-i}. Итак, общая вероятность допускания 0^i равна

$$\frac{1}{2}+\frac{1}{2}2^{-i}=\frac{1}{2}+2^{-(i+1)}.$$

Теперь рассмотрим вариант неоднородного входа w, содержащего как нули, так и единицы, например 00101. Этот вход не допускается, если первый случайный бит равен 0. Если же первый бит равен 1, то вероятность допускания составляет 2^{-i}, где i — длина входа. Таким образом, общая вероятность допускания неоднородной цепочки длиной i равна $2^{-(i+1)}$. Например, вероятность допускания 00101 — 1/64. □

Наш вывод состоит в том, что вероятность допускания любой цепочки данной рандомизированной МТ можно вычислить. Принадлежность цепочки языку зависит от того, как определено "членство" в языке рандомизированной МТ. В следующем разделе даются два разных определения допускания, приводящие к различным классам языков.

11.4.4. Класс $\mathcal{RP}$

Язык L класса $\mathcal{RP}$ ("random polynomial" — случайные полиномиальные) допускается рандомизированной МТ M в следующем смысле.

1. Если w не принадлежит L, то вероятность того, что M допускает w, равна 0.
2. Если w принадлежит L, то вероятность того, что M допускает w, не меньше 1/2.
3. Существует полином $p(n)$, для которого, если w имеет длину n, то все вычисления M, независимо от содержимого случайной ленты, останавливаются после не более $p(n)$ шагов.

Заметим, что определение класса $\mathcal{RP}$ использует два не связанных между собой свойства. Пункты 1 и 2 определяют рандомизированную МТ специального вида, которую иногда называют алгоритмом *типа Монте-Карло*. Таким образом, независимо от времени работы говорят, что рандомизированная МТ является машиной "типа Монте-Карло", если она допускает или с вероятностью 0, или с вероятностью больше 1/2, ничего не допуская с вероятностями между 0 и 1/2. В пункте 3 упоминается время работы, не зависящее от того, является ли МТ машиной типа Монте-Карло.

Пример 11.14. Рассмотрим рандомизированную МТ из примера 11.13. Она удовлетворяет условию 3, поскольку время ее работы есть $O(n)$ независимо от содержимого

случайной ленты. Однако она вообще не допускает никакого языка в смысле определения $\mathcal{RP}$. Причина в том, что, хотя однородные цепочки вроде 000 допускаются с вероятностью не меньше 1/2 и, таким образом, удовлетворяют условию 2, есть другие цепочки, вроде 001, допускаемые с вероятностью, не равной 0 и меньшей, чем 1/2 (цепочка 001 допускается с вероятностью 1/16). □

Пример 11.15. Неформально опишем рандомизированную МТ, которая одновременно полиномиальна по времени и является машиной типа Монте-Карло и, следовательно, допускает язык из $\mathcal{RP}$. Ее вход интерпретируется как граф, и вопрос состоит в том, есть ли в этом графе треугольник, т.е. три узла, попарно соединенных ребрами. Входы с треугольниками принадлежат языку, остальные — нет.

Алгоритм Монте-Карло циклически выбирает ребро (x, y) и вершину z, отличную от x и y, случайным образом. Каждый выбор определяется просмотром нескольких новых случайных битов на случайной ленте. Для каждой тройки выбранных x, y и z МТ проверяет, содержит ли вход ребра (x, z) и (y, z), и, если так, объявляет, что вход содержит треугольник.

Всего производится k выборов ребра и вершины; МТ допускает, если любой из них дает треугольник, а если нет, не допускает. Если у графа нет треугольника, то ни один из k выборов не может показать его наличие, что соответствует условию 1 в определении $\mathcal{RP}$ — если вход не принадлежит языку, то вероятность его допускания равна 0.

Предположим, что граф имеет n узлов и m ребер. Если граф имеет хотя бы один треугольник, то вероятность того, что три его узла будут выбраны в одном эксперименте, равна $\left(\frac{3}{m}\right)\left(\frac{1}{n-2}\right)$, т.е. три из m ребер находятся в треугольнике, и если любое из них выбрано, то вероятность того, что выбирается также и третий узел, равна $1/(n-2)$. Эта вероятность мала, но эксперимент повторяется k раз. Поэтому вероятность того, что ни один из k экспериментов не даст треугольника, равна

$$1-\left(1-\frac{3}{m(n-2)}\right)^k. \tag{11.4}$$

Для величины $(1-x)^k$ при малых x часто используется приближение в виде e^{-kx}, где $e = 2.718\ldots$ — основание натуральных логарифмов. Если выбрать k так, что, например $kx = 1$, то e^{-kx} будет заметно меньше 1/2 и $1 - e^{-kx}$ будет значительно больше 1/2 (около 0.63). Таким образом, можно выбрать $k = m(n-2)/3$, чтобы гарантировать, что вероятность допускания графа с треугольником, описанная формулой (11.4), не меньше 1/2. Итак, описанный алгоритм является алгоритмом типа Монте-Карло.

Нужно еще рассмотреть время работы МТ. И n, и m не больше, чем длина входа, а значение k выбирается так, что оно не больше квадрата длины (пропорционально произведению n и m). В каждом эксперименте вход просматривается не более четырех раз

(чтобы выбрать случайное ребро и узел, а затем проверить наличие еще двух ребер), поэтому время выполнения эксперимента линейно относительно длины входа. Таким образом, МТ останавливается, совершив переходов в количестве, не более чем кубическом относительно длины входа, т.е. МТ имеет полиномиальное время работы и, следовательно, удовлетворяет условию 3 определения принадлежности языка классу $\mathcal{RP}$.

Приходим к выводу, что язык графов с треугольниками принадлежит классу $\mathcal{RP}$. Отметим, что он также находится в $\mathcal{P}$, поскольку можно провести систематический полиномиальный поиск всех треугольников. Однако, как упоминалось в начале раздела 11.4, найти примеры языков, которые оказались бы в $\mathcal{RP} - \mathcal{P}$, в действительности трудно. □

11.4.5. Распознавание языков из $\mathcal{RP}$

Предположим, что для распознавания языка L у нас есть полиномиальная по времени машина Тьюринга M типа Монте-Карло. Нам дается цепочка w, и нужно узнать, принадлежит ли w языку L. Запуская M на w и используя бросание монеты или какое-либо другое устройство генерации случайных чисел для имитации создания случайных битов, получаем следующее.

1. Если w не принадлежит L, то запуск наверняка не завершится допусканием w.
2. Если w принадлежит L, то существует не менее 50% шансов, что w будет допущено.

Однако если мы просто возьмем выход данного запуска в качестве определяющего, то w может оказаться отвергнутым, хотя должно быть допущено (*ложный негативный* исход, ложный пропуск), но мы никогда не допустим его, если не должны (*ложный позитивный* исход, ложное допускание). Таким образом, следует отличать рандомизированную МТ от алгоритма, используемого для решения, находится ли w в L. В целом избежать ложных негативных исходов невозможно, хотя путем многократного повторения проверки вероятность ложного пропуска можно сделать как угодно малой.

Например, если нужно сделать вероятность ложного пропуска не больше одной биллионной, можно запустить проверку тридцать раз. Если w принадлежит L, то шансы на то, что все тридцать проверок пропустят допускание, не больше 2^{-30}, что меньше 10^{-9}, или одной биллионной. Вообще, если нам нужна вероятность ложного пропуска меньше, чем $c > 0$, мы должны запустить проверку $\log_2(1/c)$ раз. Это количество является константой, если c — константа, а один запуск рандомизированной МТ M требует полиномиального времени, так как предполагается, что L принадлежит $\mathcal{RP}$. Отсюда повторение проверки также требует полиномиального времени. Вывод из этих рассуждений формулируется в следующей теореме.

Теорема 11.16. Если L принадлежит $\mathcal{RP}$, то для любой как угодно малой константы $c > 0$ существует полиномиальный по времени рандомизированный алгоритм, решающий, принадлежит ли w языку L, который не совершает ложных допусканий, а ложные пропуски делает с вероятностью не больше c. □

11.4.6. Класс $\mathcal{ZPP}$

Второй класс языков, использующих рандомизацию, называется *безошибочным вероятностным полиномиальным* (*zero-error, probabilistic, polynomial*), или $\mathcal{ZPP}$. Класс основан на рандомизированной МТ, которая всегда останавливается и имеет ожидаемое время останова, полиномиальное относительно длины входа. Эта МТ допускает свой вход, если попадает в допускающее состояние (и при этом останавливается), и отвергает его, останавливаясь без допускания. Таким образом, определение класса $\mathcal{ZPP}$ почти совпадает с определением класса $\mathcal{P}$, за исключением того, что $\mathcal{ZPP}$ разрешает машине вести себя случайным образом, и ограничивается не время работы в худшем случае, а ожидаемое время работы.

МТ, которая всегда дает правильный ответ, но время работы которой зависит от значений некоторых случайных битов, иногда называется машиной Тьюринга типа *Лас-Вегас*, или алгоритмом типа Лас-Вегас. Таким образом, класс $\mathcal{ZPP}$ можно считать классом языков, допускаемых машинами Тьюринга типа Лас-Вегас с полиномиально ограниченным ожидаемым временем работы.

Является ли дробь 1/2 особенной в определении $\mathcal{RP}$?

Хотя в определении $\mathcal{RP}$ требовалось, чтобы вероятность допускания цепочки w из L была не меньше 1/2, можно определить класс $\mathcal{RP}$ с любой другой константой между 0 и 1 вместо 1/2. Теорема 11.16 говорит, что мы могли бы, повторяя эксперимент, совершаемый M, подходящее число раз, сделать вероятность допускания сколь угодно большой, но строго меньшей 1. Кроме того, такая же техника уменьшения вероятности пропуска цепочки из L, использованная в разделе 11.4.5, позволит нам брать рандомизированную МТ с любой вероятностью допускания цепочки w из L, превышающей 0, и увеличивать эту вероятность до 1/2 путем повторения экспериментов некоторое постоянное число раз.

В определении $\mathcal{RP}$ мы продолжим требовать 1/2 в качестве вероятности допускания, но осознавая, что для определения $\mathcal{RP}$ достаточно любой ненулевой вероятности. С другой стороны, изменение константы 1/2 изменит язык, определяемый конкретной рандомизированной МТ. Так, из примера 11.14 следует, что снижение требуемой вероятности до 1/16 приведет к тому, что цепочка 001 окажется в языке рандомизированной МТ, описанной там.

11.4.7. Соотношение между $\mathcal{RP}$ и $\mathcal{ZPP}$

Между двумя определенными выше рандомизированными классами есть простое соотношение. Для того чтобы сформулировать теорему о нем, нужно сначала рассмотреть

дополнения этих классов. Очевидно, если L принадлежит $\mathcal{ZPP}$, то $\overline{L}$ тоже принадлежит $\mathcal{ZPP}$. Причина в том, что если L допускается МТ M типа Лас-Вегас с полиномиально ограниченным ожидаемым временем, то $\overline{L}$ допускается модификацией M, в которой допускание превращено в останов без допускания, и наоборот.

Однако замкнутость $\mathcal{RP}$ относительно дополнения не очевидна, поскольку определение машины типа Монте-Карло трактует допускание и отвергание несимметрично. Таким образом, определим класс со-$\mathcal{RP}$ как множество языков L, для которых $\overline{L}$ принадлежит $\mathcal{RP}$, т.е. этот класс образован дополнениями языков из $\mathcal{RP}$.

Теорема 11.17. $\mathcal{ZPP} = \mathcal{RP} \bigcap$ со-$\mathcal{RP}$.

Доказательство. Сначала покажем, что $\mathcal{RP} \bigcap$ со-$\mathcal{RP} \subseteq \mathcal{ZPP}$. Пусть L принадлежит $\mathcal{RP} \bigcap$ со-$\mathcal{RP}$, т.е. как L, так и $\overline{L}$ имеют МТ типа Монте-Карло с полиномиально ограниченным временем. Предположим, что $p(n)$ — достаточно большой полином, ограничивающий время работы обеих машин. Построим для L машину Тьюринга M типа Лас-Вегас следующим образом.

1. Запустим машину типа Монте-Карло для L; если она допускает, M допускает и останавливается.
2. Если машина для L не допускает, запустим МТ типа Монте-Карло для $\overline{L}$. Если эта МТ допускает, M останавливается без допускания. В противном случае возвращаемся к п. 1.

Очевидно, M только допускает вход w, если w принадлежит L, и только отвергает w, если w не находится в L. Ожидаемое время работы в одном цикле (п. 1 и 2) — $2p(n)$. Вероятность того, что один цикл разрешит вопрос, не меньше 1/2. Если w принадлежит L, то п. 1 имеет 50% шансов привести M к допусканию, а если не принадлежит — п. 2 имеет 50% шансов привести M к отверганию. Таким образом, ожидаемое время работы M не больше

$$2p(n) + \frac{1}{2}2p(n) + \frac{1}{4}2p(n) + \frac{1}{8}2p(n) + \ldots = 4p(n).$$

Рассмотрим обратное утверждение. Предположим, что L принадлежит $\mathcal{ZPP}$, и покажем, что L находится как в $\mathcal{RP}$, так и в со-$\mathcal{RP}$. Нам известно, что L допускается МТ M_1 типа Лас-Вегас, ожидаемое время работы которой — некоторый полином $p(n)$. Построим для L МТ M_2 типа Монте-Карло следующим образом. M_2 имитирует $2p(n)$ шагов работы M_1. Если M_1 допускает в течение этого времени, то же делает и M_2; в противном случае она отвергает.

Предположим, что вход w длиной n не принадлежит L. Тогда M_1 наверняка не допускает w; то же сделает и M_2. Пусть вход w принадлежит L. M_1 наверняка в конце концов допускает w, но это может произойти как в пределах $2p(n)$ шагов, так и за их пределами.

Однако мы утверждаем, что вероятность того, что M_1 допускает w в пределах $2p(n)$ шагов, не меньше 1/2. Предположим, что эта вероятность является константой $c < 1/2$. Тогда ожидаемое время работы M_1 со входом w не меньше $(1 - c)2p(n)$, поскольку $1 - c$ является вероятностью того, что M_1 нужно больше, чем $2p(n)$ времени. Но если $c < 1/2$, то $2(1 - c) > 1$, и ожидаемое время работы M_1 со входом w больше $p(n)$. Получено проти-

воречие с предположением, что M_1 имеет ожидаемое время работы не больше $p(n)$. Это позволяет сделать вывод, что вероятность того, что M_2 допускает, не меньше 1/2. Итак, M_2 является МТ типа Монте-Карло с полиномиально ограниченным временем, что доказывает принадлежность L классу $\mathcal{RP}$.

Для доказательства, что L также находится в со-$\mathcal{RP}$, используется, по существу, такая же конструкция, но с отрицанием выхода M_2, т.е. для того, чтобы допустить $\overline{L}$, M_2 допускает, когда M_1 отвергает в пределах времени $2p(n)$; в противном случае M_2 отвергает. Теперь M_2 является МТ типа Монте-Карло с полиномиально ограниченным временем для $\overline{L}$. □

11.4.8. Соотношения с классами $\mathcal{P}$ и $\mathcal{NP}$

Из теоремы 11.17 следует, что $\mathcal{ZPP} \subseteq \mathcal{RP}$. Место этих классов между $\mathcal{P}$ и $\mathcal{NP}$ определяют следующие простые теоремы.

Теорема 11.18. $\mathcal{P} \subseteq \mathcal{ZPP}$.

Доказательство. Любая детерминированная полиномиально ограниченная МТ является также полиномиально ограниченной МТ типа Лас-Вегас, не использующей возможности случайных выборов. □

Теорема 11.19. $\mathcal{RP} \subseteq \mathcal{NP}$.

Доказательство. Пусть для языка L дана полиномиально ограниченная МТ M_1 типа Монте-Карло. Можно построить недетерминированную МТ M для L с тем же ограничением времени. Когда M_1 рассматривает случайный бит в первый раз, M_2 недетерминированно выбирает оба возможных значения этого бита и записывает их на свою собственную ленту, имитирующую случайную ленту M_1. M_2 допускает, когда допускает M_1, и не допускает в противном случае.

Пусть w принадлежит L. Тогда, поскольку M_1 имеет вероятность допускания w не менее 50%, должна существовать последовательность битов на ее случайной ленте, ведущая к допусканию w. M_2 выберет эту последовательность битов среди прочих, и также допустит. Таким образом, w принадлежит $L(M_2)$. Однако если w не принадлежит L, то ни одна последовательность случайных битов не приводит M_1 к допусканию, следовательно, нет и последовательности случайных выборов, приводящей к допусканию M_2. Таким образом, w не принадлежит $L(M_2)$. □

На рис. 11.8 представлены соотношения между введенными здесь и другими “близлежащими” классами.

11.5. Сложность проверки простоты

В данном разделе представлена проблема проверки, является ли целое число простым. Вначале обсуждается, почему простые числа и проверка простоты составляют не-

отъемлемую часть в системах компьютерной безопасности. Далее показывается, что проблема простоты принадлежит как $\mathcal{NP}$, так и со-$\mathcal{NP}$. Наконец, обсуждается рандомизированный алгоритм, показывающий, что эта проблема принадлежит также $\mathcal{RP}$.

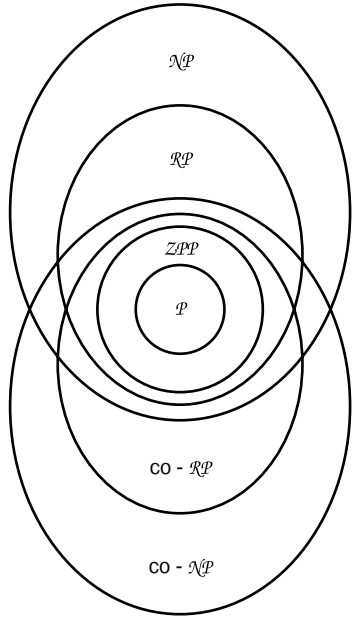


Рис. 11.8. Соотношение $\mathcal{ZPP}$, $\mathcal{RP}$ и других классов

11.5.1. Важность проверки пустоты

Целое число p называется *простым*, если его целыми положительными делителями являются только 1 и само p. Если целое число не является простым, оно называется *составным*. Каждое составное число можно записать в виде произведения простых единственным образом с точностью до порядка сомножителей.

Пример 11.20. Первые пять простых чисел — это 2, 3, 5, 7, 11 и 17. Число 504 составное, а его разложение на простые имеет вид $2^3 \times 3^2 \times 7$. □

Существует множество способов, повышающих степень компьютерной безопасности и основанных на предположении, что разложение чисел, т.е. поиск их простых делителей, является трудной задачей. В частности, схемы, основанные на так называемых RSA-шифрах (R. Rivest, A. Shamir, L. Adleman — изобретатели этих шифров), используют 128-битовые целые, представляющие собой произведения двух простых, занимающих

примерно по 64 бит.[8] Рассмотрим два сценария событий, в которых простые числа играют важную роль.

Шифрование с открытыми ключами

Вы хотите купить книгу у продавца, подключенного к сети. Продавец запрашивает номер вашей кредитной карточки, но печатать номер в форме и посылать форму по телефонным линиям или через Internet слишком рискованно, так как некий злоумышленник может подслушать линию или перехватить пакеты, идущие через Internet.

Чтобы предотвратить возможность прочитать номер вашей карточки, продавец посылает на ваш броузер ключ, возможно, 128-битовое произведение двух простых чисел, которые компьютер продавца сгенерировал специально для этого. Ваш броузер использует функцию $y = f_k(x)$, которая берет ключ k и данные x, предназначенные для шифрования. Функцию f, представляющую собой часть схемы RSA, могут знать даже потенциальные злоумышленники, но считается, что без знания разложения k обратную функцию f_k^{-1}, для которой $x = f_k^{-1}(y)$, невозможно вычислить за время, которое меньше экспоненциального по длине k.

Таким образом, даже если злоумышленник видит y и знает, как работает f, он не сможет восстановить x (в данном случае номер кредитной карточки), не зная, как k раскладывается на множители. С другой стороны, продавец, зная разложение k, сгенерированное им изначально, может легко применить f_k^{-1} и восстановить x по y.

Электронная подпись на основе публичных ключей

Исходный сценарий, для которого были построены шифры RSA, был следующим. Вам хотелось бы “подписывать” электронную почту так, чтобы другие могли легко определить, что эта почта пришла именно от вас, и чтобы никто не мог подделать ваше имя на ней. Например, вам хотелось бы подписать сообщение x = “Я обещаю заплатить Салли Ли 10 долларов”, но вы не хотите, чтобы без вашего ведома Салли или кто-то третий могли создать сообщение, якобы подписанное вами.

Для достижения этих целей вы выбираете ключ k, простые сомножители которого известны только вам. Затем публикуете k, например, на Web-странице, так что любой может применить к вашему сообщению функцию f_k. Если вы хотите подписать упомянутое сообщение x и послать его Салли, вы вычисляете $y = f_k^{-1}(x)$ и посылаете y Салли. Салли может взять k, ваш *публичный ключ*, с вашей Web-страницы и с его помощью вычислить $x = f_k(y)$. Таким образом, она знает, что вы действительно пообещали заплатить ей 10 долларов.

Если вы отказываетесь от факта отправки сообщения y, Салли может обосновать в суде, что только вам известна функция f_k^{-1}, и для нее или еще кого-то узнать эту функцию было бы “невозможно”. Таким образом, только вы могли создать y. Данная система

[8] В реально действующих системах используются 512–1024 и более битов для составных чисел и соответствующие количества для простых. См., например, A. J. Menezes, P. C. van Oorchot, S. A. Vanstone, *Handbook of applied cryptography*, CRC Press, 1997. — *Прим. перев*.

основана на вероятном, но не доказанном предположении, что произведение двух простых чисел очень трудно разложить на множители.

Требования к сложности проверки простоты

Оба описанных выше сценария считаются безопасными в том смысле, что разложение произведения двух простых чисел действительно требует экспоненциального времени. Теория сложности, представленная здесь и в главе 10, используется в изучении безопасности и криптографии следующими двумя путями.

1. Построение публичных ключей требует быстрого нахождения больших простых чисел. Одно из основных положений теории чисел состоит в том, что вероятность n-битового числа быть простым составляет порядка $1/n$. Таким образом, если бы у нас был полиномиальный по времени (относительно n, а не самого числа) способ проверки, является ли n-битовое число простым, мы могли бы выбирать числа случайно, проверять их и останавливаться, обнаружив простое число. Это давало бы нам полиномиальный алгоритм типа Лас-Вегас для обнаружения простых чисел, поскольку ожидаемое количество чисел, которые нужно проверить до появления n-битового простого, приблизительно равно n. Например, если нам нужны 64-битовые простые числа, достаточно проверить в среднем около 64 чисел, хотя в наихудшем случае понадобилось бы бесконечно много проверок. К сожалению, похоже, что гарантированно полиномиальной проверки простоты не может быть, хотя и существует полиномиальный алгоритм типа Монте-Карло, как будет показано в разделе 11.5.4.

2. Безопасность шифрования, основанного на RSA-схеме, зависит от невозможности разложить произвольное целое число за полиномиальное (относительно числа битов в ключе) время, и в частности, от невозможности разложить целое, о котором известно, что оно — произведение двух больших простых чисел. Мы были бы счастливы показать, что множество простых чисел образует NP-полный язык, или даже что множество составных NP-полно. Тогда полиномиальный алгоритм разложения доказывал бы, что $\mathcal{P} = \mathcal{NP}$, поскольку приводил бы к полиномиальным по времени проверкам для обоих указанных языков. Увы, в разделе 11.5.5 будет показано, что языки как простых, так и составных чисел принадлежат $\mathcal{NP}$. Они дополняют друг друга, поэтому, если бы какой-либо из них был NP-полным, то выполнялось бы равенство $\mathcal{NP}$ = со-$\mathcal{NP}$, а его правильность весьма сомнительна. Кроме того, принадлежность множества простых чисел классу $\mathcal{RP}$ означает, что, если бы можно было показать NP-полноту этого множества, то верным было бы равенство $\mathcal{RP} = \mathcal{NP}$, которое также маловероятно.

11.5.2. Введение в модулярную арифметику

Перед тем как рассматривать алгоритмы распознавания множества простых чисел, представим основные понятия, связанные с *модулярной арифметикой*, т.е. обычными

арифметическими операциями, которые выполняются по модулю некоторого целого числа, зачастую простого. Пусть p — произвольное (положительное) целое число. Тогда целыми по модулю p являются числа 0, 1, …, $p - 1$.

Сложение и умножение по модулю p (modulo p) определяются в применении к этому множеству из p чисел, когда выполняются обычные действия, после чего берется остаток от деления на p. Сложение совсем просто, поскольку сумма или меньше p, и дополнительные действия не нужны, или находится между p и $2p - 2$, и тогда вычитание p дает результат в пределах от 0 до $p - 2$. Модулярное сложение подчиняется обычным алгебраическим законам; оно коммутативно, ассоциативно и имеет 0 в качестве единицы. Вычитание остается обращением сложения, и модулярную разность $x - y$ можно вычислить путем обычного вычитания и прибавления p, если результат меньше 0. Обратным к x является $-x$, т.е. $0 - x$, как и в обычной арифметике. Таким образом, $-0 = 0$, а если $x \neq 0$, то $-x$ — это то же, что $p - x$.

Пример 11.21. Пусть $p = 13$. Тогда $3 + 5 = 8$, а $7 + 10 = 4$, поскольку в обычной арифметике $7 + 10 = 17$, что не меньше 13. Отнимая 13, получаем правильное значение 4. Значением –5 modulo 13 будет $13 - 5$, или 8. Разность $11 - 4$ modulo 13 равна 7, тогда как $4 - 11$ — 6 ($4 - 11 = -7$, поэтому нужно прибавить 13 и получить 6). □

Умножение по модулю p выполняется путем умножения обычных чисел и вычисления остатка от деления на p. Умножение также удовлетворяет обычным алгебраическим законам; оно коммутативно и ассоциативно, единицей является 1, а нулем (аннигилятором) — 0. Оно также дистрибутивно относительно сложения. Однако деление на ненулевые элементы сложнее, и даже существование обратных к целым по модулю p зависит от того, является ли p простым. В любом случае, если x есть одно из целых по модулю p, т.е. $0 \leq x < p$, то x^{-1}, или $1/x$ — это такое число y (если существует), для которого $xy = 1$ modulo p.

Пример 11.22. На рис. 11.9 показана таблица умножения для ненулевых целых чисел по модулю простого числа 7. Элемент в строке i и столбце j равен произведению ij modulo 7. Отметим, что каждый ненулевой элемент имеет обратный; взаимно обратны 2 и 4, 3 и 5, а 1 и 6 обратны сами себе, т.е. 2×4, 3×5, 1×1 и 6×6 равны 1. Таким образом, можно делить на любой ненулевой элемент y, вычислив y^{-1} и умножив на y^{-1}. Например, $3/4 = 3 \times 4 = 3 \times 2 = 6$.

1	2	3	4	5	6
2	4	6	1	3	5
3	6	2	5	1	4
4	1	5	2	6	3
5	3	1	6	4	2
6	5	4	3	2	1

Рис. 11.9. Умножение по модулю 7

Сравним эту ситуацию с таблицей умножения по модулю 6. Во-первых, заметим, что обратные есть *только* у 3 и 5 (они обратны самим себе). Другие числа обратных не имеют. Во-вторых, в таблице есть ненулевые элементы, произведение которых равно 0, например, 2 и 3. Такое невозможно в обычной арифметике или в арифметике по модулю простого числа. □

1	2	3	4	5
2	4	0	2	4
3	0	3	0	3
4	2	0	4	2
5	4	3	2	1

Рис. 11.10. Умножение по модулю 6

Есть еще одно различие между умножением по модулю простого числа и по модулю составного, которое оказывается очень важным для проверки простоты. *Порядок* числа a по модулю p — это наименьшая положительная степень a, равная 1. Приведем без доказательства некоторые полезные сведения, связанные с порядком.

- Если p — простое, то $a^{p-1} = 1 \text{ modulo } p$. Это утверждение называется *теоремой Ферма*.[9]
- Порядок a по модулю простого p всегда является делителем $p - 1$.
- Если p — простое, то существует a, имеющее порядок $p - 1$ modulo p.

Пример 11.23. Вернемся к таблице умножения по модулю 7 (см. рис. 11.9). Число 2 имеет порядок 3, поскольку $2^2 = 4$ и $2^3 = 1$. Порядком 3 является 6, так как $3^2 = 2$, $3^3 = 6$, $3^4 = 4$, $3^5 = 5$ и $3^6 = 1$. Аналогично находим, что 4 имеет порядок 3, 6 — 2, а 1 — 1. □

11.5.3. Сложность вычислений в модулярной арифметике

Перед тем как рассматривать применение модулярной арифметики к проверке простоты, нужно установить основные факты, связанные со временем выполнения существенных операций. Предположим, нам нужно вычислять по модулю некоторого простого p, и двоичное представление p занимает n битов, т.е. само p близко к 2^n. Как всегда, время выполнения выражается в терминах n, длины входа, а не его "значения" p. Например, счет до p занимает время $O(2^n)$, так что любое вычисление, включающее p шагов, будет *не полиномиальным* по времени (как функции от n).

[9] Его не следует путать с "последней (большой, великой) теоремой Ферма", утверждающей несуществование целых решений уравнения $x^n + y^n = z^n$ при $n \geq 3$.

На обычном компьютере или на многоленточной МТ можно сложить два числа по модулю p за время $O(n)$. Напомним, что нужно просто сложить два двоичных числа и, если результат не меньше p, вычесть p. Аналогично, на компьютере или на машине Тьюринга можно умножить два числа за время $O(n^2)$. После обычного умножения и получения результата длиной не более $2n$ бит, нужно поделить на p и взять остаток.

Возведение числа x в степень сложнее, поскольку степень сама может быть экспоненциальной относительно n. Как мы увидим, большое значение будет иметь возведение x в степень $p - 1$. Поскольку p близко к 2^n, умножение x самого на себя $p - 2$ раз потребовало бы $O(2^n)$ умножений, и даже если бы каждое умножение касалось лишь n-битовых чисел и выполнялось за время $O(n^2)$, общее время было бы $O(n^2 2^n)$, что не является полиномиальным относительно n.

К счастью, существует следующий прием “рекурсивного удвоения”, который позволяет вычислить x^{p-1} (или любую другую степень до p) за время, полиномиальное относительно n.

1. Вычислим не более n степеней x, x^2, x^4, x^8, …, пока степень не превысит $p - 1$. Каждое значение является n-битовым числом, которое находится за время $O(n^2)$ путем возведения в квадрат предыдущего элемента последовательности, поэтому общее время равно $O(n^3)$.
2. Найдем двоичное представление $p - 1$, скажем, $p - 1 = a_{n-1}\ldots a_1 a_0$. Можно записать

   $$p - 1 = a_0 + 2a_1 + 4a_2 + \ldots + 2^{n-1}a_{n-1},$$

   где каждое a_j есть либо 0, либо 1. Следовательно,

   $$x^{p-1} = x^{a_0 + 2a_1 + 4a_2 + \cdots + 2^{n-1}a_{n-1}},$$

   что представляет собой произведение тех степеней x^{2^j}, для которых $a_j = 1$. Поскольку все эти степени вычислены в п. 1 и являются n-битовыми, их произведение (или произведение части из них) можно вычислить за время $O(n^3)$.

Таким образом, все вычисление x^{p-1} занимает время $O(n^3)$.

11.5.4. Рандомизированная полиномиальная проверка простоты

Теперь обсудим, как применить рандомизированные вычисления для поиска больших простых чисел. Точнее, покажем, что язык составных чисел принадлежит $\mathcal{RP}$. Метод, который в действительности используется для генерации n-битовых простых чисел, состоит в том, что случайно выбирается n-битовое число и много раз, скажем, 50, применяется алгоритм типа Монте-Карло для распознавания составных чисел. Если некоторая проверка показывает, что число составное, то оно точно не простое. Если все 50 проверок не могут сказать, что оно составное, то вероятность того, что оно действительно составное, не больше 2^{-50}. Таким образом, можно довольно уверенно сказать, что число простое, и этим обосновать безопасность наших операций.

Полный алгоритм здесь не приводится, но обсуждается идея, имеющая очень мало исключений. Напомним, что теорема Ферма гласит: если p — простое, то x^{p-1} modulo p для любого x равно 1. Верно и то, что если p — составное, и существует x, для которого x^{p-1} modulo p не равно 1, то не менее половины чисел y от 1 до $p-1$ имеют $y^{p-1} \neq 1$.

Таким образом, используем следующий алгоритм типа Монте-Карло для составных чисел.

1. Выберем случайно x из диапазона от 1 до $p-1$.
2. Вычислим x^{p-1} modulo p. Заметим, что если p есть n-битовое число, то это вычисление занимает время $O(n^3)$ (см. конец раздела 11.5.3).
3. Если $x^{p-1} \neq 1$ modulo p, то допустим вход, т.е. x — составное. В противном случае остановимся без допускания.

Если p является простым, то $x^{p-1} = 1$, так что мы всегда останавливаемся без допускания; это одна часть условий типа Монте-Карло — если вход не принадлежит языку, то никогда не допускается. Почти для всех составных чисел не менее половины x будут иметь $x^{p-1} \neq 1$, поэтому у нас есть не менее 50% шансов допускания при любом запуске этого алгоритма, т.е. верно другое условие, налагаемое на алгоритмы типа Монте-Карло.

Представленные рассуждения были бы демонстрацией того, что проблема составных чисел принадлежит $\mathcal{RP}$, если бы не существование небольшого количества составных чисел c, дающих $x^{c-1} = 1$ modulo c для большинства x от 1 до $c-1$, в частности, для x, взаимно простых с c. Эти числа, называемые *числами Кармайкла* (*Carmichael*), требуют проведения еще одной, более сложной, проверки (она здесь не описывается), чтобы убедиться, что они составные. Наименьшим числом Кармайкла является 561, т.е. можно показать, что $x^{560} = 1$ modulo 561 для всех x, которые не делятся на 3, 11 или 17, хотя $561 = 3 \times 11 \times 17$, очевидно, составное. Итак, утверждается, но без полного доказательства, следующее.

Теорема 11.24. Множество составных чисел принадлежит $\mathcal{RP}$. □

Можно ли разложить на множители за случайное полиномиальное время?

Отметим, что алгоритм из раздела 11.5.4 может сказать, что число является составным, но не говорит, как составное число разложить на множители. Есть основания полагать, что не существует способа разложения, даже с использованием рандомизации, которому было бы достаточно полиномиального или хотя бы ожидаемого полиномиального времени. Если бы это предположение было неправильным, то приложения, описанные в разделе 11.5.1, были бы небезопасными и их нельзя было использовать.

11.5.5. Недетерминированные проверки простоты

Здесь обсуждается еще один важный и интересный результат, связанный с проверкой простоты, — язык простых чисел находится в $\mathcal{NP} \cap$ со-$\mathcal{NP}$. Следовательно, язык составных чисел, представляющий собой дополнение языка простых, также принадлежит $\mathcal{NP} \cap$ со-$\mathcal{NP}$. Отсюда следует, что вероятность NP-полноты языков простых и составных чисел ничтожна, поскольку, если бы это было так, истинным стало бы совершенно невероятное равенство $\mathcal{NP}$ = со-$\mathcal{NP}$.

Одна часть указанного утверждения проста — язык составных чисел принадлежит $\mathcal{NP}$, поэтому язык простых чисел находится в со-$\mathcal{NP}$. Докажем это утверждение.

Теорема 11.25. Множество составных чисел принадлежит $\mathcal{NP}$.

Доказательство. Недетерминированный полиномиальный алгоритм распознавания составных чисел имеет следующий вид.

1. Имея данное n-битовое число p, угадаем сомножитель f, состоящий не более, чем из n битов ($f = 1$ или $f = p$, естественно, не рассматриваются).
2. Разделим p на f и проверим, что остаток равен 0. Допускаем, если так. Данная часть детерминирована и может быть выполнена за время $O(n^2)$ на многоленточной МТ.

Если p — составное, то оно должно иметь хотя бы один сомножитель f, не равный 1 и p. НМТ, угадывающая все возможные числа, содержащие не более n битов, по одной из веток угадает f. Эта ветка ведет к допусканию. Наоборот, допускание НМТ означает, что найден делитель числа p, не равный 1 и p. Таким образом, язык описанной НМТ содержит все составные числа, и ничего более. □

Распознавание простых чисел с помощью НМТ сложнее. Можно было угадать причину (делитель) того, что число не является простым, но как проверить корректность предположения, что число *действительно* является простым? Недетерминированный полиномиальный алгоритм основан на том факте (утверждаемом, но не доказанном), что, если p — простое, то существует число x между 1 и $p - 1$, имеющее порядок $p - 1$. В частности, в примере 11.23 отмечалось, что при простом $p = 7$ числа 3 и 5 имеют порядок 6.

Можно легко угадать число x, используя недетерминированность МТ, но совершенно неясно, как проверить, что x имеет порядок $p - 1$. Причина в том, что, если определение порядка применяется непосредственно, то нужно проверить, что ни одно из чисел $x^2, x^3, \ldots, x^{p-2}$ не равно 1. Но такая проверка требует времени не менее 2^n, если p — n-битовое число.

Лучшая стратегия — использовать еще один факт, который утверждался, но не был доказан, а именно: порядок x по модулю простого p является делителем $p - 1$. Таким образом, если бы нам были известны простые делители $p - 1$[10], было бы достаточно прове-

[10] Заметим, что, если p — простое, то $p - 1$ не может быть простым, кроме как в тривиальном случае $p = 3$. Причина в том, что все простые, кроме 2, нечетны.

рить, что $x^{(p-1)/q} \neq 1$ для каждого простого сомножителя q числа $p - 1$. Если бы ни одна из этих степеней x не равнялась 1, то порядком x было бы $p - 1$. Число таких проверок есть $O(n)$, поэтому все их можно выполнить с помощью полиномиального алгоритма. Конечно, разложить $p - 1$ на простые сомножители нелегко, но можно *угадать* их, и проверить, что

а) их произведение действительно равно $p - 1$;

б) каждый из них является простым, используя данный недетерминированный полиномиальный алгоритм рекурсивно.

Детали алгоритма и доказательство того, что он является недетерминированным полиномиальным, приводятся в следующей теореме.

Теорема 11.26. Множество простых чисел принадлежит $\mathcal{NP}$.

Доказательство. Пусть p — n-битовое число. Если n не больше 2 (т.е. p — это 1, 2 или 3), то отвечаем сразу: 2 и 3 — простые, 1 — нет. В противном случае выполним следующее.

1. Угадаем список сомножителей (q_1, q_2, …, q_k), двоичные представления которых вместе занимают не более $2n$ битов, и ни одно из них не имеет более $n - 1$ битов. Одно и то же простое число может появляться несколько раз, поскольку $p - 1$ может иметь сомножитель, возводимый в степень больше 1. Например, если $p = 13$, то простые сомножители $p - 1 = 12$ образуют список (2, 2, 3). Данная часть алгоритма недетерминирована, но каждая ветвь требует времени $O(n)$.
2. Перемножим все сомножители q_i и проверим, равно ли их произведение $p - 1$. Эта часть требует времени не более $O(n^2)$ и является детерминированной.
3. Если произведение сомножителей равно $p - 1$, то рекурсивно проверим, является ли каждый из них простым, используя данный алгоритм.
4. Если не все q просты, угадаем значение x и проверим неравенство $x^{(p-1)/q} \neq 1$ для каждого из q. Эта проверка обеспечивает, что x имеет порядок $p - 1$ modulo p, поскольку, если бы это было не так, то его порядок должен был бы делиться хотя бы на одно из $(p - 1)/q$, но мы только что установили, что он не делится. В подтверждение заметим, что любое x при возведении в степень его порядка должно равняться 1. Возведения в степень можно выполнить эффективным методом, описанным в разделе 11.5.3. Таким образом, нужно выполнить не более k возведений в степень, что не больше n, и каждое можно выполнить за время $O(n^3)$, что дает общее время $O(n^4)$ для каждого шага.

Наконец, нужно проверить, что приведенный недетерминированный алгоритм полиномиален. Каждый шаг, за исключением рекурсивного шага 3, требует времени $O(n^4)$ вдоль любой недетерминированной ветви. Поскольку рекурсия нелинейна, рекурсивные вызовы можно изобразить в виде дерева (рис. 11.11). В корне находится n-битовое p,

проверяемое на простоту. Сыновьями корня являются q_j — угадываемые сомножители $p - 1$, которые также нужно проверить на простоту. Под каждым q_j находятся угадываемые сомножители $q_j - 1$, которые также нужно проверить, и так далее, пока не дойдем до чисел, состоящих не более, чем из 2 бит — листьев дерева.

Произведение сыновей любого узла меньше значения в самом этом узле, поэтому произведение значений в узлах любого уровня не более p. Время работы, выполняемой для узла со значением i, исключая работу в рекурсивных вызовах, оценивается как $a(\log_2 i)^4$ для некоторой константы a, поскольку это время прямо пропорционально четвертой степени количества битов, необходимых для двоичного представления значения i.

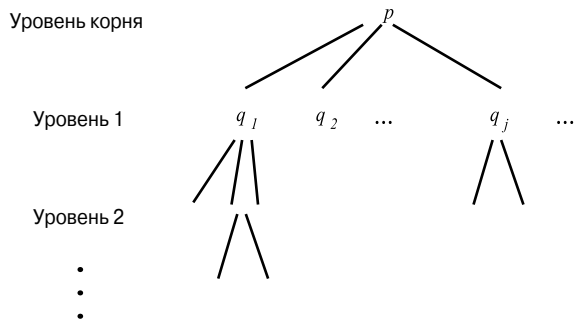


Рис. 11.11. Рекурсивные вызовы, совершаемые алгоритмом из теоремы 11.26, образуют дерево высотой и шириной не более n

Таким образом, чтобы получить верхнюю оценку времени работы на любом уровне, ограничим сверху сумму $\sum_j a(\log_2(i_j))^4$, учитывая, что произведение $i_1 i_2 \ldots$ не превосходит p. Если $i_1 = p$, и других i_j нет, то сумма равна $a(\log_2 p)^4$, т.е. не превосходит an^4, поскольку n — число битов двоичного представления p, а $\log_2 p$ не более n.

Итак, время работы на каждом уровне глубины не превышает $O(n^4)$. Так как у дерева не более n уровней, для любой ветви недетерминированной проверки, является ли p простым, достаточно $O(n^5)$. □

Теперь ясно, что языки и простых, и составных чисел находятся в $\mathcal{NP}$. Если бы один из них был NP-полным, то по теореме 11.2 это доказывало бы, что $\mathcal{NP}$ = со-$\mathcal{NP}$.

11.5.6. Упражнения к разделу 11.5

11.5.1. Вычислите следующие значения по модулю 13:

а) $11 + 9$;

б) (*) $9 - 11$;

в) 5×8;

г) (*) 5 / 8;

д) 5^8.

11.5.2. В разделе 11.5.4 утверждалось, что $x^{560} = 1$ modulo 561 для большинства значений x между 1 и 560. Выберите какие-нибудь значения x и проверьте это равенство. Конечно, сначала выразите 560 в двоичном виде, а затем вычислите x^{2^j} для соответствующих значений j, избегая 559 умножений (см. раздел 11.5.3).

11.5.3. Целое число x между 1 и $p - 1$ называется квадратичным вычетом по модулю p, если существует целое y между 1 и $p - 1$, для которого $y^2 = x$.

а) (*) Укажите квадратичные вычеты по модулю 7. Можно воспользоваться таблицей на рис. 11.9.

б) Укажите квадратичные вычеты по модулю 13.

в) (!) Докажите, что если p является простым, то число квадратичных вычетов по модулю p равно $(p - 1)/2$, т.е. половина положительных целых чисел по модулю p — квадратичные вычеты. *Указание*. Проверьте ваши данные, полученные в частях *а* и *б*. Видите ли вы образец, позволяющий понять, почему каждый квадратичный вычет является квадратом одновременно двух разных чисел? Может ли одно целое число быть квадратом трех разных чисел, если p простое?

Резюме

- *Класс со-$\mathcal{NP}$.* Язык принадлежит со-$\mathcal{NP}$, если его дополнение находится в $\mathcal{NP}$. Все языки из $\mathcal{P}$, очевидно, принадлежат со-$\mathcal{NP}$, но похоже, что существуют языки, принадлежащие $\mathcal{NP}$, но не со-$\mathcal{NP}$, и наоборот. В частности, NP-полные проблемы, скорее всего, не принадлежат со-$\mathcal{NP}$.
- *Класс $\mathcal{PS}$.* Язык принадлежит $\mathcal{PS}$ (полиномиальное пространство), если он допускается детерминированной МТ, для которой существует такой полином $p(n)$, при котором на входе длиной n МТ никогда не использует более $p(n)$ клеток ленты.
- *Класс $\mathcal{NPS}$.* Можно определить допускание недетерминированной МТ, использование ленты которой ограничено полиномиальной функцией от длины входа. Класс таких языков обозначается $\mathcal{NPS}$. Однако теорема Сэвича гласит, что $\mathcal{PS} = \mathcal{NPS}$. В частности, НМТ с ограничением пространства $p(n)$ может быть проимитирована с помощью ДМТ, использующей пространство $p^2(n)$.
- *Рандомизированные алгоритмы и машины Тьюринга.* Многие алгоритмы продуктивно используют рандомизацию. В настоящем компьютере генератор случайных чисел используется для имитации “бросания монеты”. Рандомизированная машина

Тьюринга может достичь такого же случайного поведения, если ее снабдить дополнительной лентой, на которую записывается последовательность случайных битов.

- *Класс $\mathcal{RP}$.* Язык допускается за случайное полиномиальное время, если существует полиномиальная рандомизированная машина Тьюринга, имеющая не менее 50% шансов на допускание своего входа, при условии, что этот вход принадлежит языку. Если вход не принадлежит языку, то эта МТ не допускает. Такая МТ или алгоритм называются "типа Монте-Карло".
- *Класс $\mathcal{ZPP}$.* Язык принадлежит классу безошибочного вероятностного полиномиального времени, если допускается рандомизированной МТ, которая всегда дает правильный ответ о принадлежности входа языку. Ожидаемое время работы этой МТ полиномиально, хотя в наихудшем случае может быть больше любого полинома. Такая МТ или алгоритм называются "типа Лас-Вегас".
- *Соотношения между классами языков.* Класс со-$\mathcal{RP}$ — это множество дополнений языков из $\mathcal{RP}$. Известны следующие включения: $\mathcal{P} \subseteq \mathcal{ZPP} \subseteq (\mathcal{RP} \bigcap \text{со-}\mathcal{RP})$. Кроме того, $\mathcal{RP} \subseteq \mathcal{NP}$ и, следовательно, со-$\mathcal{RP} \subseteq$ со-$\mathcal{NP}$.
- *Простые числа и $\mathcal{NP}$.* Как язык простых чисел, так и его дополнение — язык составных чисел — принадлежат $\mathcal{NP}$. Эти два факта делают невероятной NP-полноту данных двух языков. Поскольку на простых числах основаны важные системы шифрования, доказательство последнего утверждения служило бы строгим обоснованием их безопасности.
- *Простые числа и $\mathcal{RP}$.* Язык составных чисел принадлежит $\mathcal{RP}$. Рандомизированный полиномиальный алгоритм проверки, является ли число составным, обязан в общем случае допускать генерацию больших простых чисел или, по крайней мере, чисел, имеющих сколь угодно малую вероятность быть составными.

Литература

Изучение классов языков, определяемых с помощью ограничения пространства, используемого машинами Тьюринга, началось в [2]. Первые PS-полные проблемы описаны Карпом в статье [4], исследовавшей значимость NP-полноты. Оттуда же PS-полнота проблемы, представленной в упражнении 11.3.2, — является ли регулярное выражение эквивалентным Σ^*.

PS-полнота булевых формул с кванторами доказана в неопубликованной работе Л. Стокмейера, а переключательной игры Шеннона (см. упражнение 11.3.3) — в статье [1].

Принадлежность языка простых чисел классу $\mathcal{NP}$ установлена Праттом в [9], а составных классу $\mathcal{RP}$ — Рабином в [10]. Интересно, что приблизительно тогда же в [6] бы-

ло опубликовано доказательство, что множество простых чисел на самом деле принадлежит $\mathcal{P}$ при условии, что верна так называемая расширенная гипотеза Римана (тогда в ее истинность многие верили).

Вопросы данной главы подробнее освещены в нескольких книгах. Рандомизированные алгоритмы, а также полные алгоритмы проверки простоты представлены в [7]. Источником алгоритмов модулярной арифметики послужила [5]. В [3] и [8] изучены многие классы сложности, не упоминаемые здесь.

1. S. Even and R. E. Tarjan, "A combinatorial problem which is complete for polynomial space", *J. ACM* **23**:4 (1976), pp. 710–719.
2. J. Hartmanis, P. M. Lewis II, and R. E. Stearns, "Hierarchies of memory limited computations", *Proc. Sixth Annual IEEE Symposium on Switching Circuit Theory and Logical Design* (1965), pp. 179–190.
3. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading MA, 1979.
4. R. M. Karp, "Reducibility among combinatorial problems", in *Complexity of Computer Computations* (R. E. Miller, ed.), Plenum Press, New York, (1972), pp. 85–104. (Карп Р. М. Сводимость комбинаторных проблем. — Кибернетический сборник, новая серия, вып. 12. — М.: Мир, 1975. — С. 16–38.)
5. D. E. Knuth, *The Art of Computer Programming, Vol. II: Seminumerical Algorithms*, Addison-Wesley, Reading MA, 1997 (third edition). (Кнут Д. Искусство программирования. В 3 т. Т. 2: Получисленные алгоритмы. — М.: Издательский дом "Вильямс", 2000.)
6. G. L. Miller, "Riemann's hypothesis and tests for primality", *J. Computer and System Sciences* **13** (1976), pp. 300–317.
7. R. Motwani and P. Raghavan, *Randomized Agorithms*, Cambridge Univ. Press, 1995.
8. C. H. Papadimitriou, *Computational Complexity*, Addison-Wesley, Reading MA, 1994.
9. V. R. Pratt, "Every prime has a succinct certificate", *SIAM J. Computing*, **4**:3 (1975), pp. 214–220.
10. M. O. Rabin, "Probabilistic algorithms", in *Algorithms and Complexity: Recent Results and New Directions* (J. F. Traub, ed.) pp. 21–39, Academic Press, New York, 1976.
11. R. L. Rivest, A. Shamir, and L. Adliman, "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM* **21** (1978), pp. 120–126.
12. W. J. Savitch, "Relationships between deterministic and nondeterministic tape complexities", *J.Computer and System Sciences* **4**:2 (1970), pp. 177–192.

Предметно-именной указатель

Р

С

Т

Научно-популярное издание

Джон Э. Хопкрофт, Раджив Мотвани, Джеффри Д. Ульман

Введение в теорию автоматов, языков и вычислений, 2-е изд.

Литературный редактор	*О.Ю. Белозовская*
Верстка	*М.А. Удалов*
Художественный редактор	*Т.А. Тараброва*
Обложка	*С.А. Чернокозинский*
Корректоры	*Л.А. Гордиенко, О.В. Мишутина*

Издательский дом “Вильямс”
127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 19.11.2007. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. лист. 42,57. Уч.-изд. лист. 29,99.
Тираж 1000. Заказ № 0000.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15