PROLOG PROGRAMMING FOR ARTIFICIAL INTELLIGENCE

Ivan Bratko

E. Kardelj University • J. Stefan Institute
Yugoslavia

ADDISON-WESLEY
PUBLISHIHG
COMPANY

Wokingham, England • Reading, Massachusetts • Menlo Park, California
Don Mills, Ontario • Amsterdam • Bonn • Sydney • Singapore
Tokyo • Madrid • Bogota • Santiago • San Juan

И. БРАТКО

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ПРОЛОГ ДЛЯ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Перевод с английского
А.И. Лупенко и А.М.Степанова

под редакцией А.М. Степанова

Москва «МИР» 1990

ББК 22.19
Б 87
УДК 519.68

Братко И.

Б 87 Программирование на языке Пролог для искусственного интеллекта: Пер. с англ. –М.: Мир, 1990.– 560 с., ил.

ISBN 5-03-001425-X

Книга известного специалиста по программированию (Югославия), содержащая основы языка Пролог и его приложения для решения задач искусственного интеллекта. Изложение отличается методическими достоинствами – книга написана в хорошем стиле, живым языком. Книга дополняет имеющуюся на русском языке литературу по языку Пролог.

Для программистов разной квалификации, специалистов по искусственному интеллекту, для всех изучающих программирование.

Б $\frac{2404010000 - 026}{041(01) - 90}$ 140 - 90 **ББК 22.19**

Редакция литературы по математическим наукам

ISBN 5-03-001425-X (русск.)
ISBN 0-201-14224-4 (англ.)

ОТ РЕДАКТОРА ПЕРЕВОДА

По существующей традиции предисловие редактора перевода – это своего рода рецензия, в которой обычно излагается история вопроса, а затем дается обзор содержания книги и оценка ее качества (как правило, рекламного характера). В данном случае моя задача несколько упрощается, так как все это читатель, перевернув страницу, найдет в предисловии известного американского ученого, специалиста по искусственному интеллекту П. Уинстона, а затем – в предисловии автора. Мне остается только присоединиться к авторитетному мнению П. Уинстона, что перед нами прекрасно написанный учебник по Прологу, ориентированный на практическое использование в области искусственного интеллекта. Добавлю также, что для советского читателя потребность в такой книге особенно велика, поскольку в нашей стране Пролог пока еще не получил того распространения, которого он заслуживает.

Несколько замечаний относительно особенностей перевода. Кроме обычных терминологических трудностей, как правило возникающих при переводе книг по программированию, переводчикам пришлось преодолевать одну дополнительную сложность. Дело в том, что в Прологе идентификаторы (имена переменных, процедур и атомов) несут на себе значительно большую смысловую нагрузку, чем в традиционных языках программирования. Поэтому программные примеры пришлось излагать на некоей условной русской версии Пролога – в противном случае, для читателей, не владеющих английским языком, эти примеры стали бы значительно менее понятными. Мы оставили без перевода все имена встроенных операторов и процедур, все же остальные имена переводились на русский язык. Следует признать, что в ряде случаев русская

версия этих имен оказалась менее эстетически привлекательной, чем исходный английский вариант. Пытаясь наиболее точно передать смысл того или иного имени, переводчик нередко оказывался перед нелегким выбором между громоздким идентификатором (иногда из нескольких слов) и неблагозвучной аббревиатурой. Впрочем, все эти проблемы хорошо известны любому «русскоязычному» программисту.

Главы 1–8 перевел А.И. Лупейко, а предисловия и главы 9–16 – А.М. Степанов. Подготовку оригинала-макета книги на ЭВМ выполнили А.Н. Черных и Н.Г. Черных.

Эту книгу можно рекомендовать как тем читателям, которые впервые приступают к изучению Пролога и искусственного интеллекта, так и программистам, уже имеющим опыт составления пролог-программ.

А.М. Степанов

ПРЕДИСЛОВИЕ

В средние века знание латинского и греческого языков являлось существенной частью образования любого ученого. Ученый, владеющий только одним языком, неизбежно чувствовал себя неполноценным, поскольку он был лишен той полноты восприятия, которая возникает благодаря возможности посмотреть на мир сразу с двух точек зрения. Таким же неполноценным ощущает себя сегодняшний исследователь в области искусственного интеллекта, если он не обладает основательным знакомством как с Лиспом, так и с Прологом – с этими двумя основополагающими языками искусственного интеллекта, без знания которых невозможен более широкий взгляд на предмет исследования.

Сам я приверженец Лиспа, так как воспитывался в Массачусетском технологическом институте, где этот язык был изобретен. Тем не менее, я никогда не забуду того волнения, которое я испытал, увидев в действии свою первую программу, написанную в пологовском стиле. Эта программа была частью знаменитой системы Shrdlu Терри Винограда. Решатель задач, встроенный в систему, работал в «мире кубиков» и заставлял руку робота (точнее, ее модель) перемещать кубики на экране дисплея, решая при этом хитроумные задачи, поставленные оператором.

Решатель задач Винограда был написан на Микроплэнере, языке, который, как мы теперь понимаем, был своего рода Прологом в миниатюре. Любой прологоподобный язык заставляет программиста мыслить в терминах целей, поэтому, несмотря на все недостатки Микроплэнера, достоинством этой программы было то, что в ее структуре содержались многочисленные явные указания на те или иные цели. Процедуры-цели «схватить», «освободить», «избавиться», «переместить», «отпустить» и т.п. делали программу простой и компактной, а поведение ее казалось поразительно разумным.

Решатель задач Виноградa навсегда изменил мое программистское мышление. Я даже переписал его на Лиспе и привел в своем учебнике по Лиспу в качестве примера – настолько эта программа всегда поражала меня мощью заложенной в ней философии «целевого» программирования, да и само программирование в терминах целей всегда доставляло мне удовольствие.

Однако учиться целевому программированию на примерах лисповских программ – это все равно, что читать Шекспира на языке, отличном от английского. Какое-то впечатление вы получите, но сила эстетического воздействия будет меньшей, чем при чтении оригинала. Аналогично этому, лучший способ научиться целевому программированию – это читать и писать программы на Прологе, поскольку сама сущность Пролога как раз и состоит в программировании в терминах целей.

В самом широком смысле слова эволюция языков программирования – это движение от языков низкого уровня, пользуясь которыми, программист описывает, *как* что-либо следует делать, к языкам высокого уровня, на которых просто указывается, *что* необходимо сделать. Так, например, появление Фортрана освободило программистов от необходимости разговаривать с машиной на прокрустовом языке адресов и регистров. Теперь они уже могли говорить на своем (или почти на своем) языке, только изредка делая уступки примитивному миру 80-колонных перфокарт.

Однако Фортран и почти все другие языки программирования все еще остаются языками типа «как». И чемпионом среди этих языков является, пожалуй, современный модернизированный Лисп. Так, скажем, Common Lisp, имея богатейшие выразительные возможности, разрешает программисту описывать наиболее «выразительно» именно то, *как* что-либо следует делать. В то же время очевидно, что Пролог порывает с традициями языков типа «как», поскольку он определенным образом направляет программистское мышление, заставляя программиста давать определения ситуаций и формулировать задачи вместо того, чтобы во всех деталях описывать способ решения этих задач.

Отсюда следует, насколько важен вводный курс по Прологу для всех студентов, изучающих вычислительную технику и программирование – просто не сущест-

вует лучшего способа понять, что из себя представляет программирование типа «что».

Многие страницы этой книги могут служить хорошей иллюстрацией того различия, которое существует между этими двумя стилями программистского мышления. Например, в первой главе это различие иллюстрируется на задачах, относящихся к семейным отношениям. Прологовский программист дает простое и естественное описание понятия «дедушка»: дедушка – это отец родителя. На Прологе это выглядит так:

дедушка(X, Z) :- отец(X, Y), родитель(Y, Z).

Как только пролог-система узнала, что такое дедушка, ей можно задать вопрос, например: кто является дедушкой Патрика? В обозначениях Пролога этот вопрос и типичный ответ имеют вид:

?- дедушка(X, патрик).

X = джеймс;
X = карл.

Каким образом решать эту задачу, как «прочесывать» базу данных, в которой записаны все известные отношения «отец» и «родитель», – это уже забота самой пролог-системы. Программист только сообщает системе то, что ему известно, и задает вопросы. Его в большей степени интересуют знания и в меньшей – алгоритмы, при помощи которых из этих знаний извлекается нужная информация.

Поняв, что очень важно научиться Прологу, естественно задать себе следующий вопрос – как это сделать. Я убежден, что изучение языка программирования во многом сходно с изучением естественного языка. Так, например, в первом случае может пригодиться инструкция по программированию точно так же, как во втором – словарь. Но никто не изучает язык при помощи словаря, так как слова – это только часть знаний, необходимых для овладения языком. Изучающий язык должен кроме того узнать те соглашения, следуя которым, можно получать осмысленные сочетания слов, а затем научиться у мастеров слова искусству литературного стиля.

Точно так же, никто не изучает язык программирования, пользуясь только инструкцией по программированию, так как в инструкциях очень мало или

вообще ничего не говорится о том, как хорошие программисты используют элементарные конструкции языка. Поэтому необходим учебник, причем лучшие учебники обычно предлагают читателю богатый набор примеров. Ведь в хороших примерах сконцентрирован опыт лучших программистов, а именно на опыте мы, в основном, и учимся.

В этой книге первый пример появляется уже на первой странице, а далее на читателя как из рога изобилия обрушивается поток примеров прологовских программ, написанных программистом-энтузиастом, горячим приверженцем прологовской идеологии программирования. После тщательного изучения этих примеров читатель не только узнает, как «работает» Пролог, но и станет обладателем личной коллекции программ-прецедентов, готовых к употреблению: он может разбирать эти программы на части, приспосабливать каждую часть к своей задаче, а затем снова собирать их вместе, получая при этом новые программы. Такое усвоение предшествующего опыта можно считать первым шагом на пути от новичка к программисту-мастеру.

Изучение хороших программных примеров дает, как правило, один полезный побочный эффект: мы узнаем из них не только очень многое о самом программировании, но и кое-что – о какой-нибудь интересной научной области. В данной книге такой научной областью, стоящей за большинством примеров, является искусственный интеллект. Читатель узнает о таких идеях в области автоматического решения задач, как сведение задач к подзадачам, прямое и обратное построение цепочки рассуждений, ответы на вопросы «как» и «почему», а также разнообразные методы поиска.

Одним из замечательных свойств Пролога является то, что это достаточно простой язык, и студенты могли бы использовать его непосредственно в процессе изучения вводного курса по искусственному интеллекту. Я не сомневаюсь, что многие преподаватели включат эту книгу в свои курсы искусственного интеллекта с тем, чтобы студенты смогли увидеть, как при помощи Пролога абстрактные идеи приобретают конкретные и действенные формы.

Полагаю, что среди учебников по Прологу эта книга окажется особенно популярной, и не только из-за своих хороших примеров, но также из-за

целого ряда других своих привлекательных черт:

- тщательно составленные резюме появляются на всем протяжении книги;
- все вводимые понятия подкрепляются многочисленными упражнениями;
- процедуры выборки элементов структур подводят нас к понятию абстракции данных;
- обсуждение вопросов стиля и методологии программирования занимает целую главу;
- автор не только показывает приятные свойства языка, но и со всей откровенностью обращает наше внимание на трудные проблемы, возникающие при программировании на Прологе.

Все это говорит о том, что перед нами прекрасно написанная, увлекательная и полезная книга.

Патрик Г. Уинстон

Кеймбридж, Массачусетс
Январь 1986

Посвящается Бранке, Андрею и Тадею

ПРЕДИСЛОВИЕ АВТОРА

Язык программирования Пролог базируется на ограниченном наборе механизмов, включающих в себя сопоставление образцов, древовидное представление структур данных и автоматический возврат. Этот небольшой набор образует удивительно мощный и гибкий программный аппарат. Пролог особенно хорошо приспособлен для решения задач, в которых фигурируют объекты (в частности, структуры) и отношения между ними. Например, в качестве легкого упражнения, можно попробовать выразить на Прологе пространственные отношения между объектами, изображенными на обложке этой книги. Пример такого отношения: верхний шар расположен дальше, чем левый шар. Нетрудно также сформулировать и более общее положение в виде следующего правила: если X ближе к наблюдателю, чем Y, а Y – ближе, чем Z, то объект X находится ближе, чем Z. Пользуясь правилами и фактами, пролог-система может проводить рассуждения относительно имеющихся пространственных отношений и, в частности, проверить, насколько они согласуются с вышеуказанным общим правилом. Все эти возможности придают Прологу черты мощного языка для решения задач искусственного интеллекта, а также любых задач, требующих нечислового программирования.

Само название *Пролог* есть сокращение, означающее *программирование в терминах логики*. Идея использовать логику в качестве языка программирования возникла впервые в начале 70-х годов. Первыми исследователями, разрабатывавшими эту идею, были Роберт Ковальский из Эдинбурга (теоретические аспекты), Маартен ван Эмден из Эдинбурга (экспериментальная демонстрационная система) и Ален Колмероэ из Марселя (реализация). Сегодняшней

своей популярности Пролог во многом обязан эффективной реализации этого языка, полученной в Эдинбурге Дэвидом Уорреном в середине 70-х годов.

Поскольку Пролог уходит своими корнями в математическую логику, его преподавание часто начинают с изложения логики. Однако такое введение в Пролог, насыщенное математическими понятиями, приносит мало пользы в том случае, когда Пролог изучается в качестве практического инструмента программирования. Поэтому в данной книге мы не будем заниматься математическими аспектами этого языка, вместо этого мы сосредоточим свое внимание на навыках использования базовых механизмов Пролога для решения целого ряда содержательных задач. В то время, как традиционные языки программирования являются процедурно-ориентированными, Пролог основан на описательной или *декларативной* точке зрения на программирование. Это свойство Пролога коренным образом меняет программистское мышление и делает обучение программированию на Прологе увлекательным занятием, требующим определенных интеллектуальных усилий.

В первой части книги содержится введение в Пролог, в ней показано, как составлять программы на Прологе. Во второй части демонстрируется, как мощные средства языка применяются в некоторых областях искусственного интеллекта, таких как, например, решение задач, эвристический поиск, экспертные системы, машинные игры и системы, управляемые образцами. В этой части излагаются фундаментальные методы в области искусственного интеллекта. Далее они прорабатываются достаточно глубоко для того, чтобы реализовать их на Прологе и получить готовые программы. Эти программы можно использовать в качестве «кирпичиков» для построения сложных прикладных систем. В книге рассматриваются также вопросы обработки таких сложных структур данных, как графы и деревья, хотя эти вопросы, строго говоря, и не имеют прямого отношения к искусственному интеллекту. В программах искусственного интеллекта методы обработки структур применяются довольно часто, и, реализуя их, читатель приобретет самые общие навыки программирования на Прологе. В книге особое внимание уделяется простоте и ясности составляемых программ. Повсеместно мы стремились избегать программистских «хитростей», повышающих эф-

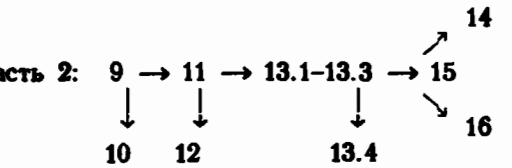
фективность за счет учета особенностей конкретной реализации Пролога.

Эта книга предназначена для тех, кто изучает Пролог и искусственный интеллект. Материал книги можно использовать в курсе лекций по искусственному интеллекту, ориентированном на прологовскую реализацию. Предполагается, что читатель имеет общее представление о вычислительных машинах, но предварительные знания в области искусственного интеллекта необязательны. От читателя не требуется также какого-либо программистского опыта. Дело в том, что богатый программистский опыт вместе с приверженностью к традиционному процедурному программированию (например, на Паскале) может стать помехой при изучении Пролога, требующего свежего программистского мышления.

Среди различных диалектов Пролога наиболее широко распространен так называемый эдинбургский синтаксис (или синтаксис DEC-10), который мы и принимаем в данной книге. Для того, чтобы обеспечить совместимость с различными реализациями Пролога, мы используем в книге сравнительно небольшое подмножество встроенных средств, имеющихся во многих вариантах Пролога.

Как читать эту книгу? В первой части порядок чтения естественным образом совпадает с порядком изложения, принятым в книге. Впрочем, часть разд. 2.4, в которой дается более формальное описание процедурной семантики Пролога, можно опустить. В главе 4 приводятся примеры программ, которые можно читать только выборочно. Вторая часть книги допускает более гибкий порядок чтения, поскольку различные главы этой части предполагаются взаимно независимыми. Однако некоторые из тем было бы естественным прочесть раньше других – это относится к основным понятиям, связанным со структурами данных (гл. 9), и к базовым стратегиям поиска (гл. 11 и 13). В приведенной ниже диаграмме показана наиболее естественная последовательность чтения глав.

Часть 1: 1 → 2 → 3 → 4 (выборочно) → 5
→ 6 → 7 → 8



Существует целый ряд исторически сложившихся и противоречащих друг другу взглядов на Пролог. Пролог быстро завоевал популярность в Европе как практический инструмент программирования. В Японии Пролог оказался в центре разработки компьютеров пятого поколения. С другой стороны, в связи с определенными историческими факторами, в США Пролог получил признание несколько позднее. Один из этих факторов был связан с предварительным знакомством с Микроплеинером, языком, близким к логическому программированию, но реализованным не эффективно. Этот отрицательный опыт, относящийся к Микропленнеру, был неоправданно распространен и на Пролог, но позднее, после появления эффективной реализации, предложенной Дэвидом Уорреном, это предубеждение было убедительно снято. Определенная сдержанность по отношению к Прологу объяснялась также существованием «ортодоксальной школы» логического программирования, сторонники которой настаивали на использовании чистой логики, не запятнанной добавлением практически полезных внелогических средств. Практикам в области применения Пролога удалось изменить эту бескомпромиссную позицию и принять более прагматический подход, позволивший удачно сочетать декларативный принцип с традиционным – процедурным. И наконец, третьим фактором, приведшим к задержке признания Пролога, явилось то обстоятельство, что в США в течение долгого времени Лисп не имел серьезных конкурентов среди языков искусственного интеллекта. Понятно поэтому, что в исследовательских центрах с сильными лисповскими традициями возникало естественное противодействие Прологу. Но со временем соперничество между Прологом и Лиспом потеряло свою остроту, и в настоящее время многие считают, что оптимальный подход состоит в сочетании идей, лежащих в основе этих двух языков.

Благодарности

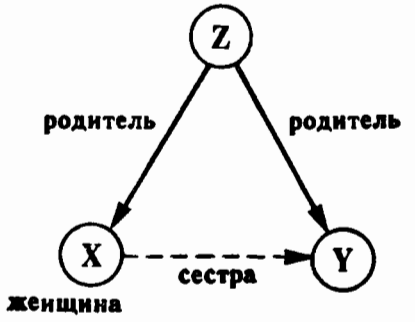
Интерес к Прологу впервые возник у меня под влиянием Дональда Мики. Я благодарен также Лоренсу Берду, Фернандо Перейра и Дэвиду Г. Уоррену, входившим в свое время в эдинбургскую группу разработчиков Пролога, за их советы по составлению программ и многочисленные дискуссии. Чрезвычайно полезными были замечания и предложения, высказанные Эндрью Макгеттриком и Патриком Уинстоном. Среди прочитавших рукопись книги и сделавших ценные замечания были также Игорь Кононенко, Таня Маярон, Игорь Мозетик, Тимоти Нибблетт и Фрэнк Зердин. Мне бы хотелось также поблагодарить Дебру Майсон-Этерингтон и Саймона Пламтри из издательства Эддисон-Уэсли за труд, вложенный в издание этой книги. И наконец, эта книга не могла бы появиться на свет без стимулирующего влияния творческой деятельности всего международного сообщества специалистов по логическому программированию.

Иван Братко

Институт Тьюринга, Глазго
Январь 1986

Часть 1

ЯЗЫК ПРОЛОГ



1 ОБЩИЙ ОБЗОР ЯЗЫКА ПРОЛОГ

В этой главе на примере конкретной программы рассматриваются основные механизмы Пролога. Несмотря на то что материал излагается в основном неформально, здесь вводятся многие важные понятия.

1.1. Пример программы: родственные отношения

Пролог – это язык программирования, предназначенный для обработки символьной нечисловой информации. Особенно хорошо он приспособлен для решения задач, в которых фигурируют объекты и отношения между ними. На рис. 1.1 представлен пример – родственные отношения. Тот факт, что Том является родителем Боба, можно записать на Прологе так:

```
родитель( том, боб).
```

Здесь мы выбрали **родитель** в качестве имени отношения, **том** и **боб** – в качестве аргументов этого отношения. По причинам, которые станут понятны позднее, мы записываем такие имена, как **том**, начиная со строчной буквы. Все дерево родственных отношений рис. 1.1 описывается следующей пролог-программой:

```
родитель( пам, боб).
родитель( том, боб).
родитель( том, лиз).
родитель( боб, энн).
родитель( боб, пат).
родитель( пам, джим).
```

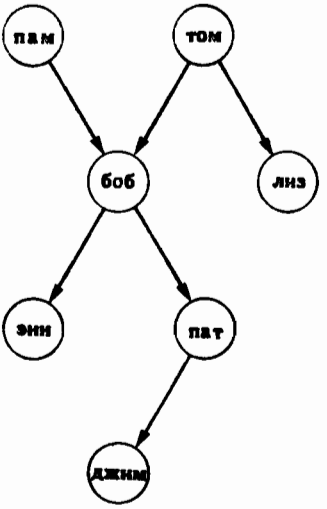


Рис. 1.1. Дерево родственных отношений.

Эта программа содержит шесть *предложений*. Каждое предложение объявляет об одном факте наличия отношения **родитель**.

После ввода такой программы в пролог-систему последней можно будет задавать вопросы, касающиеся отношения родитель. Например, является ли Боб родителем Пат? Этот вопрос можно передать пролог-системе, набрав на клавиатуре терминала:

?- родитель(боб, пат).

Найдя этот факт в программе, система ответит

yes (да)

Другим вопросом мог бы быть такой:

?- родитель(лиз, пат).

Система ответит

no (нет),

поскольку в программе ничего не говорится о том, является ли Лиз родителем Пат. Программа ответит «нет» и на вопрос

?- родитель(том, бен).

потому, что имя Бен в программе даже не упоминается.

Можно задавать и более интересные вопросы. Например: «Кто является родителем Лиз?»

?- родитель(X, лиз).

На этот раз система ответит не просто «да» или «нет». Она скажет нам, каким должно быть значение X (ранее неизвестное), чтобы вышеприведенное утверждение было истинным. Поэтому мы получим ответ:

X = том

Вопрос «Кто дети Боба?» можно передать прологсистеме в такой форме:

?- родитель(боб, X).

В этом случае возможно несколько ответов. Сначала система сообщит первое решение:

X = энн

Возможно, мы захотим увидеть и другие решения. О нашем желании мы можем сообщить системе (во многих реализациях для этого надо набрать точку с запятой), и она найдет другой ответ:

X = пат

Если мы потребуем дальнейших решений, система ответит «нет», поскольку все решения исчерпаны.

Нашей программе можно задавать и более общие вопросы: «Кто чей родитель?» Приведем другую формулировку этого вопроса:

Найти X и Y такие, что X – родитель Y.

На Прологе это записывается так:

?- родитель(X, Y).

Система будет по очереди находить все пары вида «родитель-ребенок». По мере того, как мы будем требовать от системы новых решений, они будут выводиться на экран одно за другим до тех пор, пока все они не будут найдены. Ответы выводятся следующим образом:

```
X = пам
Y = боб;

X = том
Y = боб;

X = том
Y = лиз;

. . .
```

Мы можем остановить поток решений, набрав, например, точку вместо точки с запятой (выбор конкретного символа зависит от реализации).

Нашей программе можно задавать и еще более сложные вопросы, скажем, кто является родителем родителя Джима? Поскольку в нашей программе прямо не сказано, что представляет собой отношение **родительродителя**, такой вопрос следует задавать в два этапа, как это показано на рис. 1.2.

(1) Кто родитель Джима? Предположим, что это некоторый Y.
(2) Кто родитель Y? Предположим, что это некоторый X.

Такой составной вопрос на Прологе записывается в виде последовательности двух простых вопросов:

```
?- родитель( Y, джим), родитель( X, Y).
```

Ответ будет:

```
X = боб
Y = пат
```

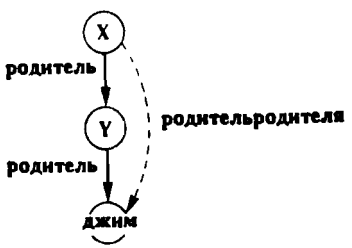


Рис. 1.2. Отношение **родительродителя**, выраженное через композицию двух отношений **родитель**.

Наш составной вопрос можно интерпретировать и так: «Найти X и Y, удовлетворяющие следующим двум требованиям»:

родитель(Y, джим) и **родитель(X, Y)**

Если мы поменяем порядок этих двух требований, то логический смысл останется прежним:

родитель(X, Y) и **родитель(Y, джим)**

Этот вопрос можно задать нашей пролог-системе и в такой форме:

?- родитель(X, Y), родитель(Y, джим).

При этом результат будет тем же.

Таким же образом можно спросить: «Кто внуки Тома?»

?- родитель(том, X), родитель(X,Y).

Система ответит так:

```
X = боб
Y = эин;

X = боб
Y = пат
```

Следующим вопросом мог бы быть такой: «Есть ли у Энн и Пат общий родитель?» Его тоже можно выразить в два этапа:

(1) Какой X является родителем Энн?
(2) Является ли (тот же) X родителем Пат?

Соответствующий запрос к пролог-системе будет тогда выглядеть так:

?- родитель(X, эин), родитель(X, пат).

Ответ:

X = боб

Наша программа-пример помогла проиллюстрировать некоторые важные моменты:

- На Прологе легко определить отношение, подобное отношению **родитель**, указав *n*-ку объектов, для которых это отношение выполняется.

- Пользователь может легко задавать пролог-системе вопросы, касающиеся отношений, определенных в программе.
- Пролог-программа состоит из предложений. Каждое предложение заканчивается точкой.
- Аргументы отношения могут быть (среди прочего): конкретными объектами, или константами (такими, как том и энн), яли абстрактными объектами, такими, как X и Y. Объекты первого типа называются *атомами*. Объекты второго типа – *переменными*.
- Вопросы к системе состоят из одиого или более целевых утверждеиий (или кратко – *целей*). Последовательиость целей, такая как

  родитель(X, энн), родитель(X, пат)

  означает конъюнкцию этих целевых утверждений:

  X – родитель Эни *и*
  X – родитель Пат.

  Пролог-система рассматривает вопросы как цели, к достнженню которых нужио стремиться.
- Ответ на вопрос может оказаться или положительным или отрицательным в зависимости от того, может ли быть соответствующая цель достигнута или нет. В случае положительного ответа мы говорим, что соответствующая цель *достижима* и *успешна*. В противном случае цель *недостижима, имеет неуспех* или *терпит неудачу*.
- Если на вопрос существует несколько ответов, пролог-система найдет столько из иих, сколько пожелает пользователь.

Упражнения

1.1. Считая, что отиошение **родитель** определеио так же, как и раньше в даином разделе (см. рис. 1.1), найдите, какими будут ответы пролог-

системы на следующие вопросы:

(a) **? – родитель(джим, X).**

(b) **? – родитель(X, джим).**

(c) **? – родитель(пам,X), родитель(X, пат).**

(d) **? – родитель(пам,X), родитель(X, Y), родитель(Y, джим).**

1.2. Сформулируйте на Прологе следующие вопросы об отношении **родитель**:

(a) Кто родитель Пат?
(b) Есть ли у Лиз ребенок?
(c) Кто является родителем родителя Пат?

1.2. Расширение программы-примера с помощью правил

Нашу программу-пример можно легко расширить многими интересными способами. Давайте сперва добавим информацию о том, каков пол людей, участвующих в отношении **родитель**. Это можно сделать, просто добавив в нее следующие факты:

```
женщина( пам).
мужчина( том).
мужчина( боб).
женщина( лиз).
женщина( пат).
женщина( энн).
мужчина( джим).
```

Мы ввели здесь два новых отношения – **мужчина** и **женщина**. Эти отношения – унарные (или одноместные). Бинарное отношение, такое как **родитель**, определяет отношение между *двумя* объектами; унарные же можно использовать для объявления наличия (отсутствия) простых свойств у объектов. Первое из приведенных выше предложений читается так: Пам – женщина. Можно было бы выразить информацию, представляемую этими двумя унарными отношениями

(мужчина и женщина), по-другому - с помощью одного бинарного отношения **пол**. Тогда новый фрагмент нашей программы выглядел бы так:

```
пол( пам, женский).
пол( том, мужской).
пол( боб, мужской).
. . .
```

В качестве дальнейшего расширения нашей программы-примера давайте введем отношение **отпрыск**, которое обратно отношению **родитель**. Можно было бы определить **отпрыск** тем же способом, что и **родитель**, т.е. представив список простых фактов наличия этого отношения для конкретных пар объектов, таких, что один является отпрыском другого. Например:

```
отпрыск( лиз, том).
```

Однако это отношение можно определить значительно элегантнее, использовав тот факт, что оно обратно отношению **родитель**, которое уже определено. Такой альтернативный способ основывается на следующем логическом утверждении:

Для всех X и Y
    Y является отпрыском X, если
    X является родителем Y.

Эта формулировка уже близка к формализму, принятому в Прологе. Вот соответствующее прологовское предложение, имеющее тот же смысл:

```
отпрыск( Y, X) :- родитель( X, Y).
```

Это предложение можно прочитать еще и так:

Для всех X и Y,
    если X - родитель Y, то
    Y - отпрыск X.

Такие предложения Пролога, как

```
отпрыск( Y, X) :- родитель( X, Y).
```

называются *правилами*. Есть существенное различие между фактами и правилами. Факт, подобный факту

```
родитель( том, лиз).
```

это нечто такое, что всегда, безусловно истинно. Напротив, правила описывают утверждения, которые

могут быть истинными, только если выполнено некоторое условие. Поэтому можно сказать, что правила имеют

- условную часть (правая половина правила) и
- часть вывода (левая половина правила).

Вывод называют также *головой* предложения, а условную часть – его *телом*. Например:

```
отпрыск( Y, X) :- родитель( X, Y).
     голова              тело
```

Если условие **родитель(X, Y)** выполняется (оно истинно), то логическим следствием из него является утверждение **отпрыск(Y, X)**.

На приведенном ниже примере проследим, как в действительности правила используются Прологом. Спросим нашу программу, является ли Лиз отпрыском Тома:

```
? - отпрыск( лиз, том).
```

В программе нет фактов об отпрысках, поэтому единственный способ ответить на такой вопрос – это применить правило о них. Правило универсально в том смысле, что оно применимо к любым объектам X и Y, следовательно, его можно применить и к таким конкретным объектам, как **лиз** и **том**. Чтобы это сделать, нужно вместо Y подставить в него **лиз**, а вместо X – **том**. В этом случае мы будем говорить, что переменные X и Y конкретизируются:

```
X = том   и   Y = лиз
```

После конкретизации мы получаем частный случай нашего общего правила. Вот он:

```
отпрыск( лиз, том) :- родитель( том, лиз).
```

Условная часть приняла вид:

```
родитель( том, лиз)
```

Теперь пролог-система попытается выяснить, выполняется ли это условие (является ли оно истинным). Для этого исходная цель

```
отпрыск( лиз, том)
```

заменяется подцелью

родитель(том, лиз)

Эта (новая) цель достигается тривиально, поскольку такой факт можно найти в нашей программе. Это означает, что утверждение, содержащееся в выводе правила, также истинно, и система ответит на вопрос **yes** (да).

Добавим теперь в нашу программу-пример еще несколько родственных отношений. Определение отношения **мать** может быть основано на следующем логическом утверждении:

Для всех X и Y
X является матерью Y, если
X является родителем Y и
X- женщина.

На Пролог это переводится в виде такого правила:

мать(X, Y) :- родитель(X, Y), женщина(X).

Запятая между двумя условиями указывает на конъюнкцию условий. Это означает, что они должны быть выполнены оба одновременно.

Рис. 1.3. Графы отношений **родительродителя, мать и отпрыск**, определенных через другие отношения.

Такие отношения как **родитель, отпрыск** и **мать** можно изобразить в виде диаграмм, приведенных на рис. 1.3. Они нарисованы с учетом следующих соглашений. Вершины графа соответствуют объектам, т.е. аргументам отношений. Дуги между вершинами соответствуют бинарным (двуместным) отношениям. Дуги направлены от первого аргумента к второму. Унарные отношения на диаграмме изображаются просто помет-

кой соответствующих объектов именем отношения. Отношения, определяемые через другие отношения, представлены штриховыми дугами. Таким образом, любую диаграмму следует понимать так: если выполнены отношения, изображенные сплошными дугами, тогда и отношение, изображенное штриховой дугой, тоже выполнено. В соответствии с рис. 1.3, отношение **родительродителя** можно сразу записать на Прологе:

```
родительродителя( X, Z) :-  родитель( X, Y),
                            родитель( Y, Z).
```

Здесь уместно сделать несколько замечаний о внешнем виде нашей программы. Пролог дает почти полную свободу расположения текста на листе. Так что можно вставлять пробелы и переходить к новой строке в любом месте текста по вкусу. Вообще мы хотим сделать так, чтобы наша программа имела красивый и аккуратный вид, а самое главное, легко читалась. Для этого мы часто будем помещать голову предложения и каждую цель на отдельной строке. При этом цели мы будем писать с отступом, чтобы сделать разницу между головой и целями более заметной. Например, правило **родительродителя** в соответствии с этими соглашениями запишется так:

```
родительродителя( X, Z) :-
    родитель( X, Y),
    родитель( Y, Z).
```

На рис. 1.4 показано отношение **сестра**:

Для любых X и Y
X является сестрой Y, если
(1) у X и Y есть общий родитель, и
(2) X – женщина.

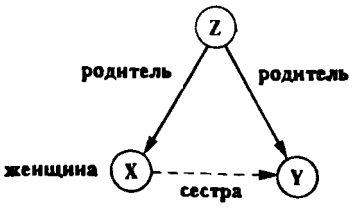


Рис. 1.4. Определение отношения **сестра**.

Граф на рис. 1.4 можно перевести на Пролог так:

```
сестра( X, Y) :-
    родитель( Z, X),
    родитель( Z, Y),
    женщина( X).
```

Обратите внимание на способ, с помощью которого выражается требование «у X и Y есть общий родитель». Была использована следующая логическая формулировка: «некоторый Z должен быть родителем X и *этот же самый* Z должен быть родителем Y». По-другому, менее красиво, можно было бы сказать так: «Z1 – родитель X, Z2 – родитель Y и Z1 равен Z2».

Теперь можно спросить:

```
?- сестра( эми, пат).
```

Как и ожидается, ответ будет «yes» (да) (см. рис. 1.1). Мы могли бы заключить отсюда, что определенное нами отношение **сестра** работает правильно. Тем не менее в нашей программе есть маленькое упущение, которое обнаружится, если задать вопрос: «Кто является сестрой Пат?»

```
?- сестра( X, пат).
```

Система найдет два ответа, один из которых может показаться неожиданным:

```
X = эми;
X = пат
```

Получается, что Пат – сестра себе самой?! Наверное, когда мы определяли отношение **сестра**, мы не имели этого ввиду. Однако ответ Пролога совершенно логичен, поскольку он руководствовался нашим правилом, а это правило ничего не говорит о том, что, если X – сестра Y, то X и Y не должны совпадать. Пролог (с полным правом) считает, что X и Y могут быть одним и тем же объектом и в качестве следствия из этого делает вывод, что любая женщина, имеющая родителя, является сестрой самой себе.

Чтобы исправить наше правило о сестрах, его нужно дополнить утверждением, что X и Y должны различаться. В следующих главах мы увидим, как это можно сделать, в данный же момент мы предположим, что отношение **различны** уже известно пролог-системе и что цель

различны(X, Y)

достигается тогда и только тогда, когда X и Y не равны. Усовершенствованное правило для отношения **сестра** примет тогда следующий вид:

```
сестра( X,Y) :-
    родитель( Z, X),
    родитель( Z, Y),
    женщина( X),
    различны( X, Y).
```

Некоторые важные моменты этого раздела:

- Пролог-программы можно расширять, добавляя в них новые предложения.
- Прологовские предложения бывают трех типов: *факты*, *правила* и *вопросы*.
- *Факты* содержат утверждения, которые являются всегда, безусловно верными.
- *Правила* содержат утверждения, истинность которых зависит от некоторых условий.
- С помощью *вопросов* пользователь может спрашивать систему о том, какие утверждения являются истинными.
- Предложения Пролога состоят из *головы* и *тела*. Тело – это список *целей*, разделенных запятыми. Запятая понимается как конъюнкция.
- Факты – это предложения, имеющие пустое тело. Вопросы имеют только тело. Правила имеют голову и (непустое) тело.
- По ходу вычислений вместо переменной может быть подставлен другой объект. Мы говорим в этом случае, что переменная *конкретизирована*.
- Предполагается, что на переменные действует квантор всеобщности, читаемый как «для всех...». Однако для переменных, появляющихся только в теле, возможны и другие формулировки. Например,

  имеетребенка(X) :- родитель(X, Y).

  можно прочитать двумя способами:

(а) *Для всех* X и Y,
если X – отец Y, то
X имеет ребенка.

(б) *Для всех* X,
X имеет ребенка, если
существует *некоторый* Y, такой, что
X – родитель Y.

Упражнения

1.3. Оттранслируйте следующие утверждения в правила на Прологе:

(а) Всякий, кто имеет ребенка, – счастлив (введите одноаргументное отношение **счастлив**).

(b) Всякий X, имеющий ребенка, у которого есть сестра, имеет двух детей (введите новое отношение **иметьдвухдетей**).

1.4. Определите отношение **внук**, используя отношение **родитель**. Указание: оно будет похоже на отношение **родительродителя** (см. рис. 1.3).

1.5. Определите отношение **тетя(X, Y)** через отношение **родитель** и **сестра**. Для облегчения работы можно сначала изобразить отношение **тетя** в виде диаграммы по типу тех, что изображены на рис. 1.3.

1.3. Рекурсивное определение правил

Давайте добавим к нашей программе о родственных связях еще одно отношение – **предок**. Определим его через отношение **родитель**. Все отношение можно выразить с помощью двух правил. Первое правило будет определять непосредственных (ближайших) предков, а второе – отдаленных. Будем говорить, что некоторый X является отдаленным предком некоторого Z, если между X и Z существует цепочка людей, связанных

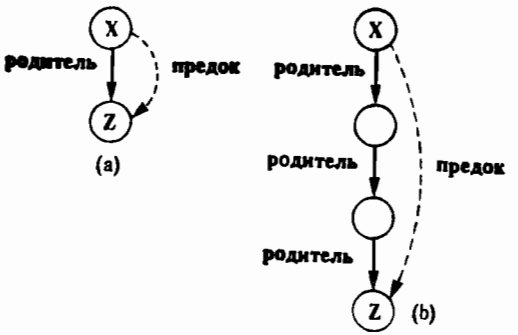


Рис. 1.5. Пример отношения **предок**:
(a) **X** – *ближайший* предок **Z**; (b) **X** – отдаленный предок **Z**.

между собой отношением родитель–ребенок, как показано на рис.1.5. В нашем примере на рис. 1.1 Том – ближайший предок Лиз и отдаленный предок Пат.

Первое правило простое и его можно сформулировать так:

Для всех X и Z,
X – предок Z, если
X – родитель Z.

Это непосредственно переводится на Пролог как

```
предок( X, Z) :-
    родитель( X, Z).
```

Второе правило сложнее, поскольку построение цепочки отношений **родитель** может вызвать некоторые трудности. Один из способов определения отдаленных родственников мог бы быть таким, как показано на рис. 1.6. В соответствии с ним отношение **предок** определялось бы следующим множеством предложений:

```
предок( X, Z) :-
    родитель( X, Z).

предок( X, Z) :-
    родитель( X, Y),
    родитель( Y, Z).
```

```
предок( X, Z) :-
     родитель( X, Y1),
     родитель( Y1, Y2),
     родитель( Y2, Z).

предок( X, Z) :-
     родитель( X, Y1),
     родитель( Y1, Y2),
     родитель( Y2, Y3),
     родитель( Y3, Z).

. . .
```

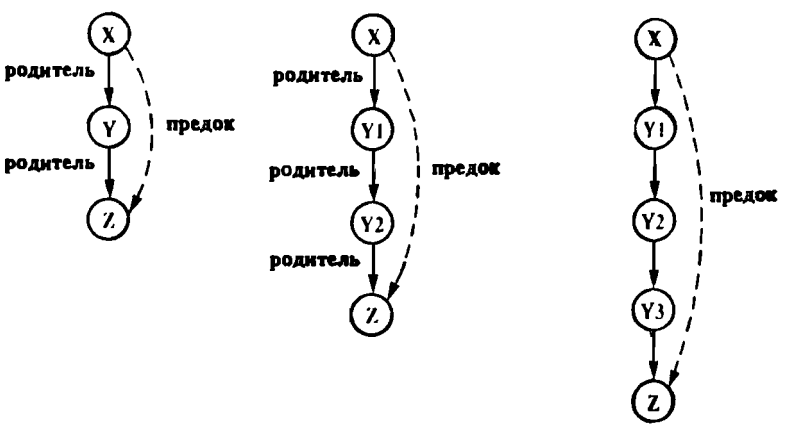


Рис. 1.6. Пары предок–потомок, разделенных разным числом поколений.

Эта программа длинна и, что более важно, работает только в определенных пределах. Она будет обнаруживать предков лишь до определенной глубины фамильного дерева, поскольку длина цепочки людей между предком и потомком ограничена длиной наших предложений в определении отношения.

Существует, однако, корректная и элегантная формулировка отношения предок – корректная в том смысле, что будет работать для предков произвольной отдаленности. Ключевая идея здесь – определить отношение предок через него самого. Рис 1.7 иллюстрирует эту идею:

Для всех X и Z,
    X - предок Z, если
    существует Y, такой, что
    (1)   X - родитель Y и
    (2)   Y - предок Z.

Предложение Пролога, имеющее тот же смысл, записывается так:

```
предок( X, Z) :-
    родитель( X, Y),
    предок( Y, Z).
```

Теперь мы построили полную программу для отношения предок, содержащую два правила: одно для ближайших предков и другое для отдаленных предков. Здесь приводятся они оба вместе:

```
предок( X, Z) :-
    родитель( X, Z).

предок( X, Z) :-
    родитель( X, Y),
    предок( Y, Z).
```

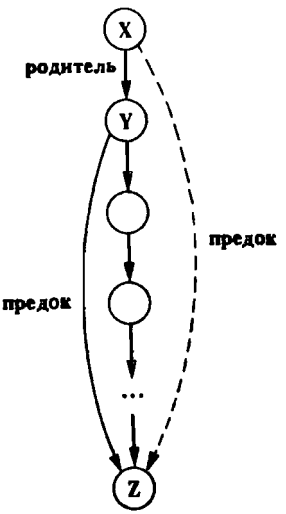


Рис. 1.7. Рекурсивная формулировка отношения **предок**.

Ключевым моментом в данной формулировке было использование самого отношения **предок** в его определении. Такое определение может озадачить – допустимо ли при определении какого-либо понятия использовать его же, ведь оно определено еще не полностью. Такие определения называются *рекурсивными*. Логически они совершенно корректны и понятны; интуитивно это ясно, если посмотреть на рис. 1.7. Но будет ли в состоянии пролог-система использовать рекурсивные правила? Оказывается, что пролог-система очень легко может обрабатывать рекурсивные определения. На самом деле, рекурсия – один из фундаментальных приемов программирования на Прологе. Без рекурсии с его помощью невозможно решать задач сколько-нибудь ощутимой сложности.

Возвращаясь к нашей программе, можно теперь задать системе вопрос: «Кто потомки Пам?» То есть: «Кто тот человек, чьим предком является Пам ?»

```
?- предок( пам, X).

X = боб;
X = энн;
X = пат;
X = джим
```

Ответы системы, конечно, правильны, и они логически вытекают из наших определений отношений **предок** и **родитель**. Возникает, однако, довольно важный вопрос: «*Как* в действительности система использует программу для отыскания этих ответов?»

Неформальное объяснение того, как система это делает, приведено в следующем разделе. Но сначала давайте объединим все фрагменты нашей программы о родственных отношениях, которая постепенно расширялась по мере того, как мы вводили в нее новые факты и правила. Окончательный вид программы показан на рис. 1.8.

При рассмотрении рис. 1.8 следует учесть два новых момента: первый касается понятия «процедура», второй – комментариев в программах.

Программа, приведенная на рис. 1.8, определяет несколько отношений – **родитель**, **мужчина**, **женщина**, **предок** и т.д. Отношение **предок**, например, определено с помощью двух предложений. Будем говорить, что эти два предложения входят в состав отношения

```
родитель( пам, боб).          % Пам - родитель Боба
родитель( том, боб).
родитель( том, лиз).
родитель( боб, энн).
родитель( боб, пат).
родитель( пат, джим).

женщина( пам).                % Пам - женщина
мужчина( том).                % Том - мужчина
мужчина( боб).
женщина( лиз).
женщина( энн).
женщина( пат).
мужчина( джим).

отпрыск( Y, X) :-             % Y - отпрыск X, если
    родитель( X, Y).          % X - родитель Y

    мать( X, Y) :-            % X - мать Y, если
        родитель( X, Y),      % X - родитель Y и
        женщина( X).          % X - женщина

родительродителя( X, Z) :-
                  % X - родитель родителя Z, если
        родитель( X, Y),      % X - родитель Y и
        родитель( Y, Z).      % Y - родитель Z

  сестра( X, Y) :-            % X - сестра Y
        родитель( Z, X),
        родитель( Z, Y)
                  % X и Y имеют общего родителя
        женщина( X, Y),       % X - женщина и
        различны( X, Y).      % X отличается от Y

предок( X, Z) :-              % Правило пр1: X - предок Z
        родитель( X, Z).

    предок( X, Z) :-          % Правило пр2: X - предок Z
        родитель( X, Y),
        предок( Y, Z).
```

Рис. 1.8. Программа о родственных отношениях.

предок. Иногда бывает удобно рассматривать в целом все множество предложений, входящих в состав одного отношения. Такое множество называется *процедурой*.

На рис. 1.8 два предложения, входящие в состав отношения **предок**, выделены именами «пр1» и «пр2», добавленными в программу в виде *комментариев*. Эти имена будут использоваться в дальнейшем для ссылок на соответствующие правила. Вообще говоря, комментарии пролог-системой игнорируются. Они нужны лишь человеку, который читает программу. В Прологе комментарии отделяются от остального текста программы специальными скобками «/*» и «*/». Таким образом, прологовский комментарий выглядит так

```
/* Это комментарий */
```

Другой способ, более практичный для коротких комментариев, использует символ процента %. Все, что находится между % и концом строки, расценивается как комментарий:

```
% Это тоже комментарий
```

Упражнение

1.6. Рассмотрим другой вариант отношения **предок**:

```
предок( X, Z) :-
    родитель( X, Z).
предок( X, Z) :-
    родитель( Y, Z),
    предок( X, Y).
```

Верно ли и такое определение? Сможете ли Вы изменить диаграмму на рис. 1.7 таким образом, чтобы она соответствовала новому определению?

1.4. Как пролог—система отвечает на вопросы

В данном разделе приводится неформальное объяснение того, *как* пролог-система отвечает на вопросы.

Вопрос к системе – это всегда последовательность, состоящая из одной или нескольких целей. Для того, чтобы ответить на вопрос, система пытается достичь всех целей. Что значит достичь цели? Достичь цели – это значит показать, что утверждения, содержащиеся в вопросе, истинны в предположении, что все отношения программы истинны. Другими словами, достичь цели – это значит показать, что она *логически* следует из фактов и правил программы. Если вопрос содержит переменные, система должна к тому же найти конкретные объекты, которые (будучи подставленными вместо переменных) обеспечивают достижение цели. Найденные конкретизации сообщаются пользователю. Если для некоторой конкретизации система не в состоянии вывести цель из остальных предложений программы, то ее ответом на вопрос будет «нет».

Таким образом, подходящей интерпретацией пролог-программы в математических терминах будет следующая: пролог-система рассматривает факты и правила в качестве множества аксиом, а вопрос пользователя – как *теорему*; затем она пытается доказать эту теорему, т.е. показать, что ее можно логически вывести из аксиом.

Проиллюстрируем этот подход на классическом примере. Пусть имеются следующие аксиомы:

Все люди смертны.
Сократ – человек.

Теорема, логически вытекающая из этих двух аксиом:

Сократ смертен.

Первую из вышеуказанных аксиом можно переписать так:

Для всех X, если X – человек, то X смертен.

Соответственно наш пример можно перевести на Пролог следующим образом:

```
смертен( X) :- человек( X).      % Все люди смертны
человек( сократ).                % Сократ - человек
?- смертен( сократ).             % Сократ смертен?
yes              (да)
```

Более сложный пример из программы о родственных отношениях, приведенной на рис. 1.8:

?- предок(том, пат).

Мы знаем, что **родитель(боб, пат)** – это факт. Используя этот факт и правило *пр1*, мы можем сделать вывод, что утверждение **предок(боб, пат)** истинно. Этот факт получен в результате *вывода* – его нельзя найти непосредственно в программе, но можно вывести, пользуясь содержащимися в ней фактами и правилами. Подобный шаг вывода можно коротко записать

родитель(боб, пат) ==> предок(боб, пат)

Эту запись можно прочитать так: из **родитель(боб, пат)** следует **предок(боб, пат)** на основании правила *пр1*. Далее, нам известен факт **родитель(том, боб)**. На основании этого факта и выведенного факта **предок(боб, пат)** можно заключить, что, в силу правила *пр2*, наше целевое утверждение **предок(том, пат)** истинно. Весь процесс вывода, состоящий из двух шагов, можно записать так:

родитель(боб, пат) ==> предок(боб, пат)

родитель(том, боб) *и* **предок(боб, пат) ==> предок(том, пат)**

Таким образом, мы показали, *какой* может быть последовательность шагов для достижения цели, т.е. для демонстрации истинности целевого утверждения. Назовем такую последовательность цепочкой доказательства. Однако мы еще не показали *как* пролог-система в действительности строит такую цепочку.

Пролог-система строит цепочку доказательства в порядке, обратном по отношению к тому, которым мы только что воспользовались. Вместо того, чтобы начинать с простых фактов, приведенных в программе, система начинает с целей и, применяя правила, подменяет текущие цели новыми, до тех пор, пока эти новые цели не окажутся простыми фактами. Если задан вопрос

?- предок(том, пат).

система попытается достичь этой цели. Для того, чтобы это сделать, она пробует найти такое предложение в программе, из которого немедленно следует упомянутая цель. Очевидно, единственными подходящими для этого предложениями являются *пр1* и *пр2*.

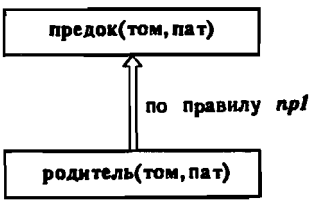


Рис. 1.9. Первый шаг вычислений. Верхняя цель истинна, если истинна нижняя.

Это правила, входящие в отношение **предок**. Будем говорить, что головы этих правил сопоставимы с целью.

Два предложения *пр1* и *пр2* описывают два варианта продолжения рассуждений для пролог-системы. Вначале система пробует предложение, стоящее в программе первым:

```
предок( X, Z) :- родитель( X, Z).
```

Поскольку цель – **предок(том, пат)**, значения переменным должны быть приписаны следующим образом:

```
X = том, Z = пат
```

Тогда исходная цель **предок(том, пат)** заменяется новой целью:

```
родитель( том, пат)
```

Такое действие по замене одной цели на другую на основании некоторого правила показано на рис. 1.9. В программе нет правила, голова которого была бы сопоставима с целью **родитель(том, пат)**, поэтому такая цель оказывается неуспешной. Теперь система делает *возврат* к исходной цели, чтобы попробовать второй вариант вывода цели верхнего уровня **предок(том, пат)**. То есть, пробуется правило *пр2*:

```
предок( X, Z) :-
    родитель( X, Y),
    предок( Y, Z).
```

Как и раньше, переменным **X** и **Z** приписываются значения:

```
X = том, Z = пат
```

В этот момент переменной Y еще не приписано никакого значения. Верхняя цель **предок(том, пат)** заменяется двумя целями:

родитель(том, Y),
предок(Y, пат)

Этот шаг вычислений показан на рис. 1.10, который представляет развитие ситуации, изображенной на рис. 1.9.

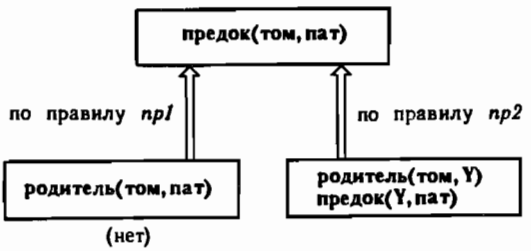


Рис. 1.10. Продолжение процесса вычислений, показанного на рис. 1.9.

Имея теперь перед собой *две* цели, система пытается достичь их в том порядке, каком они записаны. Достичь первой из них легко, поскольку она соответствует факту из программы. Процесс установления соответствия – сопоставление (унификация) вызывает приписывание переменной Y значения **боб**. Тем самым достигается первая цель, а оставшаяся превращается в

предок(боб, пат)

Для достижения этой цели вновь применяется правило *пр1*. Заметим, что это (второе) применение правила никак не связано с его первым применением. Поэтому система использует новое множество переменных правила всякий раз, как оно применяется. Чтобы указать это, мы переименуем переменные правила *пр1* для нового его применения следующим образом:

предок(X′, Z′) :–
родитель(X′, Z′).

Голова этого правила должна соответствовать нашей текущей цели **предок(боб, пат)**. Поэтому

X′ = боб, Z′ = пат

Текущая цель заменяется на

родитель(боб, пат)

Такая цель немедленно достигается, поскольку встречается в программе в качестве факта. Этот шаг завершает вычнсление, что графическн показано на рис. 1.11.

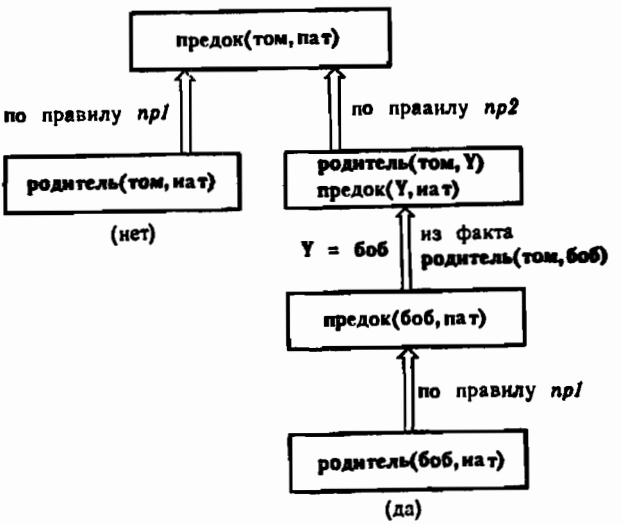


Рис. 1.11. Все шаги достиженя цели **предок(том, пат)**. Правая ветвь демонстрнрует, что цель достижима.

Графическое представленне шагов вычисления на рис. 1.11 имеет форму дерева. Вершины дерева соответствуют целям илн спнскам целей, которые требуется достнчь. Дуги между вершннами соответствуют применению (альтериативных) предложений программы, которые преобразуют цель, соответствующую одной вершине, в цель, соответствующую другой вершине. Корневая (верхняя) цель достигается тогда, когда находится путь от корня дерева (верхней вершины) к его листу, помеченному меткой «да». Лнст помечает-

ся меткой «да», если он представляет собой простой факт. Выполнение пролог-программы состоит в поиске таких путей. В процессе такого поиска система может входить и в ветви, приводящие к неуспеху. В тот момент, когда она обнаруживает, что ветвь не приводит к успеху, происходит автоматический *возврат* к предыдущей вершине, и далее следует попытка применить к ней альтернативное предложение.

Упражнение

1.7. Постарайтесь понять, как пролог-система, используя программу, приведенную на рис. 1.8, выводит ответы на указанные ниже вопросы. Попытайтесь нарисовать соответствующие диаграммы вывода по типу тех, что изображены на рис.1.9 – 1.11. Будут ли встречаться возвраты при выводе ответов на какие-либо из этих вопросов?

(a) **?- родитель(пам, боб).**
(b) **?- мать(пам, боб).**
(c) **?- родительродителя(пам, энн).**
(d) **?- родительродителя(боб, джим).**

1.5. Декларативный и процедурный смысл программ

До сих пор во всех наших примерах всегда можно было понять результаты работы программы, точно не зная, *как* система в действительности их нашла. Поэтому стоит различать два уровня смысла программы на Прологе, а именно:

- *декларативный смысл* и
- *процедурный смысл.*

Декларативный смысл касается только *отношений*, определенных в программе. Таким образом, декларативный смысл определяет, *что* должно быть результа-

том работы программы. С другой стороны, процедурный смысл определяет еще и *как* этот результат был получен, т. е. как отношения реально обрабатываются пролог-системой.

Способность пролог-системы прорабатывать многие процедурные детали самостоятельно считается одним из специфических преимуществ Пролога. Это свойство побуждает программиста рассматривать декларативный смысл программы относительно независимо от ее процедурного смысла. Поскольку результаты работы программы в принципе определяются ее декларативным смыслом, последнего (опять же в принципе) достаточно для написания программ. Этот факт имеет практическое значение, поскольку декларативные аспекты программы являются, обычно, более легкими для понимания, нежели процедурные детали. Чтобы извлечь из этого обстоятельства наибольшую пользу, программисту следует сосредоточиться главным образом на декларативном смысле и по возможности не отвлекаться на детали процесса вычислений. Последние следует в возможно большей мере предоставить самой пролог-системе.

Такой декларативный подход и в самом деле часто делает программирование на Прологе более легким, чем на таких типичных процедурно-ориентированных языках, как Паскаль. К сожалению, однако, декларативного подхода не всегда оказывается достаточно. Далее станет ясно, что, особенно в больших программах, программист не может полностью игнорировать процедурные аспекты по соображениям эффективности вычислений. Тем не менее следует поощрять декларативный стиль мышления при написании пролог-программ, а процедурные аспекты игнорировать в тех пределах, которые устанавливаются практическими ограничениями.

Резюме

- Программирование на Прологе состоит в определении отношений и в постановке вопросов, касающихся этих отношений.
- Программа состоит из предложений. Предложения бывают трех типов: *факты, правила* и *вопросы.*

- Отношение может определяться *фактами*, перечисляющими *n*-ки объектов, для которых это отношение выполняется, или же оно может определяться *правилами*.
- *Процедура* – это множество предложений об одном и том же отношении.
- *Вопросы* напоминают запросы к некоторой базе данных. Ответ системы на вопрос представляет собой множество объектов, которые удовлетворяют запросу.
- Процесс, в результате которого пролог-система устанавливает, удовлетворяет ли объект запросу, часто довольно сложен и включает в себя логический вывод, исследование различных вариантов и, возможно, *возвраты*. Все это делается автоматически самой пролог-системой и по большей части скрыто от пользователя.
- Различают два типа смысла пролог-программ: декларативный и процедурный. Декларативный подход предпочтительнее с точки зрения программирования. Тем не менее, программист должен часто учитывать также и процедурные детали.
- В данной главе были введены следующие понятия:

  предложение, факт, правило, вопрос
  голова предложения, тело предложения
  рекурсивное правило
  рекурсивное определение
  процедура
  атом, переменная
  конкретизация переменной
  цель
  цель достижима, цель успешна
  цель недостижима,
  цель имеет неуспех, цель терпит неудачу
  возврат
  декларативный смысл, процедурный смысл.

Литература

Различные реализации Пролога используют разные синтаксические соглашения. В данной книге мы применяем так называемый Эдинбургский синтаксис (его называют также синтаксисом DEC-10, поскольку он принят в известной реализации Пролога для машины DEC-10; см. Pereira и др. 1978), он используется во многих популярных пролог-системах, таких как Quintus Prolog, Poplog, CProlog, Arity/Prolog, Prolog-2 и т.д.

Bowen D. L. (1981). *DECsystem-10 Prolog User's Manual.* University of Edinburgh: Department of Artificial Intelligence.

Mellish C. and Hardy S. (1984). *Integrating Prolog in the POPLOG environment. Implementations of Prolog (J. A. Campbell, ed.).* Ellis Horwood.

Pereira F. (1982). *C-Prolog User's Manual.* University of Edinburgh: Department of Computer-Aided Architectural Design.

Pereira L. M., Pereira F., Warren D. H. D. (1978). *User's Guide to DECsystem-10 Prolog.* University of Edinburgh: Department of Artificial Intelligence.

Quintus Prolog User's Guide and Reference Manual. Palo Alto: Quintus Computer System Inc. (1985).

The Arity/Prolog Programming Laguage. Concord, Massachusetts: Arity Corporation (1986).

2 СИНТАКСИС И СЕМАНТИКА ПРОЛОГ-ПРОГРАММ

В данной главе дается систематическое изложение синтаксиса и семантики основных понятий Пролога, а также вводятся структурные объекты данных. Рассматриваются следующие темы:

- простые объекты данных (атомы, числа, переменные)
- структурные объекты
- сопоставление как основная операция над объектами
- декларативная (или непроцедурная) семантика программ
- взаимосвязь между декларативным и процедурным смыслами программ
- изменение процедурного смысла путем изменения порядка следования предложений и целей

Большая часть этих тем уже была затронута в гл. 1. Теперь их изложение будет более формальным и детализированным.

2.1. Объекты данных

На рис. 2.1 приведена классификация объектов данных Пролога. Пролог-система распознает тип объекта по его синтаксической форме в тексте программы. Это возможно благодаря тому, что синтаксис Пролога

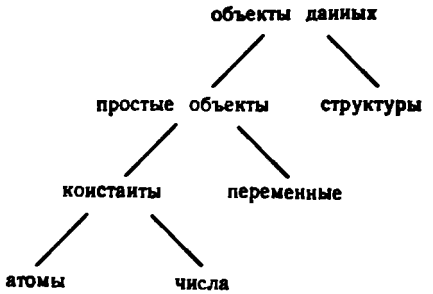


Рис. 2.1. Объекты данных Пролога.

предписывает различные формы записи для различных типов объектов данных. В гл. 1 мы уже видели способ, с помощью которого можно отличить атомы от переменных: переменные начинаются с прописной буквы, тогда как атомы – со строчной. Для того, чтобы пролог-система распознала тип объекта, ей не требуется сообщать больше никакой дополнительной информации (такой, например, как объявление типа данных).

2.1.1. Атомы и числа

В гл. 1 мы уже видели несколько простых примеров атомов и переменных. Вообще же они могут принимать более сложные формы, а именно представлять собой цепочки следующих символов:

- прописные буквы A, B, ..., Z
- строчные буквы a, b, ..., z
- цифры 0, 1, 2, ..., 9
- спецнальные символы, такие как
  + - * / < > = : . & _ ~

Атомы можно создавать тремя способами:

(1) из цепочки букв, цифр и символа подчеркивания _, начиная такую цепочку со строчной буквы:

анна
nil
x25
x_25
x_25AB
x_
x__y
альфа_бета_процедура
мисс_Джонс
сара_джонс

(2) из специальных символов:

```
<--->
======>

...
.
...
::=
```

Пользуясь атомами такой формы, следует соблюдать некоторую осторожность, поскольку часть цепочек специальных символов имеют в Прологе заранее определенный смысл. Примером может служить :- .

(3) из цепочки символов, заключенной в одинарные кавычки. Это удобно, если мы хотим, например, иметь атом, начинающийся с прописной буквы. Заключая его в кавычки, мы делаем его отличным от переменной:

'Том'
'Южная_Америка'
'Сара Джонс'

Числа в Прологе бывают целыми и вещественными. Синтаксис целых чисел прост, как это видно из следующих примеров: 1, 1313, 0, -97. Не все целые числа могут быть представлены в машине, поэтому диапазон целых чисел ограничен интервалом между некоторыми минимальным и максимальным числами, определяемыми конкретной реализацией Пролога. Обычно реализация допускает диапазон хотя бы от -16 383 до 16 383, а часто, и значительно более широкий.

Синтаксис вещественных чисел зависит от реализации. Мы будем придерживаться простых правил, видных из следующих примеров: 3.14, -0.0035, 100.2. При обычном программировании на Прологе веществен-

ные числа используются редко. Причина этого кроется в том, что Пролог – это язык, предназначенный в первую очередь для обработки символьной, а не числовой информации, в противоположность языкам типа Фортрана, ориентированным на числовую обработку. При символьной обработке часто используются целые числа, например, для подсчета количества элементов списка; нужда же в вещественных числах невелика.

Кроме отсутствия необходимости в использовании вещественных чисел в обычных применениях Пролога, существует и другая причина избегать их. Мы всегда стремимся поддерживать наши программы в таком виде, чтобы их смысл был предельно ясен. Введение вещественных чисел в некоторой степени нарушает эту ясность из-за ошибок вычислений, связанных с округлением во время выполнения арифметических действий. Например, результатом вычисления выражения 10000 + 0.0001 – 10000 может оказаться 0 вместо правильного значения 0.0001.

2.1.2. Переменные

Переменные – это цепочки, состоящие из букв, цифр и символов подчеркивания. Они начинаются с прописной буквы или с символа подчеркивания:

```
X
Результат
Объект2
Список_участников
СписокПокупок
_x23
_23
```

Если переменная встречается в предложении только один раз, то нет необходимости изобретать ей имя. Можно использовать так называемую «анонимную» переменную, которая записывается в виде одного символа подчеркивания. Рассмотрим, например, следующее правило:

```
имеетребенка( X) :- родитель( X, Y).
```

Это правило гласит: «Для всех X, X имеет ребенка,

если X является родителем некоторого Y». Здесь мы определяем свойство имеетребенка таким образом, что оно не зависит от имени ребенка. Следовательно, это как раз тот случай, когда уместно использовать анонимную переменную. Поэтому вышеприведенное правило можно переписать так:

```
имеетребенка( X) :- родитель( X, _).
```

Всякий раз, когда в предложении появляется одиночный символ подчеркивания, он обозначает новую анонимную переменную. Например, можно сказать, что существует некто, кто имеет ребенка, если существуют два объекта, такие, что один из них является родителем другого:

```
некто_имеет_ребенка :- родитель( _, _).
```

Это предложение эквивалентно следующему:

```
некто_имеет_ребенка :- родитель( X, Y).
```

Однако оно имеет совершенно другой смысл, нежели

```
некто_имеет_ребенка :- родитель( X, X).
```

Если анонимная переменная встречается в вопросе, то ее значение не выводится при ответе системы на этот вопрос. Если нас интересуют люди, имеющие детей, но не имена этих детей, мы можем просто спросить:

```
?- родитель( X, _).
```

Лексический диапазон имени – одно предложение. Это значит, что если, например, имя X15 встречается в двух предложениях, то оно обозначает две разные переменные. Однако внутри одного предложения каждое его появление обозначает одну и ту же переменную. Для констант ситуация другая: один и тот же атом обозначает один и тот же объект в любом предложении, иначе говоря, – во всей программе.

2.1.3. Структуры

Структурные объекты (или просто структуры) – это объекты, которые состоят из нескольких компонент.

Эти компоненты, в свою очередь, могут быть структурами. Например, дату можно рассматривать как структуру, состоящую из трех компонент: день, месяц, год. Хотя они и составлены из нескольких компонент, структуры в программе ведут себя как единые объекты. Для того, чтобы объединить компоненты в структуру, требуется выбрать *функтор*. Для нашего примера подойдет функтор дата. Тогда дату 1-е мая 1983 г. можно записать так:

дата(1, май, 1983)

(см. рис. 2.2).

Все компоненты в данном примере являются константами (две компоненты – целые числа и одна – атом). Компоненты могут быть также переменными или структурами. Произвольный день в мае можно представить структурой:

дата(День, май, 1983)

Заметим, что **День** является переменной и ей можно приписать произвольное значение на некотором более позднем этапе вычислений.

Такой метод структурирования данных прост и эффективен. Это является одной из причин того, почему Пролог естественно использовать для решения задач обработки символьной информации.

Синтаксически все объекты данных в Прологе представляют собой *термы*. Например,

май

и

дата(1, май, 1983)

суть термы.

Все структурные объекты можно изображать в виде деревьев (пример см. на рис. 2.2). Корнем дерева служит функтор, ветвями, выходящими из него, – компоненты. Если некоторая компонента тоже является структурой, тогда ей соответствует поддерево в дереве, изображающем весь структурный объект.

Наш следующий пример показывает, как можно использовать структуры для представления геометрических объектов (см. рис. 2.3). Точка в двумерном пространстве определяется двумя координатами; отрезок определяется двумя точками, а треугольник можно задать тремя точками. Введем следующие функторы:

точка	для точек
отрезок	для отрезков и
треугольник	для треугольников.

Рис. 2.2. Дата – пример структурного объекта:
(а) его представление в виде дерева; (б) запись на Прологе.

Тогда объекты, приведенные на рис. 2.3, можно представить следующими прологовскими термами:

```
P1 = точка(1,1)
P2 = точка(2,3)
S = отрезок( P1, P2) =
    отрезок( точка(1,1), точка(2,3) )
T = треугольник(точка(4,2), точка(6,4),
                точка(7,1) )
```

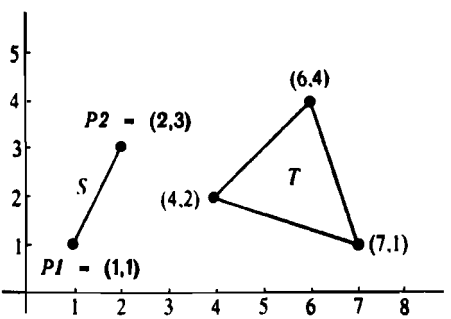


Рис. 2.3. Простые геометрические объекты.

Соответствующее представление этих объектов в виде деревьев приводится на рис. 2.4. Функтор, служащий

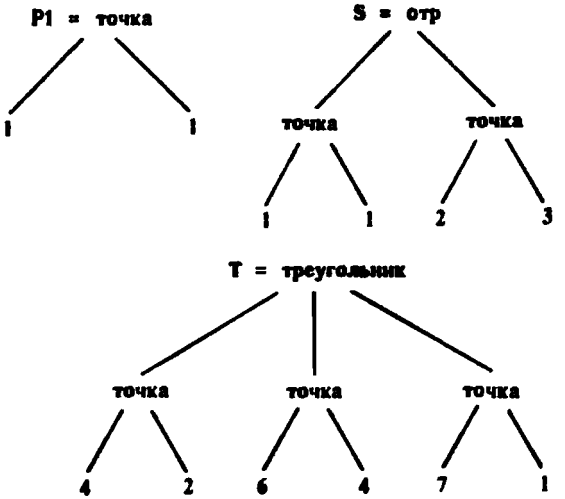


Рис. 2.4. Представление объектов с рис.2.3 в виде деревьев.

корием дерева, называется *главным функтором* терма.

Если бы в такой же программе фигурировали точки трехмерного пространства, то можно было бы для их представления использовать другой функтор, скажем точка3:

точка3(X, Y, Z)

Можно, однако, воспользоваться одним и тем же именем точка одновременно и для точек двумерного и трехмерного пространств и написать, например, так:

точка(X1, Y1) и точка(X, Y, Z)

Если одно и то же имя появляется в программе в двух различных смыслах, как в вышеупомянутом примере с точкой, то пролог-система будет различать их по числу аргументов и интерпретировать это имя как два функтора: один – двухаргументный; второй – трех. Это возможно потому, что каждый функтор определяется двумя параметрами:

(1) именем, синтаксис которого совпадает с синтаксисом атомов;

(2) *арностью* – т. е. числом аргументов.

Как уже объяснялось, все структурные объекты в Прологе – это деревья, представленные в программе термами. Рассмотрим еще два примера, чтобы показать, насколько удобно сложные объекты данных представляются с помощью прологовских термов. На рис. 2.5 показана древовидная структура, соответствующая арифметическому выражению

$(a + в)*(c - 5)$

В соответствии с введенным к настоящему моменту синтаксисом, такое выражение, используя символы *, + и – в качестве функторов, можно записать следующим образом:

```
*( +( a, в), -( c, 5) )
```

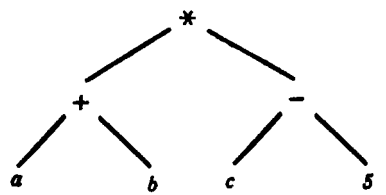


Рис.2.5. Древовидная структура, соответствующая арифметическому выражению $(a + w)*(s - 5)$.

Это, конечно, совершенно правильный прологовский терм, однако это не та форма, которую нам хотелось бы иметь при записи арифметических выражений. Хотелось бы применять обычную инфиксную запись, принятую в математике. На самом деле Пролог допускает использование инфиксной нотации, при которой символы *, + и – записываются как инфиксные операторы. Детали того, как программист может определять свои собственные операторы, мы приведем в гл. 3.

В качестве последнего примера рассмотрим некоторые простые электрические цепи, изображенные на рис. 2.6. В правой части рисунка помещены древовидные представления этих цепей. Атомы r1, r2, r3 и r4 – имена резисторов. Функторы пар и посл обозначают соответственно параллельное и последовательное соединение резисторов. Вот соответствующие прологовские термы:

```
посл( r1, r2)
пар( r1, r2)
```

```
пар( r1, пар( r2, r3) )
пар( r1, посл( пар( r2, r3), r4) )
```

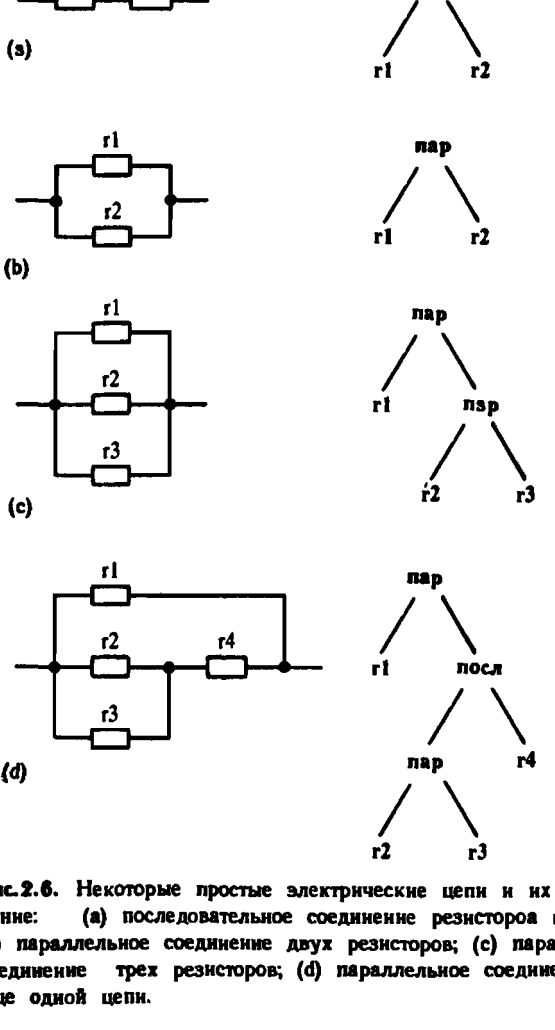


Рис. 2.6. Некоторые простые электрические цепи и их представление: (a) последовательное соединение резисторов r1 и r2; (b) параллельное соединение двух резисторов; (c) параллельное соединение трех резисторов; (d) параллельное соединение r1 и еще одной цепи.

Упражнения

2.1. Какие из следующих выражений представляют собой правильные объекты в смысле Пролога? Что это за объекты (атомы, числа, переменные, структуры)?

(a) **Диана**
(в) **диана**
(c) **'Диана'**
(d) **_диана**
(e) **'Диана едет на юг'**
(f) **едет(диана, юг)**
(g) **45**
(h) **5(X, Y)**
(i) **+(север, запад)**
(j) **три(Черные(Кошки))**

2.2. Предложите представление для прямоугольников, квадратов и окружностей в виде структурных объектов Пролога. Используйте подход, аналогичный приведенному на рис. 2.4. Например, прямоугольник можно представнть четырьмя точкамн (а может быть, только тремя точкамн). Напншнте несколько термов конкретных объектов такого тнпа с использованнем предложенного вами представления.

2.2. Сопоставление

В предыдущем разделе мы внделн, как нспользуются термы для представлення сложных объектов данных. Нанболее важной операцией над термамн является *сопоставление.* Сопоставленне само по себе может производить содержательные вычисления.

Пусть даны два терма. Будем говорить, что они *сопоставимы*, если:

(1) они идентнчны или

(2) переменным в обоих термах можно приписать в качестве значений объекты (т.е. коикретизировать их) таким образом, чтобы после под-

становки этих объектов в термы вместо переменных, последние стали идентичными.

Например, термы **дата(Д, М, 1983)** и **дата(Д1, май, Y1)** сопоставимы. Одной из конкретизаций, которая делает эти термы идентичными, является следующая:

- **Д** заменяется на **Д1**
- **М** заменяется на **май**
- **Y1** заменяется на **1983**

Более компактно такая подстановка записывается в привычной форме, т. е. в той, в которой пролог-система выводит результаты:

```
Д = Д1
М = май
Y1 = 1983
```

С другой стороны, **дата(Д, М, 1983)** и **дата(Д1, М1, 1944)** не сопоставимы, как и термы **дата(X, Y, Z)** и **точка(X, Y, Z)**.

Сопоставление – это процесс, на вход которого подаются два терма, а он проверяет, соответствуют ли эти термы друг другу. Если термы не сопоставимы, будем говорить, что этот процесс терпит *неуспех*. Если же они сопоставимы, тогда процесс находит конкретизацию переменных, делающую эти термы тождественными, и завершается *успешно*.

Рассмотрим еще раз сопоставление двух дат. Запрос на проведение такой операции можно передать системе, использовав оператор '=':

```
?- дата( Д, М, 1983) = дата( Д1, май, Y1).
```

Мы уже упоминали конкретизацию **Д = Д1, М = май, Y1 = 1983**, на которой достигается сопоставление. Существуют, однако, и другие конкретизации, делающие оба терма идентичными. Вот две из них:

```
Д = 1
Д1 = 1
М = май
Y1 = 1983

Д = третий
Д1 = третий
М = май
Y1 = 1983
```

Говорят, что эти конкретизации являются *менее общими* по сравнению с первой, поскольку они ограничивают значения переменных Д и Д1 в большей степени, чем это необходимо. Для того, чтобы сделать оба терма нашего примера идентичными, важно лишь, чтобы Д и Д1 имели одно и то же значение, однако само это значение может быть произвольным. Сопоставление в Прологе всегда дает *наиболее общую конкретизацию*. Таковой является конкретизация, которая ограничивает переменные в наименьшей степени, оставляя им, тем самым, наибольшую свободу для дальнейших конкретизаций, если потребуются новые сопоставления. В качестве примера рассмотрим следующий вопрос:

```
?- дата( Д, М, 1983) = дата( Д1, май, Y1),
   дата( Д, М, 1983) = дата( 15, М, Y).
```

Для достижения первой цели система припишет переменным такие значения:

```
Д  = Д1
М  = май
Y1 = 1983
```

После достижения второй цели, значения переменных станут более конкретными, а именно:

```
Д  = 15
Д1 = 15
М  = май
Y1 = 1983
Y  = 1983
```

Этот пример иллюстрирует также и тот факт, что переменным по мере вычисления последовательности целей приписываются обычно все более и более конкретные значения.

Общие правила выяснения, сопоставимы ли два терма S и T, таковы:

---

(1) **Если S и T – константы, то S и T сопоставимы, только если они являются одним и тем же объектом.**

(2) **Если S – переменная, а T – произвольный объект, то они сопоставимы, и S приписывается значение T. Наоборот, если T –**

переменная, а S – произвольный объект, то T приписывается значение S.

(3) Если S и T – структуры, то они сопоставимы, только если

(a) S и T имеют одинаковый главный функтор и

(b) все их соответствующие компоненты сопоставимы.

Результирующая конкретизация определяется сопоставлением компонент.

Последнее из этих правил можно наглядно представить себе, рассмотрев древовидное изображение термов, такое, например, как на рис. 2.7. Процесс сопоставления начинается от корня (главных функторов). Поскольку оба функтора сопоставимы, процесс продолжается и сопоставляет соответствующие пары аргументов. Таким образом, можно представить себе, что весь процесс сопоставления состоит из следующей последовательности (более простых) операций сопоставления:

```
треугольник = треугольник,
точка(1,1) = X,
A = точка(4,Y),
точка(2,3) = точка(2,Z).
```

Весь процесс сопоставления успешен, поскольку все сопоставления в этой последовательности успешны. Результирующая конкретизация такова:

```
X = точка(1,1)
A = точка(4,Y)
Z = 3
```

В приведенном ниже примере показано, как сопоставление само по себе можно использовать для содержательных вычислений. Давайте вернемся к простым геометрическим объектам с рис. 2.4 и напишем фрагмент программы для распознавания горизонтальных и вертикальных отрезков. «Вертикальность» – это свойство отрезка, поэтому его можно формализовать в Прологе в виде унарного отношения. Рис. 2.8 помогает сформулировать это отношение. Отрезок

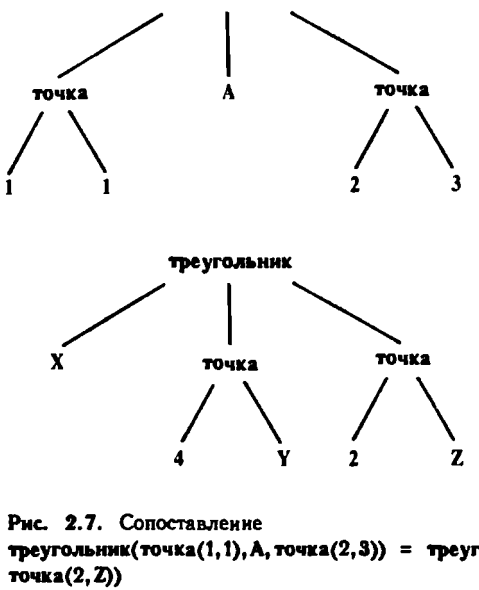


Рис. 2.7. Сопоставление
треугольник(точка(1,1),A,точка(2,3)) = треугольник(X,точка(4,Y), точка(2,Z))

является вертикальным, если x-координаты его точек-концов совпадают; никаких других ограничений на отрезок не накладывается. Свойство «горизонтальности» формулируется аналогично, нужно только в этой формулировке x и y поменять местами. Следующая программа, содержащая два факта, реализует эти формулировки:

```
верт( отр( точка(X,Y), точка(X,Y1) ) ).
гор( отр( точка(X,Y), точка(X1,Y) ) ).
```

С этой программой возможен такой диалог:

```
?- верт( отр( точка(1,1), точка(1,2) ) ).
да

?- верт( отр( точка(1,1), точка(2,Y) ) ).
нет

?- гор( отр( точка(1,1), точка(2,Y) ) ).
Y = 1
```

На первый вопрос система ответила «да», потому что цель, поставленная в вопросе, сопоставима с одним из фактов программы. Для второго вопроса сопоставимых фактов не нашлось. Во время ответа на третий вопрос при сопоставлении с фактом о горизонтальных отрезках Y получил значение 1.

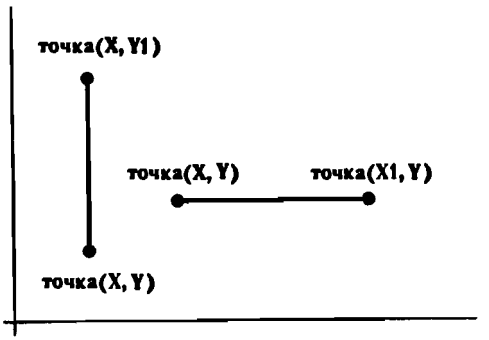


Рис. 2.8. Пример вертикальных и горизонтальных отрезков прямых.

Сформулируем более общий вопрос к программе: «Существуют ли какие-либо вертикальные отрезки, начало которых лежит в точке (2,3)?»

```
?- верт( отр( точка(2,3), P) ).
P = точка(2,Y)
```

Такой ответ означает: «Да, это любой отрезок, с концом в точке (2,Y), т. е. в произвольной точке вертикальной прямой $x = 2$». Следует заметить, что ответ пролог-системы возможно будет выглядеть не так красиво, как только что описано, а (в зависимости от реализации) приблизительно следующим образом:

```
P = точка(2,_136)
```

Впрочем, разница здесь чисто внешняя. В данном случае _136 это неинициализированная переменная. Имя _136 – законное имя прологовской переменной, которое система построила сама во время вычислений. Ей приходится генерировать новые имена, для того чтобы переименовывать введенные пользователем переменные в программе. Это необходимо по двум

причинам: первая – одинаковые имена обозначают в разных предложениях разные переменные; и вторая – при последовательном применении одного и того же предложения используется каждый раз его «копия» с новым набором переменных.

Другим содержательным вопросом к нашей программе является следующий: «Существует ли отрезок, который одновременно и горизонтален и вертикален?»

```
?- верт( S), гор( S).
S = отр( точка(X,Y), точка(X,Y) )
```

Такой ответ пролог-системы следует понимать так: «да, любой отрезок, выродившийся в точку, обладает как свойством вертикальности, так и свойством горизонтальности одновременно». Этот ответ снова получен лишь из сопоставления. Как и раньше, в ответе вместо X и Y могут появиться некоторые имена, сгенерированные системой.

Упражнения

2.3. Будут ли следующие операции сопоставления успешными или неуспешными? Если они будут успешными, то какова будет результирующая конкретизация переменных?

(a) **точка(A, B) = точка(1, 2)**
(b) **точка(A, B) = точка(X, Y, Z)**
(c) **плюс(2, 2) = 4**
(d) **+(2, D) = +(E, 2)**
(e) **треугольник(точка(–1, 0),P2,P3) = треугольник(P1,точка(1,0),точка(0,Y))**

Результирующая конкретизация определяет семейство треугольников. Как бы Вы описали это семейство?

2.4. Используя представление отрезков, применявшееся в данном разделе, напишите терм, соответствующий любому отрезку на вертикальной прямой $x = 5$.

2.5. Предположим, что прямоугольник представлен термом **прямоугольник(P1, P2, P3, P4)**, где **P** – вершины прямоугольника, положительно упорядоченные. Определите отношение

регулярный(R)

которое имеет место, если R – прямоугольник с вертикальными и горизонтальными сторонами.

2.3. Декларативный смысл пролог-программ

В главе 1 мы уже видели, что пролог-программу можно понимать по-разному: с декларативной и процедурной точек зрения. В этом и следующем разделах мы рассмотрим более формальное определение декларативного и процедурного смыслов программ базисного Пролога. Но сначала давайте еще раз взглянем на различия между этими двумя семантиками.

Рассмотрим предложение

P :- Q, R.

где P, Q и R имеют синтаксис термов. Приведем некоторые способы декларативной интерпретации этого предложения:

P – истинно, если Q и R истинны.
Из Q и R следует P.

А вот два варианта его «процедурного» прочтения:

Чтобы решить задачу P, *сначала* реши подзадачу Q, а *затем* – подзадачу R.
Чтобы достичь P, *сначала* достигни Q, а *затем* R.

Таким образом, различие между «декларативным» и «процедурным» прочтениями заключается в том, что последнее определяет не только логические связи между головой предложения и целями в его теле, но еще и *порядок,* в котором эти цели обрабатываются.

Формализуем теперь декларативный смысл.

Декларативный смысл программы определяет, является ли данная цель истинной (достижимой) и, если да, при каких значениях переменных она достигается. Для точного определения декларативного смысла нам потребуется понятие *конкретизации* предложения. Конкретизацией предложения C называется результат подстановки в него на место каждой переменной не-

которого терма. *Вариантом* предложения С называется такая конкретизация С, при которой каждая переменная заменена на другую переменную. Например, рассмотрим предложение:

```
имеетребенка( X) :- родитель( X, Y).
```

Приведем два варианта этого предложения:

```
имеетребенка( A)  :- родитель( A, B).
имеетребенка( X1) :- родитель( X1, X2).
```

Примеры конкретизаций этого же предложения:

```
имеетребенка( питер) :- родитель( питер, Z).
имеетребенка( барри) :-  родитель( барри,
                         маленькая(каролина) ).
```

Пусть дана некоторая программа и цель G, тогда, в соответствии с декларативной семантикой, можно утверждать, что

Цель G истинна (т.е. достижима или логически следует из программы) тогда и только тогда, когда

(1) в программе существует предложение С, такое, что

(2) существует такая его (С) конкретизация I, что

(a) голова I совпадает с G и
(b) все цели в теле I истинны.

Это определение можно распространить на вопросы следующим образом. В общем случае вопрос к пролог-системе представляет собой *список* целей, разделенных запятыми. Список целей называется *истинным* (достижимым), если *все* цели в этом списке истинны (достижимы) при *одинаковых* конкретизациях переменных. Значения переменных получаются из наиболее общей конкретизации.

Таким образом, запятая между целями обозначает *конъюнкцию* целей: они *все* должны быть истинными. Однако в Прологе возможна и *дизъюнкция* целей: должна быть истинной *по крайней мере одна* из целей. Дизъюнкция обозначается точкой с запятой.

Например:

```
P :- Q; R.
```

читается так: P – истинно, если истинно Q *или* истинно R. То есть смысл такого предложения тот же, что и смысл следующей пары предложений:

```
P :- Q.
P :- R.
```

Запятая связывает (цели) сильнее, чем точка с запятой. Таким образом, предложение

```
P :- Q, R; S, T, U.
```

понимается как:

```
P :-( Q, R); (S, T, U).
```

и имеет тот же смысл, что и два предложения

```
P :- Q, R.
P :- S, T, U.
```

Упражнения

2.6. Рассмотрим следующую программу:

```
f( 1, один).
f( s(1), два).
f(   s(s(1)),   три).
f( s(s(s(X))), N) :-
        f(X, N).
```

Как пролог-система ответит на следующие вопросы? Там, где возможны несколько ответов, приведите по крайней мере два.

```
(a)   ?- f( s(1), A).
(b)   ?- f(s(s(1)),два).
(c)   ?- f( s(s(s(s(s(s(1)))))), C).
(d)   ?- f(D, три).
```

2.7. В следующей программе говорится, что два человека являются родственниками, если

(a) один является предком другого, или
(b) у них есть общий предок, или
(c) у них есть общий потомок.

```
родственники( X, Y) :-
    предок( X, Y).

родственники( X, Y) :-
    предок( Y, X).

родственники( X, Y) :-
            % X и Y имеют общего предка
    предок( Z, X),
    предок( Z, Y).

родственники( X, Y) :-
            % X и Y имеют общего потомка
    предок( X, Z),
    предок( Y, Z).
```

Сможете ли вы сократить эту программу, используя запись с точкамн с запятой?

2.8. Перепишите следующую программу, не пользуясь точками с запятой.

```
преобразовать( Число, Слово) :-
    Число = 1, Слово = один;
    Число = 2, Слово = два;
    Число = 3, Слово = три.
```

2.4. Процедурная семантика

Процедурная семантика определяет, *как* пролог-система отвечает на вопросы. Ответить на вопрос – это значит удовлетворить список целей. Этого можно добиться, приписав встречающимся переменным значения таким образом, чтобы цели логически следовали из программы. Можно сказать, что процедурная семантика Пролога – это процедура вычисления списка целей с учетом заданной программы. «Вычислить цели» это значит попытаться достичь их.

Назовем эту процедуру вычислить. Как показано на рис. 2.9, входом и выходом этой процедуры являются:

входом – программа и список целей,
выходом – признак успех/неуспех и подстановка переменных.

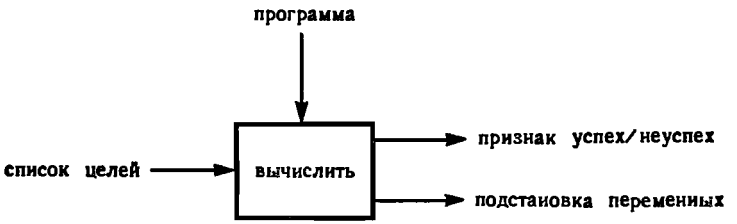


Рис. 2.9. Входы и выходы процедуры вычисления списка целей.

Смысл двух составляющих выхода такой:

(1) Признак успех/неуспех принимает значение «да», если цели достижимы, и «нет» – в противном случае. Будем говорить, что «да» сигнализирует об *успешном* завершении и «нет» – о *неуспехе*.

(2) Подстановка переменных порождается только в случае успешного завершения; в случае неуспеха подстановка отсутствует.

ПРОГРАММА

```
большой( медведь).          % Предложение 1
большой( слон).             % Предложение 2
маленький( кот).            % Предложение 3
коричневый ( медведь).      % Предложение 4
черный ( кот).              % Предложение 5
серый( слон).               % Предложение 6

темный( Z) :-               % Предложение 7:
      черный( Z).           % любой черный
                            % объект является темным
темный( Z) :-               % Предложение 8:
      коричневый( Z).       % Любой коричневый
                            % объект является темным
```

ВОПРОС

```
?- темный( X), большой( X)  % Кто одновременно темный
                            % и большой?
```

ШАГИ ВЫЧИСЛЕНИЯ

(1) Исходный список целевых утверждений:

темный(X), большой(X).

(2) Просмотр всей программы от начала к концу и поиск предложения, у которого голова сопоставима с первым целевым утверждением

темный(X).

Найдена формула 7:

темный(Z) :- черный(Z).

Замена первого целевого утверждения конкретизированным телом предложения 7 – порождение нового списка целевых утверждений.

черный(X), большой(X)

(3) Просмотр программы для нахождения предложения, сопоставимого с **черный(X)**. Найдено предложение 5: **черный (кот)**. У этого предложения нет тела, поэтому список целей при соответствующей конкретизации сокращается до

большой(кот)

(4) Просмотр программы в поисках цели **большой(кот)**. Ни одно предложение не найдено. Поэтому происходит возврат к шагу (3) и отмена конкретизации **X = кот**. Список целей теперь снова

черный(X), большой(X)

Продолжение просмотра программы ниже предложения 5. Ни одно предложение не найдено. Поэтому возврат к шагу (2) и продолжение просмотра ниже предложения 7. Найдено предложение (8):

темный(Z) :- коричневый(Z).

Замена первой цели в списке на **коричневый(X)**, что дает

коричневый(X), большой(X)

(5) Просмотр программы для обнаружения предложения, сопоставимого **коричневый(X)**. Найдено предложение **коричневый(медведь)**. У этого

предложения нет тела, поэтому список целей уменьшается до

большой(медведь)

(6) Просмотр программы и обнаружение предложения **большой(медведь)**. У него нет тела, поэтому список целей становится пустым. Это указывает на успешное завершение, а соответствующая конкретизация переменных такова:

Рис. 2.10. Пример, иллюстрирующий процедурную семантику Пролога: шаги вычислений, выполняемых процедурой **вычислить**.

В главе 1, в разд. «Как пролог-система отвечает на вопросы» мы уже фактически рассмотрели, что делает процедура **вычислить**. В оставшейся части данного раздела приводится несколько более формальное и систематическое описание этого процесса, которое можно пропустить без серьезного ущерба для понимания остального материала книги.

Конкретные операции, выполняемые в процессе вычисления целевых утверждений, показаны на рис. 2.10. Возможно, следует изучить этот рисунок прежде, чем знакомиться с последующим общим описанием.

Чтобы вычислить список целевых утверждений

G1, G2, ..., Gm

процедура **вычислить** делает следующее:

- **Если список целей пуст – завершает работу *успешно*.**
- **Если список целей не пуст, продолжает работу, выполняя (описанную далее) операцию 'ПРОСМОТР'.**
- ***ПРОСМОТР*: Просматривает предложения программы от начала к концу до обнаружения первого предложения С, такого, что голова С сопоставима с первой целью G1. Если такого предложения обнаружить не удается, то работа заканчивается *неуспехом*.**

Если C найдено и имеет вид

H :- B1, ..., Bn.

то переменные в C переименовываются, чтобы получить такой вариант C′ предложения C, в котором нет общих переменных со списком G1, ..., Gm. Пусть C′ – это

H′ :- B1′, ..., Bn′.

Сопоставляется G1 с H′; пусть S – результирующая конкретизация переменных. В списке целей G1, G2, ..., Gm, цель G1 заменяется на список B1′, ..., Bn′, что порождает новый список целей:

B1′, ..., Bn′,G2, ..., Gm

(Заметим, что, если C – факт, тогда n=0, и в этом случае новый список целей оказывается короче, нежели исходный; такое уменьшение списка целей может в определенных случаях превратить его в пустой, а следовательно, – привести к успешному завершению.)

Переменные в новом списке целей заменяются новыми значениями, как это предписывает конкретизация S, что порождает еще один список целей

B1″, ..., Bn″,G2′,..., Gm′

- **Вычисляет (используя рекурсивно ту же самую процедуру) этот новый список целей. Если его вычисление завершается успешно, то и вычисление исходного списка целей тоже завершается успешно. Если же его вычисление порождает неуспех, тогда новый список целей отбрасывается и происходит возврат к просмотру программы. Этот просмотр продолжается, начиная с предложения, непосредственно следующего за предложением C (C – предложение, использовавшееся последним) и делается попытка достичь успешного завершения с помощью другого предложения.**

Более компактная запись этой процедуры в обозначениях, близких к Паскалю, приведена на рис. 2.11.

Здесь следует сделать несколько дополнительных замечаний, касающихся процедуры **вычислить** в том

виде, в котором она приводится. Во-первых, в ней явно не указано, как порождается окончательная результирующая конкретизация переменных. Речь идет о конкретизации S, которая приводит к успешному завершению и которая, возможно, уточнялась последующими конкретизациями во время вложенных рекурсивных вызовов **вычислить**.

---

procedure ***вычислить (Прогр, СписокЦелей, Успех)***

Входные параметры:

Прогр: список предложений
СписокЦелей: список целей

Выходной параметр:

Успех: истинностное значение; ***Успех*** принимает значение истина, если список целевых утверждений (их конъюнкция) истинен с точки зрения ***Прогр***

Локальные переменные:

Цель: цель
ДругиеЦели: список целей
Достигнуты: истинностное значение
Сопоставились: истинностное значение
Конкрет: конкретизация переменных
H, H', $B1$, $B1'$, ..., B_n, B_n': цели

Вспомогательные функции:

пустой(L): возвращает истину, если L – пустой список

голова(L): возвращает первый элемент списка L

хвост(L): возвращает остальную часть списка L

конкат(L1,L2): создает конкатенацию списков – присоединяет список L2 к концу списка L1

сопоставление(T1, T2, Сопоставились, Конкрет): пытается сопоставить термы T1 и T2; если они сопоставимы, то ***Сопоставились*** – истина, а ***Конкрет*** представляет собой конкретизацию переменных

подставить(Конкрет,Цели): производит подстановку переменных в ***Цели*** согласно ***Конкрет***

```
begin
  if пустой( СписокЦелей) then Успех := истина
  else
    begin
      Цель := голова( СписокЦелей);
      ДругиеЦели := хвост( СписокЦелей);
      Достигнута := ложь;
      while not Достигнута and
        «в программе есть еще предложения» do
        begin
          Пусть следующее предложение в Прогр есть
              H :- B1, ..., Bn.
          Создать вариант этого предложения
              H' :- B1', ..., Bn'.
          сопоставление( Цель, H',
                                Сопоставились, Конкрет)
          if Сопоставились then
            begin
              НовыеЦели :=
                конкат([B1',...,Bn'],ДругиеЦели);
              НовыеЦели :=
                подставить(Конкрет,НовыеЦели);
              вычислить(Прогр,НовыеЦели,Достигнуты)
            end
          end;
      Успех := Достигнуты
    end
end;
```

Рис. 2.11. Вычисление целевых утверждений Пролога.

Всякий раз, как рекурсивный вызов процедуры **вычислить** приводит к неуспеху, процесс вычислений возвращается к **ПРОСМОТРУ** и продолжается с того предложения С, которое использовалось последним. Поскольку применение предложения С не привело к успешному завершению, пролог-система должна для продолжения вычислений попробовать альтернативное предложение. В действительности система аннулирует результаты части вычислений, приведших к неуспеху, и осуществляет возврат в ту точку (предложение С), в которой эта неуспешная ветвь начиналась. Когда процедура осуществляет возврат в некоторую точку, все конкретизации переменных, сделанные после этой точки, аннулируются. Такой порядок обеспечивает

систематическую проверку пролог-системой всех возможных альтернативных путей вычисления до тех пор, пока не будет найден путь, ведущий к успеху, или же до тех пор, пока не окажется, что все пути приводят к неуспеху.

Мы уже знаем, что даже после успешного завершения пользователь может заставить систему совершить возврат для поиска новых решений. В нашем описании процедуры вычислить эта деталь была опущена.

Конечно, в настоящих реализациях Пролога в процедуру вычислить добавлены и еще некоторые усовершенствования. Одно из них - сокращение работы по просмотрам программы с целью повышения эффективности. Поэтому на практике пролог-система не просматривает все предложения программы, а вместо этого рассматривает только те из них, которые касаются текущего целевого утверждения.

Упражнение

2.9. Рассмотрите программу на рис.2.10 и по типу того, как это сделано на рис. 2.10, проследите процесс вычисления пролог-системой вопроса

```
?- большой( X), темный( X).
```

Сравните свое описание шагов вычисления с описанием на рис. 2.10, где вычислялся, по существу, тот же вопрос, но с другой последовательностью целей:

```
?- темный( X), большой( X).
```

В каком из этих двух случаев системе приходится производить большую работу для нахождения ответа?

2.5. Пример: обезьяна и банан

Задача об обезьяне и банане часто используется в качестве простого примера задачи из области искусственного интеллекта. Наша пролог-программа,

способная ее решить, показывает, как механизмы сопоставления и автоматических возвратов могут применяться для подобных целей. Мы сначала составим программу, не принимая во внимание процедурную семантику, а затем детально изучим ее процедурное поведение. Программа будет компактной и наглядной.

Рассмотрим следующий вариант данной задачи. Возле двери комнаты стоит обезьяна. В середине этой комнаты к потолку подвешен банан. Обезьяна голодна и хочет съесть банан, однако она не может дотянуться до него, находясь на полу. Около окна этой же комнаты на полу лежит ящик, которым обезьяна может воспользоваться. Обезьяна может предпринимать следующие действия: ходить по полу, залезать на ящик, двигать ящик (если она уже находится около него) и схватить банан, если она стоит на ящике прямо под бананом. Может ли обезьяна добраться до банана?

Одна из важных проблем при программировании состоит в выборе (адекватного) представления решаемой задачи в терминах понятий используемого языка программирования. В нашем случае мы можем считать, что «обезьяний мир» всегда находится в некотором *состоянии*, и оно может изменяться со временем. Текущее состояние определяется взаиморасположением объектов. Например, исходное состояние мира определяется так:

(1) Обезьяна у двери.
(2) Обезьяна на полу.
(3) Ящик у окна.
(4) Обезьяна не имеет банана.

Удобно объединить все эти четыре информационных фрагмента в один структурный объект. Давайте в качестве такого объединяющего функтора выберем слово «состояние». На рис. 2.12 в виде структурного объекта изображено исходное состояние.

Нашу задачу можно рассматривать как игру для одного игрока. Давайте формализуем правила этой игры. Первое, целью игры является ситуация, в которой обезьяна имеет банан, т. е. любое состояние, у которого в качестве четвертой компоненты стоит «имеет»:

состояние(-, -, -, имеет)

Второе, каковы разрешенные ходы, переводящие мир

из одного состояния в другое? Существуют четыре типа ходов:

(1) схватнть банан,
(2) залезть иа ящик,
(3) подвинуть ящик,
(4) перейти в другое место.

Рис. 2.12. Исходное состояние обезьяньего мира, представленное в виде структурного объекта. Его четыре компоненты суть горизонтальная позиция обезьяны, вертикальная позиция обезьяны, позиция ящика, наличие или отсутствие у обезьяны банана.

Не всякий ход допустим при всех возможных состояниях мира. Например, ход «схватить» допустим, только если обезьяиа стоит на ящике прямо под баианом (т.е. в середиие комнаты) и еще не имеет баиана. Эти правила можио формализовать в Прологе в виде трехместного отношения ход:

ход(Состояние1, М, Состояние2)

Три аргумента этого отиошения определяют ход следующим образом:

Состояние1 ---------> Состояние2
М

Состояние1 это состояние до хода, **М** – выполняемый ход, и **Состояние2** – состояние после хода.

Ход «схватить», вместе с необходимыми ограничениямн иа состояние перед этим ходом, можио выразить такой формулой:

```
ход( состояние(середина, наящике, середина, неимеет),
                                  %Перед ходом
     схватить,                    %Ход
     состояние(середина, наящике, середина, имеет) ).
                                  %После хода
```

В этом факте говорится о том, что после хода у обезьяны уже есть банан и что она осталась на ящике в середине комнаты.

Таким же способом можно выразить и тот факт, что обезьяна, находясь на полу, может перейти из любой горизонтальной позиции P1 в любую позицию P2. Обезьяна может это сделать независимо от позиции ящика, а также независимо от того, есть у нее банан или нет. Все это можно записать в виде следующего прологовского факта:

```
ход( состояние(P1, наполу, B, H),
     перейти( P1, P2),        %Перейти из P1 в P2
     состояние( P2, наполу, B, H) ).
```

Заметим, что в этом предложении делается много утверждений и, в частности:

- выполненный ход состоял в том, чтобы «перейти из некоторой позиции P1 в некоторую позицию P2»;
- обезьяна находится на полу, как до, так и после хода;
- ящик находится в некоторой точке B, которая осталась неизменной после хода;
- состояние «имеет банан» остается неизменным после хода.

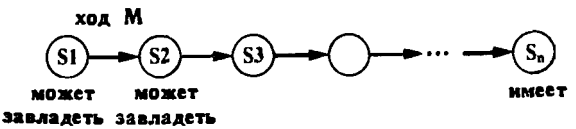


Рис. 2.13. Рекурсивная формулировка отношения **можетзавладеть**.

Данное предложение на самом деле определяет все множество возможных ходов указанного типа, так как оно применимо к любой ситуации, сопоставимой с состоянием, имеющим место перед входом. Поэтому такое предложение иногда называют *схемой* хода. Благодаря понятию переменной, имеющемуся в Прологе, такие схемы легко на нем запрограммировать.

Два других типа ходов: «подвинуть» и «залезть» — легко определить аналогичным способом.

Главный вопрос, на который должна ответить наша программа, это вопрос: «Может ли обезьяна, находясь в некотором начальном состоянии S, завладеть бананом?» Его можно сформулировать в виде предиката

можетзавладеть(S)

где аргумент S – состояние обезьяньего мира. Программа для **можетзавладеть** может основываться на двух наблюдениях:

(1) Для любого состояния S, в котором обезьяна уже имеет банан, предикат **можетзавладеть** должен, конечно, быть истинным; в этом случае никаких ходов не требуется. Вот соответствующий прологовский факт:

можетзавладеть(состояние(-, -, -, имеет)).

(2) В остальных случаях требуется один или более ходов. Обезьяна может завладеть бананом в любом состоянии S1, если для него существует ход из состояния P1 в некоторое состояние S2, такое, что, попав в него, обезьяна уже сможет завладеть бананом (за нуль или более ходов). Этот принцип показан на рис. 2.13. Прологовская формула, соответствующая этому правилу, такова:

```
можетзавладеть( S1) :-
    ход( S1, M, S2),
    можетзавладеть( S2).
```

Теперь мы полностью завершили нашу программу, показанную на рис. 2.14.

Формулировка **можетзавладеть** рекурсивна и совершенно аналогична формулировке отношения **предок** из гл. 1 (ср. рис. 2.13 и 1.7). Этот принцип используется в Прологе повсеместно.

Мы создали нашу программу «непроцедурным» способом. Давайте теперь изучим ее *процедурное* поведение, рассмотрев следующий вопрос к программе:

```
?- можетзавладеть(
     состояние(удвери, наполу, уокна, неммеет) ).
```

Ответом пролог-системы будет «да». Процесс, выполняемый ею при этом, обрабатывает, в соответствии с

процедурной семантикой Пролога, последовательность списков целей. Для этого требуется некоторый перебор ходов, для отыскания верного из нескольких альтернативных. В некоторых точках при таком переборе будет сделан неверный ход, ведущий в тупиковую ветвь процесса вычислений. На этом этапе автоматический возврат позволит исправить положение. На рис. 2.15 изображен процесс перебора.

```
% Разрешенные ходы

ход(состояние(середина, наящике, середина, неимеет),
    схватить,                    % Схватить банан
    состояние(середина, наящике, середина, имеет)).

ход( состояние( Р, наполу, Р, Н),
    залезть,                     % Залезть на ящик
    состояние( Р, наящике, Р, Н) ).

ход( состояние( Р1, наполу, Р1, Н),
    подвинуть( Р1, Р2),          % Подвинуть ящик с Р1 на Р2
    состояние( Р2, наполу, Р2, Н) ).

ход( состояние( Р1, наполу, В, Н),
    перейти( Р1, Р2),            % Перейти с Р1 на Р2
    состояние( Р2, наполу, В, Н) ).

% можетзавладеть(Состояние): обезьяна может завладеть
% бананом, находясь в состоянии Состояние

можетзавладеть( состояние( -, -, -, имеет) ).

% может 1: обезьяна уже его имеет

можетзавладеть( Состояние1) :-

% может 2: Сделать что-нибудь, чтобы завладеть им

    ход( Состояние1, Ход, Состояние2),

% сделать что-нибудь

    можетзавладеть( Состояние2).

% теперь может завладеть
```

Рис. 2.14. Программа для задачи об обезьяне и банане.

Для ответа на наш вопрос системе пришлось сделать лишь один возврат. Верная последовательность ходов была найдена почти сразу. Причина такой эффективности программы кроется в том порядке, в

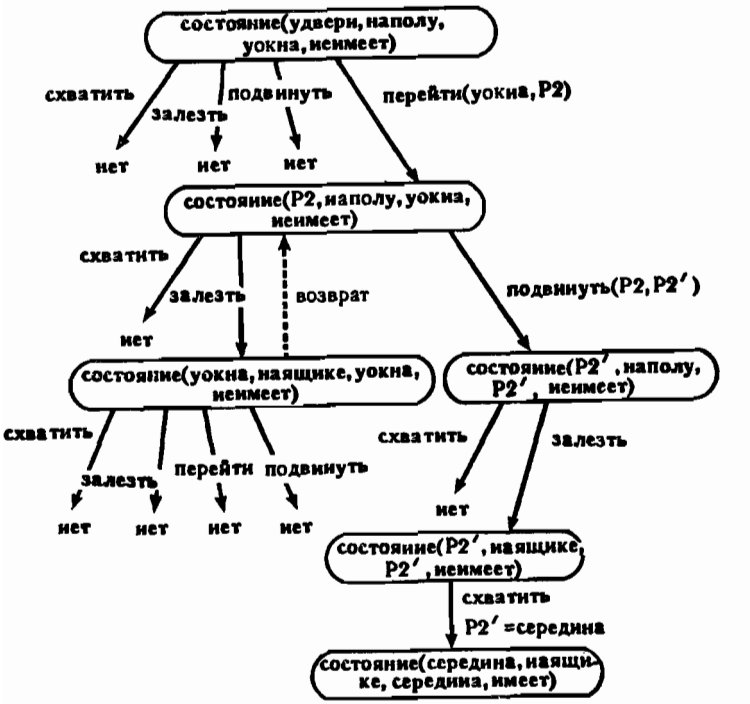


Рис. 2.15. Поиск банана обезьяной. Перебор начинается в верхнем узле и распространяется вниз, как показано. Альтернативные ходы перебираются слева направо. Возврат произошел только один раз.

котором в ней расположены предложения, касающиеся отношения **ход.** В нашем случае этот порядок (к счастью) оказался весьма подходящим. Однако возможен и менее удачный порядок. По правилам игры обезьяна могла бы с легкостью ходить туда-сюда, даже не касаясь ящика, или бесцельно двигать ящик в разные стороны. Как будет видно из следующего раздела, более тщательное исследование обнаруживает, что порядок предложений в нашей программе является, на самом деле, критическим моментом для успешного решения задачи.

2.6. Порядок предложений и целей

2.6.1. Опасность бесконечного цикла

Рассмотрим следующее предложение:

```
p :- p.
```

В нем говорится: «p истинно, если p истинно». С точки зрения декларативного смысла это совершенно корректно, однако в процедурном смысле оно бесполезно. Более того, для пролог-системы такое предложение может породить серьезную проблему. Рассмотрим вопрос:

```
?- p.
```

При использовании вышеприведенного предложения цель p будет заменена на ту же самую цель p; она в свою очередь будет заменена снова на p и т.д. В этом случае система войдет в бесконечный цикл, не замечая, что никакого продвижения в вычислениях не происходит.

Данный пример демонстрирует простой способ ввести пролог-систему в бесконечный цикл. Однако подобное зацикливание могло встретиться и в некоторых наших предыдущих программах, если бы мы изменили порядок предложений, или же порядок целей в них. Будет полезно рассмотреть несколько примеров.

В программе об обезьяне и банане предложения, касающиеся отношения **ход**, были упорядочены следующим образом: схватить, залезть, подвинуть, перейти (возможно, для полноты следует добавить еще «слезть»). В этих предложениях говорится, что можно схватить, можно залезть и т.д. В соответствии с процедурной семантикой Пролога порядок предложений указывает на то, что обезьяна предпочитает схватывание залезанию, залезание – передвиганию и т.д. Такой порядок предпочтений на самом деле помогает обезьяне решить задачу. Но что могло случиться, если бы этот порядок был другим? Предположим, что предложение с «перейти» оказалось бы первым. Про-

цесс вычнслення нашей нсходной целн из предыдущего раздела

```
?- можетзавладеть(
        состояние(удвери, наполу, уокна, неимеет) ).
```

протекал бы на этот раз так. Первые четыре списка целей (с соответствующим образом перенменованными переменными) остались бы такими же, как и раньше:

```
(1)   можетзавладеть(
      состояние(удвери, наполу, уокна, неимеет) ).
```

Применение второго предложения из **можетзавладеть** («может2») породило бы

```
(2)   ход(
      состояние(удвери, наполу, уокна, неимеет), M', S2'),
      можетзавладеть( S2' )
```

С помощью хода **перейти(уокна, P2')** получилось бы

```
(3)   можетзавладеть(
          состояние(P2', наполу, уокна, неимеет) )
```

Повторное использование предложеиня «может2» превратило бы список целей в

```
(4)   ход(
          состояние(P2', наполу, уокна, неимеет), M'', S2''),
          можетзавладеть( S2'')
```

С этого момента начались бы отличия. Первым предложением, голова которого сопоставима с первой целью из этого списка, было бы теперь «перейти» (а не «залезть», как раньше). Конкретизация стала бы следующей:

```
S2'' = состояние( P2'', наполу, уокна, неимеет).
```

Поэтому список целей стал бы таким:

```
(5)   можетзавладеть(
          состояние( P2'', наполу, уокна, неимеет) )
```

Применение предложения «может2» дало бы

```
(6)  ход(
       состояние(P2'', наполу, уокна, неимеет), M''', S2'''),
       можетзавладеть( S2''')
```

Снова первым было бы поиробовано «перейти» и получилось бы

(7) **можетзавладеть(состояние(P2″′, наполу, уокна, неимеет))**

Сравнним теперь цели (3), (5) и (7). Они похожи и отличаются лишь одиой переменной, которая по очереди имела имена P′, P″ и P″′. Как мы знаем, успешность цели не зависит от конкретиых имен переменных в ней. Это означает, что, начиная со списка целей (3), процесс вычислений никуда не продвинулся. Фактически мы замечаем, что по очереди многократно используются одии и те же два предложения: «может2» и «перейти». Обезьяна перемещается, даже не пытаясь воспользоваться ящиком. Поскольку продвижения иет, такая ситуация продолжалась бы (теоретически) бесконечно: пролог-система не сумела бы осознать, что работать в этом направлении нет смысла.

Данный пример показывает, как пролог-система может пытаться решить задачу таким способом, при котором решение никогда не будет достигиуто, хотя оно существует. Такая ситуация не является редкостью при программировании на Прологе. Да и при программировании на других языках бесконечные циклы не такая уж редкость. Что *действительно* необычно при сравнении Пролога с другими языками, так это то, что декларативиая семантика пролог-программы может быть правильной, но в то же самое время ее процедурная семантика может быть ошибочной в том смысле, что с помощью такой программы нельзя получить правильный ответ на вопрос. В таких случаях система не способна достичь цели потому, что она пытается добраться до ответа, но выбирает при этом неверный путь.

Теперь уместно спросить: «Не можем ли мы внести какое-либо более существенное изменение в нашу программу, так чтобы полностью исключить опасность зацикливания? Или же нам всегда придется рассчитывать на удачный порядок предложений и целей?» Как оказывается, программы, в особенности большие, были бы через чур ненадежиыми, если бы можно было рассчитывать лишь на некоторый удачиый порядок. Существует несколько других методов, позволяющих избежать зацикливаиия и являющихся более общими и надежными, чем сам по себе метод упорядочивания. Такие методы будут систематически использоваться дальше в книге, в особенности в тех главах, в ко-

торых пойдет речь о нахождении путей (в графах), о решении интеллектуальных задач и о переборе.

2.6.2. Варианты программы, полученные путем переупорядочивания предложений и целей

Уже в примерах программ гл. 1 существовала скрытая опасность зацикливания. Определение отношения **предок** в этой главе было таким:

```
предок( X, Z) :-
    родитель( X, Z).
предок( X, Z) :-
    родитель( X, Y),
    предок( Y, Z).
```

Проанализируем некоторые варианты этой программы. Ясно, что все варианты будут иметь одинаковую декларативную семантику, но разные процедурные семантики.

В соответствии с декларативной семантикой Пролога мы можем, не меняя декларативного смысла, изменить

(1) порядок предложений в программе и
(2) порядок целей в телах предложений.

Процедура **предок** состоит из двух предложений, и одно из них содержит в своем теле две цели. Возможны, поэтому, четыре варианта данной программы, все с одинаковым декларативным смыслом. Эти четыре варианта можно получить, если

(1) поменять местами оба предложения и
(2) поменять местами цели в каждом из этих двух последовательностей предложений.

Соответствующие процедуры, названные **пред1**, **пред2**, **пред3** и **пред4**, показаны на рис. 2.16.

Есть существенная разница в поведении этих четырех декларативно эквивалентных процедур. Чтобы это продемонстрировать, будем считать отношение **родитель** определенным так, как показано на рис. 1.1 гл. 1, и посмотрим, что произойдет, если мы спросим, является ли Том предком Пат, используя все четыре варианта отношения **предок**:

```
?- пред1( том, пат).
да
?- пред2( том, пат).
да
?- пред3( том, пат).
да
?- пред4( том, пат).
```

```
% Четыре версии программы предок
% Исходная версия
пред1( X, Z) :-
    родитель( X, Z).
пред1( X, Z) :-
    родитель( X, Y),
    пред1( Y, Z).
% Вариант a: изменение порядка предложений в исходной версии
пред2( X, Z) :-
    родитель( X, Y),
    пред2( Y, Z).
пред2( X, Z) :-
    родитель( X, Z).
% Вариант b: изменение порядка целей во втором предложении
% исходной версии
пред3( X, Z) :-
    родитель( X, Z).
пред3( X, Z) :-
    пред3( X, Y),
    родитель( Y, Z).
% Вариант c: изменение порядка предложений и целей в исходной
% версии
пред4( X, Z) :-
    пред4( X, Y),
    родитель( Y, Z).
пред4( X, Z) :-
    родитель( X, Z).
```

Рис. 2.16. Четыре версии программы **предок**.

В последнем случае пролог-система не сможет найти ответа. И выведет на терминал сообщение: «Не хватает памяти».

На рис. 1.11 гл. 1 были показаны все шагн вычислений по пред1 (в главе 1 она называлась предок), предпринятые для ответа на этот вопрос. На рис 2.17 показаны соответствующие вычисления по пред2, пред3 и пред4. На рис. 2.17 (с) ясно видно, что работа пред4 – бесперспективна, а рис. 2.17(а) показывает, что пред2 довольно неэффективна по сравнению с пред1: пред2 пронзводит значительно больший перебор и делает больше возвратов по фамнльному дереву.

Такое сравнение должно напомнить нам об общем практическом правиле при решении задач: обычно бывает полезным прежде всего попробовать самое простое соображение. В нашем случае все версии отношения предок основаны на двух соображениях:

- более простое – нужно проверить, не удовлетворяют ли два аргумента отношения предок отношению родитель;
- более сложное – найти кого-либо «между» этими двумя людьми (кого-либо, кто связан с ними отношениями родитель и предок).

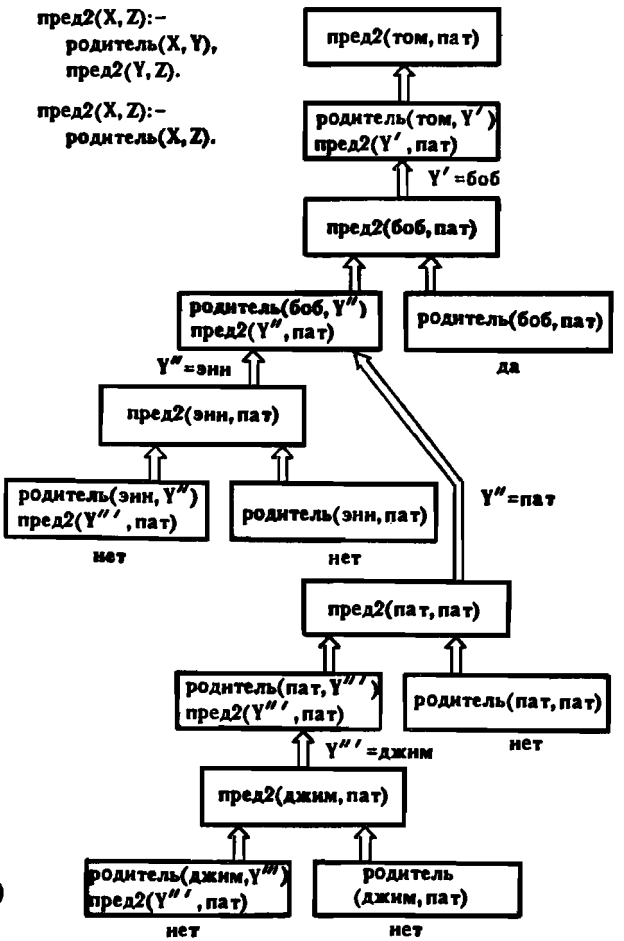
Из всех четырех вариантов отношения предок, пред1 использует наиболее простое соображение в первую очередь. В противоположность этому пред4 всегда сначала пробует использовать самое сложное. Пред2 и пред3 находятся между этими двумя крайностями. Даже без детального изучения процессов вычислений ясно, что пред1 следует предпочесть просто на основании правила «самое простое пробуй в первую очередь».

Наши четыре варианта процедуры предок можно далее сравнить, рассмотрев вопрос: «На какие типы вопросов может отвечать тот или иной конкретный вариант и на какие не может?» Оказывается, пред1 и пред2 оба способны найти ответ на любой вид вопроса относительно предков; пред4 никогда не находит ответа, а пред3 иногда может найти, иногда нет. Вот пример вопроса, на который пред4 ответить не может:

```
?- пред3( лиз, джим).
```

Такой вопрос тоже вводит систему в бесконечную

рекурсию. Следовательно, и **пред3** нельзя признать верным с точки зрения процедурного смысла.



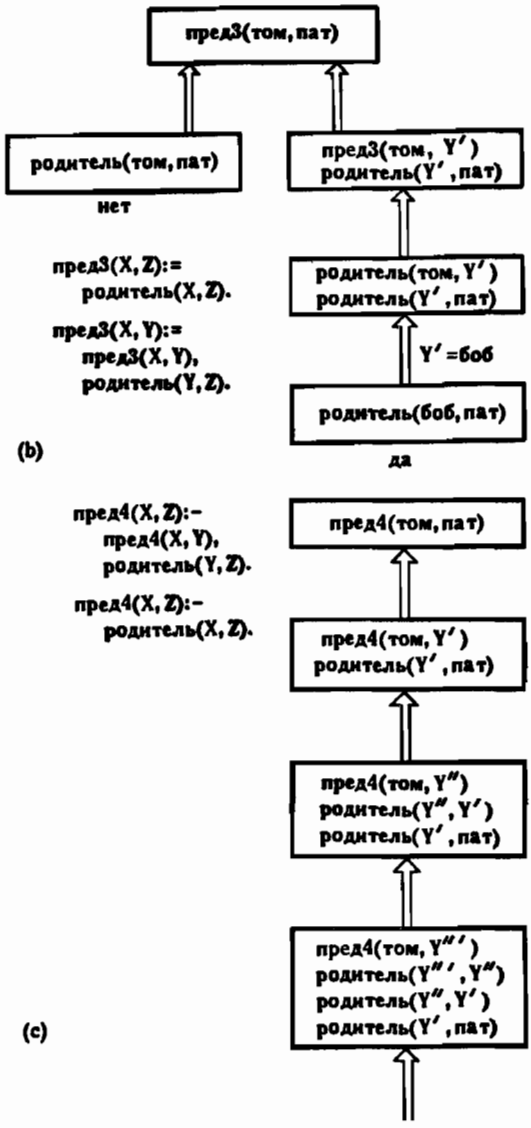


Рис. 2.17. Поведение трех вариантов формулировки отношения предок при ответе на вопрос, является ли Том предком Пат?

2.6.3. Сочетание декларативного и процедурного подходов

В предыдущем разделе было показано, что порядок целей и предложений имеет существенное значение. Более того, существуют программы, которые верны в декларативном смысле, но на практике не работают. Такое противоречие между декларативным и процедурным смыслами может вызвать недовольство. Кто-нибудь спросит: «А почему вообще не забыть о декларативном смысле?» Такое пожелание становится особенно сильным, когда рассматриваются предложения типа:

предок(X, Z) :- предок(X, Z).

Это предложение верно в декларативном смысле, но совершенно бесполезно в качестве рабочей программы.

Причина, по которой не следует забывать о декларативном смысле, кроется в том, что прогресс, достигнутый в технологии программирования, получен на пути продвижения от учета всех процедурных деталей к концентрации внимания на декларативных аспектах, которые обычно легче формулировать и понимать. Сама система, а не программист, должна нести бремя заботы о процедурных деталях. В этом Пролог оказывает некоторую помощь, хотя, как мы видели в данном разделе, помощь лишь частичную: иногда он правильно прорабатывает эти процедурные детали, иногда – нет. Многие придерживаются мнения, что лучше иметь хоть какую-то декларативную семантику, чем никакой (отсутствие декларативной семантики характерно для многих других языков программирования). Практическим следствием такого взгляда является тот факт, что часто довольно легко получить работающую программу, имея программу декларативно корректную. Поэтому практичным следует признать такой подход: сосредоточиться на декларативных аспектах задачи, затем пропустить на машине полученную программу и, если она окажется процедурно неправильной, попытаться изменить порядок следования предложений и целей.

2.7. Замечания о взаимосвязи между Прологом и логикой

Пролог восходит к математической логике, поэтому его синтаксис и семантику можно наиболее точно описать при помощи логики. Так часто и поступают при обучении этому языку. Однако такой подход к ознакомлению с Прологом предполагает знание читателем определенных понятий математической логики. С другой стороны, знание этих понятий явно необязательно для того, чтобы понять и использовать Пролог в качестве инструмента для практического программирования, а цель данной книги – научить именно этому. Для тех же читателей, которые особенно заинтересуются взаимосвязями между Прологом и логикой, мы сейчас перечислим основные из них, а также приведем некоторые подходящие источники.

Синтаксис Пролога – это синтаксис предложений *логики предикатов первого порядка*, записанных в так называемой *форме предложений* (форме, при которой кванторы не выписываются явно), а точнее, в виде частного случая таких предложений – в виде *формул Хорна* (предложений, имеющих самое большее один положительный литерал). Клоксин и Меллиш (1981 г.) приводят пролог-программу, которая преобразует предложения исчисления предикатов первого порядка в форму предложений. Процедурный смысл Пролога основывается на *принципе резолюций*, применяющемся для автоматического доказательства теорем, который был предложен Робинсоном в его классической статье (1965 г.). В Прологе используется особая стратегия доказательства теоремы методом резолюций, носящая название SLD. Введение в исчисление предикатов первого порядка и доказательство теорем, основанное на методе резолюций, можно найти у Нильсона (1981 г.). Математические вопросы, касающиеся свойств процедурной семантики Пролога в их связи с логикой, проанализированы Ллойдом (1984 г.).

Сопоставление в Прологе соответствует некоторому действию в логике, называемому *унификацией*. Мы, однако, избегаем слова «уинфикация», так как по

соображениям эффективиости в большинстве пролог-систем сопоставленне реализовано таким образом, что оно ие полностью соответствует унификацин (см. упражнеиие 2.10). Тем ие менее, с практической точки зрения, такая приближеииая уиификация вполие допустима.

Упражнение

2.10. Что будет, если пролог-системе задать такой вопрос:

```
?- X = f( X).
```

Успешиым нли иеуспешиым будет здесь сопоставление? По определеиию уиифнкацни в логике, сопоставление должно быть иеуспешиым, а что будет в соответствии с нашим определеннем сопоставления из раздела 2.2? Попробуйте объяснить, почему миогие реализации Пролога отвечают на вышеприведеиный вопрос так:

```
X = f(f(f(f(f(f(f(f(f(f(f(f(f( ...
```

Резюме

К иастоящему момеиту мы изучили иечто вроде базового Пролога, который называют еще «чистый Пролог». Он «чист», потому что довольно точно соответствует формальиой логике. Расширеиия, преследующие цель приспособить язык к иекоторым практическим нуждам, будут изучены дальше (гл. 3, 5, 6, 7). Важиыми моментами даиной главы являются следующие:

- Простые объекты в Прологе – это *атомы*, *переменные* и *числа*. Структурные объекты, или *структуры*, используются для представления объектов, которые состоят из нескольких компонент.
- Структуры строятся посредством *функторов*. Каждый функтор определяется своими имеием и арностью.

- Тип объекта распознается исключительно по его синтаксической форме.
- *Область известности (лексический диапазон)* переменных – одно предложение. Поэтому одно и то же имя в двух предложениях обозначает две разные переменные.
- Структуры могут быть естественным образом изображены в виде деревьев. Пролог можно рассматривать как язык обработки деревьев.
- Операция *сопоставление* берет два терма и пытается сделать их идентичными, подбирая соответствующую конкретизацию переменных в обоих термах.
- Сопоставление, если оно завершается успешно, в качестве результата выдает *наиболее общую* конкретизацию переменных.
- *Декларативная семантика* Пролога определяет, является ли целевое утверждение истинным, исходя из данной программы, и если оно истинно, то для какой конкретизации переменных.
- Запятая между целями означает их конъюнкцию. Точка с запятой между целями означает их дизъюнкцию.
- *Процедурная семантика* Пролога – это процедура достижения списка целей в контексте данной программы. Процедура выдает истинность или ложность списка целей и соответствующую конкретизацию переменных. Процедура осуществляет автоматический возврат для перебора различных вариантов.
- Декларативный смысл программ на «чистом Прологе» не зависит от порядка предложений и от порядка целей в предложениях.
- Процедурный смысл существенно зависит от порядка целей и предложений. Поэтому порядок может повлиять на эффективность программы; неудачный порядок может даже привести к бесконечным рекурсивным вызовам.
- Имея декларативно правильную программу, можно улучшить ее эффективность путем изменения

порядка предложений и целей при сохранении ее декларативной правильности. Переупорядочивание – один из методов предотвращения зацикливания.

- Кроме переупорядочивания существуют и другие, более общие методы предотвращения зацикливания, способствующие получению процедурно правильных программ.
- В данной главе обсуждались следующие понятия:

объекты данных:
атом, число, переменная, структура
терм
функтор, арность функтора
главный функтор терма
сопоставление термов
наиболее общая конкретизация
декларативная семантика
конкретизация предложений,
вариант предложения
процедурная семантика
вычисление целевого утверждения

Литература

Clocksin W. F. and Mellish C. S. (1981). *Programming in Prolog*. Springer-Verlag. [Имеется перевод: Клоксин У., Меллиш К. Программирование на языке Пролог. – М.: Мир, 1987.]

Lloyd J. W. (1984). *Foundations of Logic Programming*. Springer-Verlag.

Nilsson N. J. (1981). *Principles of Artificial Intelligence*. Tioga; Springer-Verlag.

Robinson A.J. (1965). A machine-oriented logic based on the resolution principle. *JACM* 12: 23–41. [Имеется перевод: Робинсон Дж. Машинно-ориентированная логика, основанная на принципе резолюции.– В кн. Кибернетический сборник, вып. 7, 1970, с. 194–218.]

3 СПИСКИ, ОПЕРАТОРЫ, АРИФМЕТИКА

В этой главе мы будем изучать специальные способы представления списков. Список – один из самых простых и полезных типов структур. Мы рассмотрим также некоторые программы для выполнения типовых операций над списками и, кроме того, покажем, как можно просто записывать арифметические выражения и операторы, что во многих случаях позволит улучшить «читабельность» программ. Базовый Пролог (глава 2), расширенный этими тремя добавлениями, станет удобной основой для составления интересных программ.

3.1. Представление списков

Список – это простая структура данных, широко используемая в нечисловом программировании. Список – это последовательность, составленная из произвольного числа элементов, например **энн, теннис, том, лыжи**. На Прологе это записывается так:

```
[ энн, теннис, том, лыжи ]
```

Однако таково лишь внешнее представление списков. Как мы уже видели в гл. 2, все структурные объекты Пролога – это деревья. Списки не являются исключением из этого правила.

Каким образом можно представить список в виде стандартного прологовского объекта? Мы должны рассмотреть два случая: пустой список и не пустой список. В первом случае список записывается как

атом []. Во втором случае список следует рассматривать как структуру состоящую из двух частей:

(1) первый элемент, называемый *головой* списка;

(2) остальная часть списка, называемая *хвостом*.

Например, для списка

[энн, теннис, том, лыжи]

энн – это голова, а хвостом является список

[теннис, том, лыжи]

В общем случае, головой может быть что угодно (любой прологовский объект, например, дерево или переменная); хвост же должен быть списком. Голова соединяется с хвостом при помощи специального функтора. Выбор этого функтора зависит от конкретной реализации Пролога; мы будем считать, что это точка:

.(Голова, Хвост)

Поскольку **Хвост** – это список, он либо пуст, либо имеет свои собственную голову и хвост. Таким образом, выбранного способа представления списков достаточно для представления списков любой длины. Наш список представляется следующим образом:

.(энн, .(теннис, .(том, .(лыжи, [])))))

На рис. 3.1 изображена соответствующая древовидная структура. Заметим, что показанный выше пример содержит пустой список []. Дело в том, что самый последний хвост является одноэлементным списком:

[лыжи]

Хвост этого списка пуст

[лыжи] = .(лыжи, [])

Рассмотренный пример показывает, как общий принцип структуризации объектов данных можно применить к спискам любой длины. Из нашего примера также видно, что такой примитивный способ представления в случае большой глубины вложенности подэлементов в хвостовой части списка может привести к довольно запутанным выражениям. Вот почему в Прологе предусматривается более лаконичный способ изображения

списков, при котором они записываются как последовательности элементов, заключенные в квадратные скобки. Программист может использовать оба способа, но представление с квадратными скобками, конечно, в большинстве случаев пользуется предпочтением. Мы, однако, всегда будем помнить, что это всего лишь косметическое улучшение и что во внутреннем представлении наши списки выглядят как деревья. При выводе же они автоматически преобразуются в более лаконичную форму представления. Так, например, возможен следующий диалог:

```
?- Список1 = [a,b,c],
   Список2 = (a, .(b, .(c,[]) ) ).

Список1 = [a,b,c]
Список2 = [a,b,c]

?- Увлечения1 = .( теннис, .(музыка, [] ) ),
   Увлечения2 = [лыжи, еда],
   L = [энн, Увлечения1, том, Увлечения2].

Увлечения1 = [теннис, музыка]
Увлечения2 = [лыжи, еда]
L = [энн,[теннис,музыка],том,[лыжи,еда]]
```

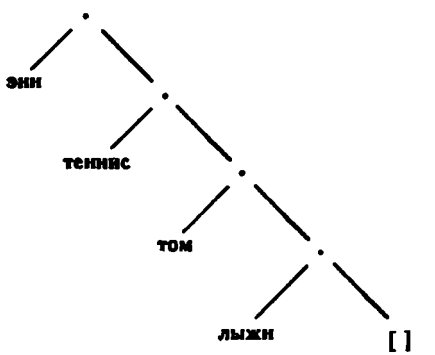


Рис. 3.1. Представление списка [энн, теннис, том, лыжи] в виде дерева.

Приведенный пример также напоминает нам о том, что элементами списка могут быть любые объекты, в частности тоже списки.

На практике часто бывает удобным трактовать хвост списка как самостоятельный объект. Например, пусть

L = [a,b,c]

Тогда можно написать:

Хвост = [b,c] и L = .(a,Хвост)

Для того, чтобы выразить это при помощи квадратных скобок, в Прологе предусмотрено еще одно расширение нотации для представления списка, а именно вертикальная черта, отделяющая голову от хвоста:

L = [a | Хвост]

На самом деле вертикальная черта имеет более общий смысл: мы можем перечислить любое количество элементов списка, затем поставить символ «|», а после этого – список остальных элементов. Так, только что рассмотренный пример можно представить следующими различными способами:

[a,b,c] = [a|[b,c]] = [a,b|[c]] = [a,b,c|[]]

Подытожим:

- Список – это структура данных, которая либо пуста, либо состоит из двух частей: *головы* и *хвоста*. Хвост в свою очередь сам является списком.
- Список рассматривается в Прологе как специальный частный случай двоичного дерева. Для повышения наглядности программ в Прологе предусматриваются специальные средства для списковой нотации, позволяющие представлять списки в виде

  [Элемент1, Элемент2, ...]

  или

  [Голова | Хвост]

  или

  [Элемент1, Элемент2, ... | Остальные]

3.2. Некоторые операции над списками

Списки можно применять для представления множеств, хотя и существует некоторое различие между этими понятиями: порядок элементов множества не существенен, в то время как для списка этот порядок имеет значение; кроме того, один и тот же объект может встретиться в списке несколько раз. Однако наиболее часто используемые операции над списками аналогичны операциям над множествами. Среди них

- проверка, является ли некоторый объект элементом списка, что соответствует проверке объекта на принадлежность множеству;
- конкатенация (сцепление) двух списков, что соответствует объединению множеств;
- добавление нового объекта в список или удаление некоторого объекта из него.

В оставшейся части раздела мы покажем программы, реализующие эти и некоторые другие операции над списками.

3.2.1. Принадлежность к списку

Мы представим отношение принадлежности как

принадлежит(X, L)

где X – объект, а L – список. Цель **принадлежит(X, L)** истинна, если элемент X встречается в L. Например, верно что

принадлежит(b, [a,b,c])

и, наоборот, не верно, что

принадлежит(b, [a,[b,c]])

но

принадлежит([b,c], [a,[b,c]])

истинно. Составление программы для отношения принадлежности может быть основано на следующих соображениях:

(1) X есть голова L, либо
(2) X принадлежит хвосту L.

Это можно записать в виде двух предложений, первое из которых есть простой факт, а второе – правило:

```
принадлежит( X, [X |Хвост ] ).

принадлежит( X, [Голова |Хвост ] ) :-
    принадлежит( X, Хвост).
```

3.2.2. Сцепление (конкатенация)

Для сцепления списков мы определим отношение

```
конк( L1, L2, L3)
```

Здесь L1 и L2 – два списка, а L3 – список, получаемый при их сцеплении. Например,

```
конк( [a,b], [c,d], [a,b,c,d] )
```

истинно, а

```
конк( [a,b], [c,d], [a,b,a,c,d] )
```

ложно. Определение отношения **конк**, как и раньше, содержит два случая в зависимости от вида первого аргумента L1:

(1) Если первый аргумент пуст, тогда второй и третий аргументы представляют собой один и тот же список (назовем его L), что выражается в виде следующего прологовского факта:

```
конк( [], L, L ).
```

(2) Если первый аргумент отношения **конк** не пуст, то он имеет голову и хвост и выглядит так:

```
[X | L1]
```

На рис. 3.2 показано, как производится сцепление списка [X | L1] с произвольным списком L2. Результат сцепления – список [X | L3], где L3 получен после сцепления списков L1 и

L2. На Прологе это можно записать следующим образом:

```
конк( [X | L1], L2, [X | L3]) :-
    конк( L1, L2, L3).
```

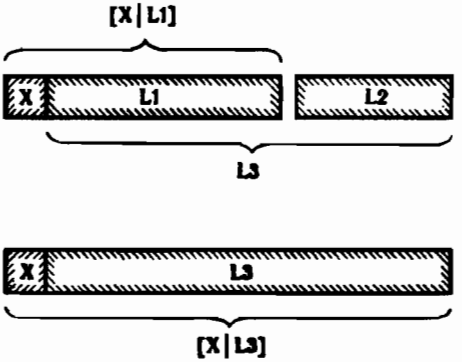


Рис. 3.2. Конкатенация списков.

Составленную программу можно теперь использовать для сцепления заданных списков, например:

```
?- конк( [a,b,c], [1,2,3], L ).
L = [a,b,c,1,2,3]

?- конк( [a,[b,c],d], [a,[],b], L ).
L = [a, [b,c], d, a, [], b]
```

Хотя программа для конк выглядит довольно просто, она обладает большой гибкостью и ее можно использовать многими другими способами. Например, ее можно применить как бы в обратном направлении для *разбиения* заданного списка на две части:

```
?- конк( L1, L2, [a,b,c] ).

L1 = []
L2 = [a,b,c];

L1 = [a]
L2 = [b,c];

L1 = [a,b]
L2 = [c];
```

```
L1 = [a,b,c]
L2 = [];

no                (нет)
```

Список [a,b,c] разбивается на два списка четырьмя способами, и все они были обнаружены нашей программой при помощи механизма автоматического перебора.

Нашу программу можно также прнменнть для поиска в списке комбннации элементов, отвечающей некоторому условию, задаваемому в виде шаблона илн образца. Например, можно найти все месяцы, предшествующие данному, и все месяцы, следующне за ним, сформулировав такую цель:

```
?- конк( До, [май | После ],
         [янв,фев,март,апр,май,июнь,
             июль,авг,сент,окт,ноябрь,дек]).

   До = [янв,фев,март,апр]

   После = [июнь,июль,авг,сент,окт,ноябрь,дек].
```

Далее мы сможем найти месяц, непосредственно предшествующий маю, и месяц, иепосредственно следующий за ним, задав вопрос:

```
?- конк( _, [Месяц1,май, Месяц2 | _],
         [янв,фев,март,апр,май,июнь,
             июль,авг,сент,окт,ноябрь,дек]).

   Месяц1 = апр
   Месяц2 = июнь
```

Более того, мы сможем, например, удалить из некоторого списка L1 все, что следует за тремя последовательными вхождениями элемента z в L1 вместе с этими тремя z. Например, это можно сделать так:

```
?- L1 = [a,b,z,z,c,z,z,z,d,e],
   конк( L2, [z,z,z | _], L1).

L1 = [a,b,z,z,c,z,z,z,d,e]
L2 = [a,b,z,z,c]
```

Мы уже запрограммировалн отношение принадлежности. Однако, используя конк, можно было бы определить

это отношение следующим эквивалентным способом:

```
принадлежит1( X, L) :-
    конк( L1, [X | L2], L).
```

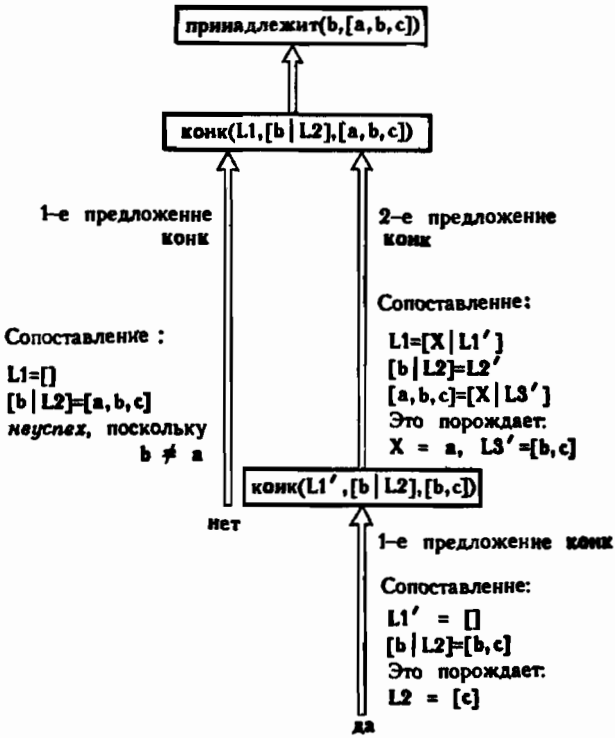


Рис. 3.3. Процедура принадлежит1 находит элемент в заданном списке, производя по нему последовательный поиск.

В этом предложении сказано: «X принадлежит L, если список L можно разбить на два списка таким образом, чтобы элемент X являлся головой второго из них. Разумеется, принадлежит1 определяет то же самое отношение, что и принадлежит. Мы использовал

другое имя только для того, чтобы различать таким образом две разные реализации этого отношения. Заметим, что, используя анонимную переменную, можно записать вышеприведенное предложение так:

```
принадлежит1( X, L) :-
    конк( _, [X | _], L).
```

Интересно сравнить между собой эти две реализации отношения принадлежности. **Принадлежит** имеет довольно очевидный процедурный смысл:

Для проверки, является ли X элементом списка L, нужно

(1) сначала проверить, не совпадает ли голова списка L с X, а затем
(2) проверить, не принадлежит ли X хвосту списка L.

С другой стороны, **принадлежит1**, наоборот, имеет очевидный декларативный смысл, но его процедурный смысл не столь очевиден.
Интересным упражнением было бы следующее: выяснить, как в действительности **принадлежит1** что-либо вычисляет. Некоторое представление об этом мы получим, рассмотрев запись всех шагов вычисления ответа на вопрос:

```
?- принадлежит1( b, [a,b,c] ).
```

На рис. 3.3 приведена эта запись. Из нее можно заключить, что **принадлежит1** ведет себя точно так же, как и **принадлежит**. Он просматривает список элемент за элементом до тех пор, пока не найдет нужный или пока не кончится список.

Упражнения

3.1. (a) Используя отношение **конк**, напишите цель, соответствующую вычеркиванию трех последних элементов списка L, результат – новый список L1. Указание: L – конкатенация L1 и трехэлементного списка.

(b) Напишите последовательность целей для порождения списка L2, получающегося из списка L вычеркиванием его трех первых и трех последних элементов.

3.2. Определите отношение

последний(Элемент, Список)

так, чтобы **Элемент** являлся последним элементом списка **Список**. Напишите два варианта определения: (a) с использованием отношения **конк**, (b) без использования этого отношения.

3.2.3. Добавление элемента

Наиболее простой способ добавить элемент в список – это вставить его в самое начало так, чтобы он стал его новой головой. Если X – это новый элемент, а список, в который X добавляется – L, тогда результирующий список – это просто

[X | L]

Таким образом, для того, чтобы добавить новый элемент в начало списка, не надо использовать никакой процедуры. Тем не менее, если мы хотим определить такую процедуру в явном виде, то ее можно представить в форме такого факта:

добавить(X, L, [X | L]).

3.2.4. Удаление элемента

Удаление элемента X из списка L можно запрограммировать в виде отношения

удалить(X, L, L1)

где L1 совпадает со списком L, у которого удален элемент X. Отношение **удалить** можно определить аналогично отношению принадлежности. Имеем снова два случая:

(1) Если X является головой списка, тогда результатом удаления будет хвост этого списка.

(2) Если X находится в хвосте списка, тогда его нужно удалить оттуда.

```
удалить( X, [X | Хвост], Хвост).

удалить( X, [Y | Хвост], [ Y | Хвост1] ) :-
    удалить( X, Хвост, Хвост1).
```

как и принадлежит, отношение удалить по природе своей недетерминировано. Если в списке встречается несколько вхождений элемента X, то удалить сможет исключить их все при помощи возвратов. Конечно, вычисление по каждой альтернативе будет удалять лишь одно вхождение X, оставляя остальные в неприкосновенности. Например:

```
?- удалить( a, [a,b,a,a], L].

L = [b,a,a];
L = [a,b,a];
L = [a,b,a];

no                (нет)
```

При попытке исключить элемент, не содержащийся в списке, отношение удалить потерпит неудачу.

Отношение удалить можно использовать в обратном направлении для того, чтобы добавлять элементы в список, вставляя их в произвольные места. Например, если мы хотим во все возможные места списка [1,2,3] вставить атом a, то мы можем это сделать, задав вопрос: «Каким должен быть список L, чтобы после удаления из него элемента a получился список [1,2,3]?»

```
?- удалить( a, L, [1,2,3] ).

L = [a,1,2,3];
L = [1,a,2,3];
L = [1,2,a,3];
L = [1,2,3,a];

no                (нет)
```

Вообще операция по внесению X в произвольное место некоторого списка Список, дающее в результате БольшийСписок, может быть определена предложением:

```
внести( X, Список, БольшийСписок) :-
    удалить( X, БольшийСписок, Список).
```

В принадлежит1 мы изящно реализовали отношение принадлежности через конк. Для проверки на принадлежность можно также использовать и удалить. Идея простая: некоторый X принадлежит списку Список, если X можно из него удалить:

```
принадлежит2( X, Список) :-
    удалить( X, Список, _).
```

3.2.5. Подсписок

Рассмотрим теперь отношение подсписок. Это отношение имеет два аргумента – список L и список S, такой, что S содержится в L в качестве подсписка. Так отношение

```
подсписок( [c,d,e], [a,b,c,d,e,f] )
```

имеет место, а отношение

```
подсписок( [c,e], [a,b,c,d,e,f] )
```

нет. Пролог-программа для отношения подсписок может основываться на той же идее, что и принадлежит1, только на этот раз отношение более общо (см. рис. 3.4).

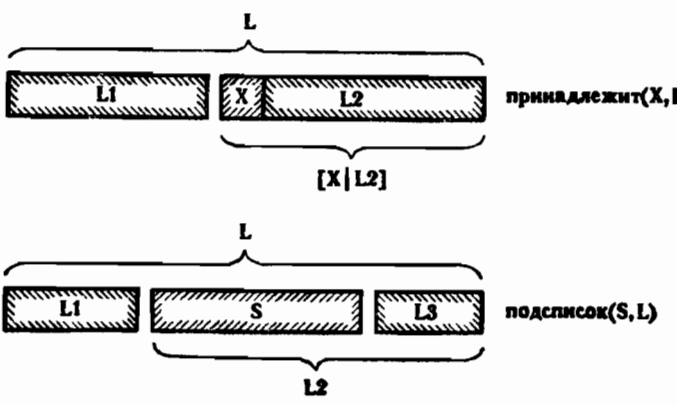


Рис. 3.4. Отношения принадлежит и подсписок.

Его можно сформулнровать так:

S является подспнском L, если

(1) L можно разбнть на два спнска L1 н L2 н
(2) L2 можно разбнть на два спнска S н L3.

Как мы вндели раньше, отношенне **конк** можно нспользовать для разбнення спнсков. Поэтому вышепрнведенную формулнровку можно выразить на Прологе так:

```
подсписок( S, L) :-
    конк( L1, L2, L),
    конк( S, L3, L2).
```

Ясно, что процедуру **подсписок** можно гнбко нспользовать разлнчнымн способами. Хотя она предназначалась для проверки, является ли какой-лнбо спнсок подспнском другого, ее можно использовать, например, для нахождення всех подспнсков данного спнска:

```
?- подсписок( S, [a,b,c] ).

S = [];
S = [a];
S = [a,b];
S = [a,b,c];
S = [b];
...
```

3.2.6. Перестановки

Иногда бывает полезно постронть все перестановкн некоторого заданного спнска. Для этого мы определим отношенне **перестановка** с двумя аргументамн. Аргументы – это два спнска, однн нз которых является перестановкой другого. Мы намереваемся порождать перестановкн списка с помощью механнзма автоматнческого перебора, нспользуя процедуру **перестановка**, подобно тому, как это делается в следующем прнмере:

```
?- перестановка( [a,b,c], P).

P = [a,b,c];
P = [a,c,b];
P = [b,a,c];
...
```

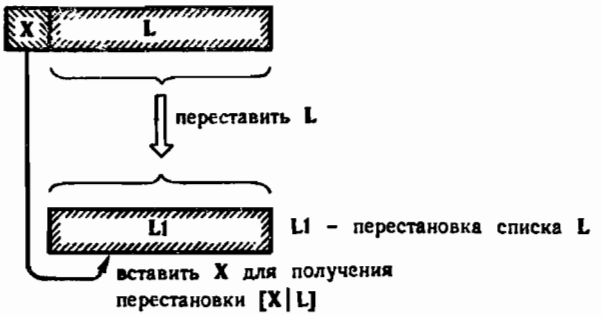


Рис. 3.5. Одни из способов построения перестановки списка [X|L].

Программа для отношения **перестановка** в свою очередь опять может основываться на рассмотрении двух случаев в зависимости от вида первого списка:

(1) Если первый список пуст, то и второй список должен быть пустым.

(2) Если первый список не пуст, тогда он имеет вид [X|L], и перестановку такого списка можно построить так, как это показано на рис. 3.5: вначале получить список L1 – перестановку L, а затем внести X в произвольную позицию L1.

Два прологовских предложения, соответствующих этим двум случаям, таковы:

```
перестановка( [], []).

перестановка( [X | L], P) :-
    перестановка( L, L1),
    внести( X, L1, P).
```

Другой вариант этой программы мог бы предусматривать удаление элемента X из первого списка, перестановку оставшейся его части – получение списка P, а затем добавление X в начало списка P. Соответствующая программа такова:

```
перестановка2( [], []).

перестановка2( L, [X | P] ) :-
    удалить( X, L, L1),
    перестановка2( L1, P).
```

Поучительно проделать несколько экспериментов с нашей программой перестановки. Ее нормальное использование могло бы быть примерно таким:

```
?- перестановка( [красный,голубой,зеленый], Р).
```

Как и предполагалось, будут построены все шесть перестановок:

```
Р = [ красный, голубой, зеленый];
Р = [ красный, зеленый, голубой];
Р = [ голубой, красный, зеленый];
Р = [ голубой, зеленый, красный];
Р = [ зеленый, красный, голубой];
Р = [ зеленый, голубой, красный];

no                    (нет)
```

Приведем другой вариант использования процедуры **перестановка**:

```
?- перестановка( L, [a,b,c] ).
```

Наша первая версия, **перестановка**, произведет успешную конкретизацию L всеми шестью перестановками. Если пользователь потребует новых решений, он никогда не получит ответ «нет», поскольку программа войдет в бесконечный цикл, пытаясь отыскать новые несуществующие перестановки. Вторая версия, **перестановка2**, в этой ситуации найдет только первую (идентичную) перестановку, а затем сразу зациклится. Следовательно, при использовании этих отношений требуется соблюдать осторожность.

Упражнения

3.3. Определите два предиката

четнаядлина(Список) и **нечетнаядлина(Список)**

таким образом, чтобы они были истинными, если их аргументом является список четной или нечетной длины соответственно. Например, список [a,b,c,d] имеет четную длину, а [a,b,c] – нечетную.

3.4. Определите отношение

обращение(Список, ОбращенныйСписок),

которое обращает списки. Например,

обращение([a,b,c,d],[d,c,b,a]).

3.5. Определнте предикат **палиндром(Список)**. Список называется палиндромом, если он читается одинаково, как слева направо, так и справа налево. Напрнмер, [м, а, д, а, м].

3.6. Определнте отношение

сдвиг(Список1, Список2)

таким образом, чтобы **Список2** представлял собой **Список1**, «цнклнческн сдвинутый» влево на один снмвол. Например,

```
?- сдвиг( [1,2,3,4,5], L1),
   сдвиг( L1, L2)
```

дает

```
L1 = [2,3,4,5,1]
L2 = [3,4,5,1,2]
```

3.7. Определнте отношенне

перевод(Список1, Список2)

для перевода списка чисел от 0 до 9 в список соответствующнх слов. Например,

перевод([3,5,1,3], [три, пять, один, три])

Используйте в качестве вспомогательных следующне отношения:

```
означает( 0, нуль).
означает( 1, один).
означает( 2, два).
...
```

3.8. Определнте отношение

подмножество(Множество, Подмножество)

где **Миожество** и **Подмножество** – два списка, представляющне два множества. Желательно иметь возможиость использовать это отношение не только для проверки включения одного миожества в другое, но и для порождения всех возможных подмножеств заданного множества. Например:

?- подмножество([a,b,c], S).

```
S = [a,b,c];
S = [b,c];
S = [c];
S = [];
S = [a,c];
S = [a];
...
```

3.9. Определите отношение

разбениесписка(Список, Список1, Список2)

так, чтобы оно распределяло элементы списка между двумя списками **Список1** и **Список2** и чтобы эти списки были примерно одинаковой длины. Например,

разбениесписка([a,b,c,d,e],[a,c,e],[b,d]).

3.10. Перепишите программу об обезьяне и бананах из главы 2 таким образом, чтобы отношение

можетзавладеть(Состояние, Действия)

давало не только положительный или отрицательный ответ, но и порождало последовательность действий обезьяны, приводящую ее к успеху. Пусть **Действия** будет такой последовательностью, представленной в виде списка ходов:

```
Действия = [ перейти(дверь,окно),
             передвинуть(окно,середина),
             залезть, схватить ]
```

3.11. Определите отношение

линеаризация(Список, ЛинейныйСписок)

где **Список** может быть списком списков, а **ЛинейныйСписок** – это тот же список, но «выровненный» таким образом, что элементы его подсписков составляют один линейный список. Например:

```
? - линеаризация([a,d,[c,d],[],[[[e]]],f, L).
L = [a,b,c,d,e,f]
```

3.3. Операторная запись (нотация)

В математике мы привыкли записывать выражения в таком виде:

*2*a + b*c*

где + и * – это операторы, а 2, *a*, *b*, *c* – аргументы. В частности, + и * называют *инфиксными* операторами, поскольку они появляются *между* своими аргументами. Такие выражения могут быть представлены в виде деревьев, как это сделано на рис. 3.6, и записаны как прологовские термы с + и * в качестве функторов:

```
+( *(2,a), *(b,c) )
```

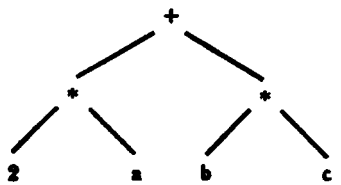


Рис. 3.6. Представление выражения 2*a+b*c в виде дерева.

Поскольку мы обычно предпочитаем записывать такие выражения в привычной инфиксной форме операторов, Пролог обеспечивает такое удобство. Поэтому наше выражение, записанное просто как

```
2*a + b*c
```

будет воспринято правильно. Однако это лишь внешнее представление объекта, которое будет автоматически преобразовано в обычную форму прологовских термов. Такой терм выводится пользователю снова в своей внешней инфиксной форме.

Выражения рассматриваются Прологом просто как дополнительный способ записи, при котором не вводятся какие-либо новые принципы структуризации объектов данных. Если мы напишем a + b, Пролог

поймет эту запись, как если бы написали +(a,b). Для того, чтобы Пролог правильно воспринимал выражении типа a + b*c, он должен знать, что * связывает сильнее, чем +. Будем говорить, что + имеет более низкий приоритет, чем *. Поэтому вариан интерпретации выражений зависит от приоритетов операторов. Например, выражение a + b*c, в принципе можно понимать и как

```
+( a, *(b,c) )
```

и как

```
*( +( a,b), c)
```

Общее правило состоит в том, что оператор с самым низким приоритетом расценивается как главный функтор терма. Если мы хотим, чтобы выражении, содержащие + и *, понимались в соответствии с обычными соглашениями, то + должен яметь более низкий приоритет, чем *. Тогда выражение a + b*c означает то же, что и a + (b*c). Если имеется в виду другая интерпретации, то это надо указать ивно с помощью скобок, напрямер (a+b)*c.

Программист может вводить свои собственные операторы. Так, например, можно определить атомы имеет и поддерживает в качестве инфиксных операторов, а затем записывать в программе факты вида:

```
питер имеет информацию.
пол поддерживает стол.
```

Эти факты в точности эквивалентны следующим:

```
имеет( питер, информацию).
поддерживает( пол, стол).
```

Программист определяет новые операторы, вводи в программу особый вид предложений, которые иногда называют *директивами*. Такие предложении играют роль определений новых операторов. Определение оператора должно появиться в программе раньше, чем любое выражение, использующее этот оператор. Например, оператор имеет можно определить директивой

```
:- op( 600, xfx, имеет).
```

Такая запись сообщит Прологу, что мы хотим использовать «имеет» в качестве оператора с приоритом

600 и типом 'xfx', обозначающим одну из разновидностей инфиксного оператора. Форма спецификатора 'xfx' указывает на то, что оператор, обозначенный через 'f', располагается между аргументами, обозначенными через 'x'.

Обратите внимание на то, что определения операторов не содержат описания каких-либо операций или действий. В соответствии с принципами языка *ни с одним оператором не связывается каких-либо операций над данными* (за исключением особых, редких случаев). Операторы обычно используются так же, как и функторы, только для объединения объектов в структуры и не вызывают действий над данными, хотя само слово «оператор», казалось бы, должно подразумевать какое-то действие.

Имена операторов это атомы, а их приоритеты – точнее, номера их приоритетов – должны находиться в некотором диапазоне, зависящем от реализации. Мы будем считать, что этот диапазон располагается в пределах от 1 до 1200[1].

Существуют три группы типов операторов, обозначаемые спецификаторами, похожими на **xfx**:

(1) инфиксные операторы трех типов:

xfx xfy yfx

(2) префиксные операторы двух типов:

fx fy

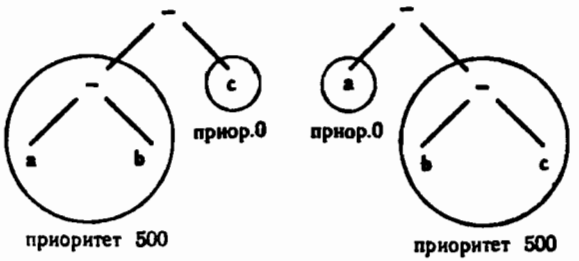
(3) постфиксные операторы двух типов:

xf yf

Спецификаторы выбраны с таким расчетом, чтобы нагляднее отразить структуру выражения, в котором 'f' соответствует оператору, а 'x' и 'y' представляют его аргументы. Расположение 'f' между аргументами указывает на то, что оператор инфиксный. Префиксные и постфиксные спецификаторы содержат только один аргумент, который, соответственно, либо следует за оператором, либо предшествует ему.

---

[1] Чем выше приоритет, тем меньше его номер. – *Прим. перев.*



Интерпретация 1: (a - b) - c Интерпретация 2: a - (b - c)

Рис. 3.7. Две интерпретации выражения a-b-c в предположении, что '-' имеет приоритет 500. Если тип '-' есть yfx, то интерпретация 2 неверна, так как приоритет b-c не выше, чем приоритет '-'.

Между 'x' и 'y' есть разница. Для ее объяснения нам потребуется ввести понятие *приоритета аргумента*. Если аргумент заключен в скобки или не имеет структуры (является простым объектом), тогда его приоритет равен 0; если же он структурный, тогда его приоритет равен приоритету его главного функтора. С помощью 'x' обозначается аргумент, чей приоритет должен быть строго выше приоритета оператора (т. е. его номер строго меньше номера приоритета оператора); с помощью 'y' обозначается аргумент, чей приоритет выше или равен приоритету оператора.

Такие правила помогают избежать неоднозначности при обработке выражений, в которых встречаются операторы с одинаковым приоритетом. Например, выражение

```
a-b-c
```

обычно понимается как (a-b)-c , а не как a-(b-c). Чтобы обеспечить такую обычную интерпретацию, оператор '-' следует определять как yfx. На рис. 3.7 показано, каким образом исключается вторая интерпретация.

В качестве еще одного примера рассмотрим оператор not (логическое отрицание «не»). Если not оп-

ределен как fy, тогда выражение

```
not not p
```

записано верно; однако, если not определен как fx, оно некорректно, потому что аргументом первого not является структура not p, которая имеет тот же приоритет, что и not. В этом случае выражение следует писать со скобками:

```
not (not p)
```

```
:- op( 1200, xfx, ':-').
:- op( 1200, fx, [:-, ?-] ).
:- op( 1100, xfy, ';').
:- op( 1000, xfy, ',').
:- op(700,xfx,[=,is,<,>,=<,>=,==,=\=,\==,=:=]).
:- op( 500, yfx, [+, -] ).
:- op( 500, fx, [+, -, not] ).
:- op( 400, yfx, [*,/, div] ).
:- op( 300, xfx, mod).
```

Рис. 3.8. Множество предопределенных операторов.

Для удобства некоторые операторы в пролог-системах определены заранее, чтобы ими можно было пользоваться сразу, без какого-либо определения их в программе. Набор таких операторов и их приоритеты зависят от реализации. Мы будем предполагать, что множество этих «стандартных» операторов ведет себя так, как если бы оно было определено с помощью предложений, приведенных на рис. 3.8. Как видно из того же рисунка, несколько операторов могут быть определены в одном предложении, если только они все имеют одинаковый приоритет и тип. В этом случае имена операторов записываются в виде списка. Использование операторов может значительно повысить наглядность, «читабельность» программы. Для примера предположим, что мы пишем программу для обработки булевских выражений. В такой программе мы, возможно, захотим записать утверждение одной

из теорем де Моргана, которое в математических обозначениях записывается так:

$$\sim (A \,\&\, B) \Longleftrightarrow \sim A \vee \sim B$$

Приведем один из способов записи этого утверждения в виде прологовского предложения:

эквивалентно(not(и(A,B)), или(not(A),not(B))).

Однако хорошим стилем программирования было бы попытаться сохранить по возможности больше сходства между видом записи исходной задачи и видом, используемом в программе ее решения. В нашем примере этого можно достичь почти в полной мере, применив операторы. Подходящее множество операторов для наших целей можно определить так:

```
:- op( 800, xfx, <===>).
:- op( 700, xfy, v).
:- op( 600, xfy, &).
:- op( 500, fy, ~).
```

Теперь правило де Моргана можно записать в виде следующего факта:

```
~(A & B) <===> ~A v ~B.
```

В соответствии с нашими определениями операторов этот терм понимается так, как это показано на рис. 3.9.

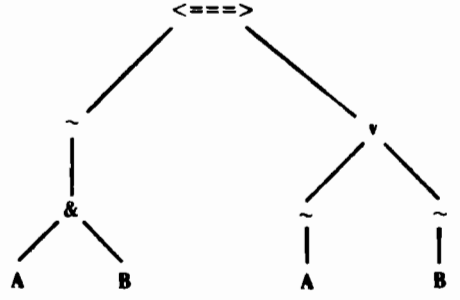


Рис. 3.9. Интерпретация терма ~(A & B) <===> ~A v ~B

Подытожим:

- Наглядность программы часто можно улучшить, использовав операторную нотацию. Операторы бывают инфиксные, префиксные и постфиксные.
- В принципе, с оператором не связываются никакие действия над данными, за исключением особых случаев. Определение оператора не содержит описания каких-либо действий, оно лишь вводит новый способ записи. Операторы, как и функторы, лишь связывают компоненты в единую структуру.
- Программист может вводить свои собственные операторы. Каждый оператор определяется своим именем, приоритетом и типом.
- Номер приоритета – это целое число из некоторого диапазона, скажем, между 1 и 1200. Оператор с самым большим номером приоритета соответствует главному функтору выражения, в котором этот оператор встретился. Операторы с меньшими номерами приоритетов связывают свои аргументы сильнее других операторов.
- Тип оператора зависит от двух условий: (1) его расположения относительно своих аргументов, (2) приоритета его аргументов по сравнению с его собственным. В спецификаторах, таких, как **xfy**, **x** обозначает аргумент, чей номер приоритета строго меньше номера приоритета оператора; **y** – аргумент с номером приоритета, меньшим или равным номеру приоритета оператора.

Упражнения

3.12. Если принять такие определения

```
:- op( 300, xfy, играет_в).
:- op( 200, xfy, и).
```

то два следующих терма представляют собой синтаксически правильные объекты:

```
Терм1 = джимми играет_в футбол и сквош
Терм1 = сьюзан играет_в теннис и баскетбол и
        волейбол
```

Как эти термы интерпретируются пролог-системой? Каковы их главные функторы и какова их структура?

3.13. Предложите подходящее определение операторов («работает», «в», «нашем»), чтобы можно было писать предложения типа:

```
диана работает секретарем в нашем отделе.
```

а затем спрашивать:

```
?- Кто работает секретарем в нашем отделе.
Кто = диана

?- диана работает Кем.
Кем = секретарем в нашем отделе
```

3.14. Рассмотрим программу:

```
t( 0+1, 1+0).

t( X+0+1, X+1+0).

t( X+1+1, Z) :-
    t( X+1, X1),
    t( X1+1, Z).
```

Как данная программа будет отвечать на нижеперечисленные вопросы, если '+' – это (как обычно) инфиксный оператор типа yfx?

(a) `?- t( 0+1, A).`

(b) `?- t( 0+1+1, B).`

(c) `?- t( 1+0+1+1+1, C).`

(d) `?- t( D, 1+1+1+0).`

3.15. В предыдущем разделе отношения между списками мы записывали так:

```
принадлежит( Элемент, Список),
конк( Список1, Список2, Список3),
удалить( Элемент, Список, НовыйСписок), ...
```

Предположим, что более предпочтительной для нас является следующая форма записи:

```
Элемент входит_в Список,
конкатенация_списков Список1 и Список2
    дает Список3,
удаление_элемента Элемент из_списка Список
    дает НовыйСписок, ...
```

Определите операторы «входит_в», «конкатенация_списков», «и» и т.д. таким образом, чтобы обеспечить эту возможность. Переопределите также и соответствующие процедуры.

3.4. Арифметические действия

Пролог рассчитан главным образом на обработку символьной информации, при которой потребность в арифметических вычислениях относительно мала. Поэтому и средства для таких вычислений довольно просты. Для осуществления основных арифметических действий можно воспользоваться несколькими предопределенными операторами.

+	сложение
–	вычитание
*	умножение
/	деление
mod	модуль, остаток от целочисленного деления

Заметьте, что это как раз тот исключительный случай, когда оператор может и в самом деле произвести некоторую операцию. Но даже и в этом случае требуется дополнительное указание на выполнение действия. Пролог-система знает, как выполнять вычисления, предписываемые такими операторами, но этого недостаточно для их непосредственного использования. Следующий вопрос – наивная попытка произвести арифметическое действие:

```
?- X = 1 + 2.
```

Пролог-система «спокойно» ответит

```
X = 1 + 2
```

а не X = 3, как, возможно, ожидалось. Причина этого проста: выражение 1 + 2 обозначает лишь прологовский терм, в котором + является функтором, а 1 и 2 – его аргументами. В вышеприведенной цели нет ничего, что могло бы заставить систему выполнить операцию сложения. Для этого в Прологе существует специальный оператор is (есть). Этот оператор заставит систему выполнить вычисление. Таким образом, чтобы правильно активизировать арифметическую операцию, надо написать:

```
?- X is 1 + 2.
```

Вот теперь ответ будет

```
X = 3
```

Сложение здесь выполняется специальной процедурой, связанной с оператором +. Мы будем называть такие процедуры *встроенными*.

В Прологе не существует общепринятой нотации для записи арифметических действий, поэтому в разных реализациях она может слегка различаться. Например, оператор '/' может в одних реализациях обозначать целочисленное деление, а в других – вещественное. В данной книге под '/' мы подразумеваем вещественное деление, для целочисленного же будем использовать оператор **div**. В соответствии с этим, на вопрос

```
?- X is 3/2,
   Y is 3 div 2.
```

ответ должен быть такой:

```
X = 1.5
Y = 1
```

Левым аргументом оператора is является простой объект. Правый аргумент – арифметическое выражение, составленное с помощью арифметических операторов, чисел и переменных. Поскольку оператор is запускает арифметические вычисления, к моменту

начала вычисления этой цели все ее переменные должны быть уже конкретизированы какими-либо числами. Приоритеты этих предопределенных арифметических операторов (см. рис. 3.8) выбраны с таким расчетом, чтобы операторы применялись к аргументам в том порядке, который принят в математике. Чтобы изменить обычный порядок вычислений, применяются скобки (тоже, как в математике). Заметьте, что +, −, *, / и div определены, как yfx, что определяет порядок их выполнения слева направо. Например,

```
X is 5 - 2 - 1
```

понимается как

```
X is (5 - 2) - 1
```

Арифметические операции используются также и при *сравнении* числовых величин. Мы можем, например, проверить, что больше – 10000 или результат умножения 277 на 37, с помощью цели

```
?- 277 * 37 > 10000.
yes                (да)
```

Заметьте, что точно так же, как и is, оператор '>' вызывает выполнение вычислений.

Предположим, у нас есть программа, в которую входит отношение рожд, связывающее имя человека с годом его рождения. Тогда имена людей, родившихся между 1950 и 1960 годами включительно, можно получить при помощи такого вопроса:

```
?- рожд( Имя, Год),
   Год >= 1950,
   Год <= 1960.
```

Ниже перечислены операторы сравнения:

X > Y	X больше Y
X < Y	X меньше Y
X >= Y	X больше или равен Y
X =< Y	X меньше или равен Y
X =:= Y	величины X и Y совпадают (равны)
X =\= Y	величины X и Y не равны

Обратите внимание на разницу между операторами сравнения '=' и '=:=', например, в таких целях как X = Y и X =:= Y. Первая цель вызовет сопоставление объектов X и Y, и, если X и Y сопоставимы, возможно, приведет к конкретизации каких-либо переменных в этих объектах. Никаких вычислений при этом производиться не будет. С другой стороны, X =:= Y вызовет арифметическое вычисление и не может привести к конкретизации переменных. Это различие можно проиллюстрировать следующими примерами:

```
?- 1 + 2 =:= 2 + 1.
yes
?- 1 + 2 = 2 + 1.
no
?- 1 + A = B + 2.
A = 2
B = 1
```

Давайте рассмотрим использование арифметических операций на двух простых примерах. В первом примере ищется наибольший общий делитель; во втором – определяется количество элементов в некотором списке.

Если заданы два целых числа X и Y, то их наибольший общий делитель Д можно найти, руководствуясь следующими тремя правилами:

(1) Если X и Y равны, то Д равен X.

(2) Если X > Y, то Д равен наибольшему общему делителю X разности Y – X.

(3) Если Y < X, то формулировка аналогична правилу (2), если X и Y поменять в нем местами.

На примере легко убедиться, что эти правила действительно позволяют найти наибольший общий делитель. Выбрав, скажем, X = 20 и Y = 25, мы, руководствуясь приведенными выше правилами, после серии вычитаний получим Д = 5.

Эти правила легко сформулировать в виде прологовской программы, определив трехаргументное отношение, скажем

```
нод( X, Y, Д)
```

Тогда наши три правила можно выразить тремя предложениями так:

```
нод( X, X, X).
нод( X, Y, Д) :-
    X < Y,
    Y1 is Y - X,
    нод( X, Y1, Д).
нод( X, Y, Д) :-
    Y < X,
    нод( Y, X, Д).
```

Разумеется, с таким же успехом можно последнюю цель в третьем предложении заменить двумя:

```
X1 is X - Y,
нод( X1, Y, Д)
```

В нашем следующем примере требуется произвести некоторый подсчет, для чего, как правило, необходимы арифметические действия. Примером такой задачи может служить вычисление длины какого-либо списка; иначе говоря, подсчет числа его элементов. Определим процедуру

```
длина( Список, N)
```

которая будет подсчитывать элементы списка **Список** и конкретизировать **N** полученным числом. Как и раньше, когда речь шла о списках, полезно рассмотреть два случая:

(1) Если список пуст, то его длина равна 0.

(2) Если он не пуст, то **Список=[Голова1|Хвост]** и его длина равна 1 плюс длина хвоста **Хвост**.

Эти два случая соответствуют следующей программе:

```
длина( [], 0).
длина( [_| Хвост], N) :-
    длина( Хвост, N1),
    N is 1 + N1.
```

Применить процедуру **длина** можно так:

```
?- длина( [a,b,[c,d],e], N).
N = 4
```

Заметим, что во втором предложении этой процедуры две цели его тела нельзя поменять местами. Причина этого состоит в том, что переменная N1 должна быть конкретизирована до того, как начнет вычисляться цель

```
N is 1 + N1
```

Таким образом мы видим, что введение встроенной процедуры **is** привело нас к примеру отношения, чувствительного к порядку обработки предложений и целей. Очевидно, что процедурные соображения для подобных отношений играют жизненно важную роль.

Интересно посмотреть, что произойдет, если мы попытаемся запрограммировать отношение **длина** без использования **is**. Попытка может быть такой:

```
длина1( [], 0).
длина1( [ _ | Хвост], N) :-
    длина1( Хвост, N1),
    N = 1 + N1.
```

Теперь уже цель

```
?- длина1( [a,b,[c,d],e], N).
```

породит ответ:

```
N = 1+(1+(1+(1+0)))
```

Сложение ни разу в действительности не запускалось и поэтому ни разу не было выполнено. Но в процедуре **длина1**, в отличие от процедуры **длина**, мы можем поменять местами цели во втором предложении:

```
длина1( [ _ | Хвост], N) :-
    N = 1 + N1,
    длина1( Хвост, N1).
```

Такая версия **длина1** будет давать те же результаты, что и исходная. Ее можно записать короче:

```
длина1( [ _ | Хвост], 1 + N) :-
    длина1( Хвост, N).
```

и она и в этом случае будет давать те же результаты. С помощью **длина1**, впрочем, тоже можно вычислить количество элементов списка:

```
?- длина( [a,b,c], N), Длина is N.

N = 1+(1+(1+0))
Длина = 3
```

Итак:

- Для выполнения арифметических действий используются встроенные процедуры.
- Арифметические операции необходимо явно запускать при помощи встроенной процедуры **is**. Встроенные процедуры связаны также с предопределенными операторами **+**, **–**, *****, **/**, **div** и **mod**.
- К моменту выполнения операций все их аргументы должны быть конкретизированы числами.
- Значения арифметических выражений можно сравнивать с помощью таких операторов, как **<**, **=<** и т.д. Эти операторы вычисляют значения своих аргументов.

Упражнения

3.16. Определите отношение

max(X, Y, Max)

так, чтобы **Max** равнялось **наибольшему из двух** чисел **X** и **Y**.

3.17. Определите предикат

максспис(Список, Max)

так, чтобы **Max** равнялось **наибольшему из чисел, входящих в Список.**

3.18. Определите предикат

сумспис(Список, Сумма)

так, чтобы **Сумма** равиялось сумме чисел, входящих в **Список**.

3.19. Определите предикат

упорядоченный(Список)

который принимает значение истина, если **Список** представляет собой упорядоченный список чисел. Например: **упорядоченный([1,5,6,6,9,12]).**

3.20. Определите предикат

подсумма(Множ, Сумма, ПодМнож)

где **Множ** – это список чисел, **Подмнож** – подмножество этих чисел, а сумма чисел из **ПодМнож** равна **Сумма**. Например:

```
?- подсумма( [1,2.5.3.2], 5, ПМ).

ПМ = [1,2,2];
ПМ = [2,3];
ПМ = [5];
...
```

3.21. Определите процедуру

между(N1, N2, X)

которая, с помощью перебора, порождает все целые числа X, отвечающие условию N1 ≤X ≤ N2.

3.22. Определите операторы 'если', 'то', 'иначе' и ':=' таким образом, чтобы следующее выражение стало правильным термом:

```
если X > Y то Z := X иначе Z := Y
```

Выберите приоритеты так, чтобы 'если' стал главным функтором. Затем определите отношение 'если' так, чтобы оно стало как бы маленьким интерпретатором выражений типа 'если–то–иначе'. Например, такого

```
если Вел1 > Вел2 то Перем := Вел3
иначе Перем := Вел4
```

где **Вел1**, **Вел2**, **Вел3** и **Вел4** – числовые величины

(или переменные, конкретизированные числами), а **Перем** – переменная. Смысл отношения 'если' таков: если значение **Вел1** больше значения **Вел2**, тогда **Перем** конкретизируется значением **Вел3**, в противном случае – значением **Вел4**. Приведем пример использования такого интерпретатора:

```
?- X = 2, Y = 3,
   Вел2 is 2*X,
   Вел4 is 4*X,
   Если Y>Вел2 то Z:=Y иначе Z:=Вел4.

   Если Z > 5  то W := 1 иначе W :=0.

X = 2
Y = 3
Z = 8
W = 1

Вел2 = 4
Вел4 = 8
```

Резюме

- Список – часто используемая структура. Он либо пуст, либо состоит из *головы* и *хвоста*, который в свою очередь также является списком. Для списков в Прологе имеется специальная нотация.
- В данной главе рассмотрены следующие операции над списками: принадлежность к списку, конкатенация, добавление элемента, удаление элемента, удаление подсписка.
- *Операторная запись* позволяет программисту приспособить синтаксис программ к своим конкретным нуждам. С помощью операторов можно значительно повысить наглядность программ.
- Новые операторы определяются с помощью директивы op, в которой указываются его имя, тип и приоритет.

- Как правило, с оператором не связывается никакой операции; оператор это просто синтаксическое удобство, обеспечивающее альтернативный способ записи термов.
- Арифметические операции выполняются с помощью встроенных процедур. Вычисление арифметических выражений запускается процедурой **is**, а также предикатами сравнения <, =< и т.д.
- Понятия, введенные в данной главе:

  список, голова списка, хвост списка
  списковая нотация
  операторы, операторная нотация
  инфиксные, префиксные и постфиксные операторы
  приоритет операторов
  арифметические встроенные процедуры

4 ИСПОЛЬЗОВАНИЕ СТРУКТУР: ПРИМЕРЫ

Структуры данных вместе с сопоставлением, автоматическими возвратами и арифметикой представляют собой мощный инструмент программирования. В этой главе мы расширим навыки использования этого инструмента при помощи следующих учебных программных примеров: получение структурированной информации из базы данных, моделирование недетерминированного автомата, планирование маршрута поездки и решение задачи о расстановке восьми ферзей на шахматной доске. Мы увидим также, как в Прологе реализуется принцип абстракции данных.

4.1. Получение структурированной информации из базы данных

Это упражнение развивает навыки представления структурных объектов данных и управления ими. Оно показывает также, что Пролог является естественным языком запросов к базе данных.

База данных может быть представлена на Прологе в виде множества фактов. Например, в базе данных о семьях каждая семья может описываться одним предложением. На рис. 4.1 показано, как информацию о каждой семье можно представить в виде структуры. Каждая семья состоит из трех компонент: мужа, жены и детей. Поскольку количество детей в разных семьях может быть разным, то их целесообразно представить в виде списка, состоящего из произвольного числа элементов. Каждого члена семьи в свою очередь можно представить структурой, состоящей из

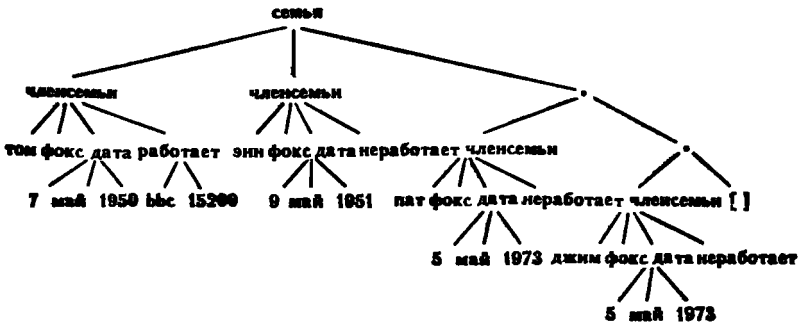


Рис. 4.1. Структурированная информация о семье.

четырех компонент: имени, фамилии, даты рождения и работы. Информация о работе – это либо «не работает», либо указание места работы и оклада (дохода). Информацию о семье, изображенной на рис. 4.1, можно занести в базу данных с помощью предложения:

```
семья(
    членсемьи( том,фокс,дата(7,май,1950),
                работает(bbc,15200) ),
    членсемьи( эин, фокс, дата(9,май,1951),
                неработает),
    [членсемьи( пат, фокс, дата(5,май,1973),
                неработает),
     членсемьи( джим, фокс, дата(5,май,1973),
                неработает) ] ).
```

Тогда база данных будет состоять из последовательности фактов, подобных этому, и описывать все семьи, представляющие интерес для нашей программы.

В действительности Пролог очень удобен для извлечения необходимой информации из такой базы данных. Здесь хорошо то, что можно ссылаться на объекты, не указывая в деталях всех их компонент. Можно задавать только *структуру* интересующих нас объектов и оставлять конкретные компоненты без точного описания или лишь с частичным описанием. На рис. 4.2 приведено несколько примеров. Так, в запросах к базе данных можно ссылаться на всех Армстронгов с помощью терма

```
семья( членсемьи( _, армстронг, _, _ ), _, _ )
```

Символы подчеркивания обозначают различные анонимные переменные, значения которых нас не заботят. Далее можно сослаться на все семьи с тремя детьми при помощи терма:

```
семья( _, _, [_, _, _ ] )
```

Чтобы найти всех замужних женщин, имеющих по крайней мере троих детей, можно задать вопрос:

```
?- семья(_,членсемьи(Имя,Фамилия,_,_),[_,_,_|_]).
```

Главным моментом в этих примерах является то, что указывать интересующие нас объекты можно не только по их содержимому, но и по их структуре. Мы задаем одну структуру и оставляем ее аргументы в виде слотов (пропусков).

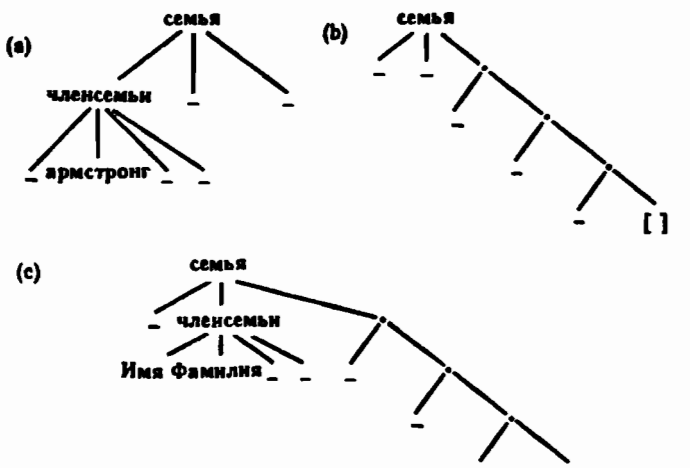


Рис. 4.2. Описания объектов по их структурным свойствам; (a) любая семья Армстронгов; (b) любая семья, имеющая ровно трех детей; (c) любая семья, имеющая по крайней мере трех ребенка. Структура (c) дает возможность получить имя и фамилию жены конкретизацией переменных Имя и Фамилия.

Можно создать набор процедур, который служил бы утилитой, делающей взаимодействие с нашей базой данных более удобным. Такие процедуры являлись бы частью пользовательского интерфейса. Вот некоторые полезные процедуры для нашей базы данных:

```
муж( X) :-                          % X - муж
    семья( X, _, _).

жена( X) :-                         % X - жена
    семья( _, X, _).

ребенок( X) :-                      % X - ребенок
    семья( _, _, Дети),
    принадлежит( X, Дети).

принадлежит( X, [X | L ].

принадлежит( X, [Y | L ] :-
    принадлежит( X, L).

существует( Членсемьи) :-
            % Любой член семьи в базе данных
    муж( Членсемьи);
    жена( Членсемьи);
    ребенок( Членсемьи).

датарождения(Членсемьи(_,_,Дата,_),Дата).

доход( Членсемьи(_,_,_,работает(_,S) ),S).
                                    % Доход работающего

доход( Членсемьи( _, _, _, неработает), 0).
                                    % Доход неработающего
```

Этими процедурами можно воспользоваться, например, в следующих запросах к базе данных:

- Найти имена всех людей из базы данных:

  ```
  ?- существует(членсемьи(Имя,Фамилия,_,_)).
  ```

- Найти всех детей, родившихся в 1981 году:

  ```
  ?- ребенок( X),
     датарождения( X, дата( _, _, 1981) ).
  ```

- Найти всех работающих жен:

  ```
  ?- жена(членсемьи(Имя,Фамилия,_,работает(_,_))).
  ```

- Найти имена и фамилии людей, которые не работают и родились до 1963 года:

```
?- существует( членсемьи( Имя, Фамилия,
     дата(_,_,Год), неработает) ),
   Год < 1963.
```

- Найти людей, родившихся до 1950 года, чей доход меньше, чем 8000:

```
?- существует( Членсемьи),
   датарождения( Членсемьи, дата(_,_,Год) ),
   Год < 1950,
   доход( Членсемьи, Доход),
   Доход < 8000.
```

- Найти фамилии людей, имеющих по крайней мере трех детей:

```
?-семья(членсемьи(_,Фамилия,_,_),_,[_,_,_|_]).
```

Для подсчета общего дохода семьи полезно определить сумму доходов людей из некоторого списка в виде двухаргументного отношения:

```
общий( Список_Людей, Сумма_их_доходов)
```

Это отношение можно запрограммировать так:

```
общий( [], 0).                    % Пустой список людей
общий( [ Человек | Список], Сумма) :-
   доход( Человек, S),
                  % S - доход первого человека
   общий( Список, Остальные),
                  % Остальные - сумма доходов остальных
   Сумма is S + Остальные.
```

Теперь общие доходы всех семей могут быть найдены с помощью вопроса:

```
?- семья( Муж, Жена, Дети),
   общий( [Муж,  Жена | Дети], Доход).
```

Пусть отношение длина подсчитывает количество элементов списка, как это было определено в разд. 3.4. Тогда мы можем найти все семьи, которые имеют доход на члена семьи, меньший, чем 2000, при помощи вопроса:

```
?- семья( Муж, Жена, Дети),
   общий( [ Муж, Жена | Дети], Доход),
   длина([ Муж, Жена | Дети], N),
   Доход/N < 2000.
```

Упражнения

4.1. Напишите вопросы для поиска в базе данных о семьях.

(a) семей без детей;
(b) всех работающих детей;
(c) семей, где жена работает, а муж нет;
(d) всех детей, разница в возрасте родителей которых составляет не менее 15 лет.

4.2. Определите отношение

близнецы(Ребенок1, Ребенок2)

для поиска всех близнецов в базе данных о семьях.

4.2. Абстракция данных

Абстракцию данных можно рассматривать как процесс организации различных фрагментов информации в единые логические единицы (возможно, нерархически), придавая ей при этом некоторую концептуально осмысленную форму. Каждая информационная единица должна быть легко доступна в программе. В идеальном случае все детали реализации такой структуры должны быть невидимы пользователю этой структуры. Самое главное в этом процессе – дать программисту возможность использовать информацию, не думая о деталях ее действительного представления.

Обсудим один из способов реализации этого принципа на Прологе. Рассмотрим снова пример с семьей из предыдущего раздела. Каждая семья – это набор некоторых фрагментов информации. Все эти фрагменты

объединены в естественные информационные единицы, такие, как «члены семьи» или «семья», и с ними можно обращаться как с едиными объектами. Предположим опять, что информация о семье структурирована так же, как на рис. 4.1. Определим теперь некоторые отношения, с помощью которых пользователь может получать доступ к конкретным компонентам семьи, не зная деталей рис. 4.1. Такие отношения можно назвать *селекторами*, поскольку они позволяют выбирать конкретные компоненты. Имя такого отношения-селектора будет совпадать с именем компоненты, которую нужно выбрать. Отношение будет иметь два аргумента: первый – объект, который содержит компоненту, и второй – саму компоненту:

```
отношение_селектор(Объект, Выбранная_компонента)
```

Вот несколько селекторов для структуры семья:

```
муж( семья( Муж, _, _), Муж).
жена( семья( _, Жена, _), Жена).
дети(семья(_, _, СписокДетей), СписокДетей).
```

Можно также создать селекторы для отдельных детей семьи:

```
первыйребенок( Семья, Первый) :-
    дети( Семья, [Первый | _]).
второйребенок( Семья, Второй) :-
    дети( Семья, [_, Второй | _]).
...
```

Можно обобщить этот селектор для выбора N-го ребенка:

```
nребенок( N, Семья, Ребенок) :-
    дети( Семья, СписокДетей),
    n_элемент( N, СписокДетей, Ребенок)
                              % N-й элемент списка
```

Другим интересным объектом является «член семьи». Вот некоторые связанные с ним селекторы, соответствующие рис. 4.1:

```
имя( членсемьи( Имя, _, _, _), Имя).
фамилия( членсемьи( _, Фамилия, _, _), Фамилия).
датарождения( членсемьи( _, _, Дата), Дата).
```

Какие преимущества мы можем получить от использования отношений-селекторов? Определив их, мы можем теперь забыть о конкретном виде структуры представления информации. Для пополнения и обработки этой информации нужно знать только имена отношений-селекторов и в оставшейся части программы пользоваться только ими. В случае, если информация представлена сложной структурой, это легче, чем каждый раз обращаться к ней в явном виде. В частности, в нашем примере с семьей пользователь не обязан знать, что дети представлены в виде списка. Например, предположим, мы хотим сказать, что Том Фокс и Джим Фокс принадлежат к одной семье и что Джим – второй ребенок Тома. Используя приведенные выше отношения -селекторы, мы можем определить двух человек, назовем их **Человек1** и **Человек2**, и семью. Следующий список целей приводит к желаемому результату:

```
имя( Человек1, том), фамилия( Человек1, фокс),
                                % Человек1 - Том Фокс
имя( Человек2, джим), фамилия( Человек1, фокс),
                                % Человек2 - Джим Фокс
муж( Семья, Человек1),
второйребенок( Семья, Человек2)
```

Использование отношений-селекторов облегчает также и последующую модификацию программ. Представьте себе, что мы захотели повысить эффективность программы, изменив представление информации. Все, что нужно сделать для этого, – изменить определения отношений-селекторов, и вся остальная программа без изменений будет работать с этим новым представлением.

Упражнение

4.3. Завершите определение отношения ***n*ребенок**, определив отношение

***n*_элемент(N, Список, X)**

которое выполняется , если X является N-м элементом списка Список.

4.3. Моделирование недетерминированного автомата

Данное упражнение показывает, как абстрактную математическую конструкцию можно представить на Прологе. Кроме того, программа, которая получится, окажется значительно более гибкой, чем предполагалось вначале.

Недетерминированный конечный автомат – это абстрактная машина, которая читает символы из входной цепочки и решает, *допустить* или *отвергнуть* эту цепочку. Автомат имеет несколько *состояний* и всегда находится в одном из них. Он может изменить состояние, перейдя из одного состояния в другое. Внутреннюю структуру такого автомата можно представить графом переходов, как показано на рис. 4.3. В этом примере S_1, S_2, S_3 и S_4 – *состояния* автомата. Стартовав из начального состояния (в нашем примере это S_1), автомат переходит из состояния в состояние по мере чтения входной цепочки. Переход зависит от текущего входного символа, как указывают метки на дугах графа переходов.

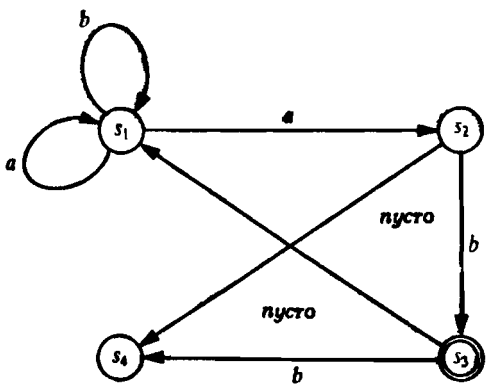


Рис. 4.3. Пример недетерминированного конечного автомата.

Переход выполняется всякий раз при чтении входного символа. Заметим, что переходы могут быть недетерминированными. На рис. 4.3 видно, что если автомат находится в состоянии S_1 и текущий входной символ равен a, то переход может осуществиться как в S_1, так и в S_2. Некоторые дуги помечены меткой *пусто*, обозначающей «пустой символ». Эти дуги соответствуют «спонтанным переходам» автомата. Такой переход называется *спонтанным*, потому что он выполняется без чтения входной цепочки. Наблюдатель, рассматривающий автомат как черный ящик, не сможет обнаружить, что произошел какой-либо переход.

Состояние S_3 обведено двойной линией, это означает, что S_3 – *конечное состояние*. Про автомат говорят, что он *допускает* входную цепочку, если в графе переходов существует путь, такой, что:

(1) он начинается в начальном состоянии,
(2) он оканчивается в конечном состоянии, и
(3) метки дуг, образующих этот путь, соответствуют полной входной цепочке.

Решать, какой из возможных переходов делать в каждый момент времени – исключительно внутреннее дело автомата. В частности, автомат сам решает, делать ли спонтанный переход, если он возможен в

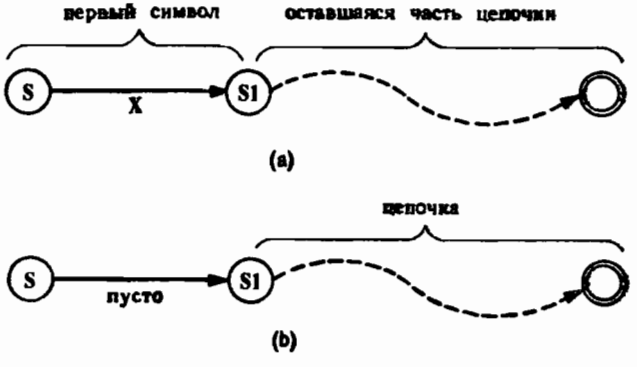


Рис. 4.4. Допущение цепочки: (a) при чтении первого символа X; (b) при совершении спонтанного перехода.

текущем состоянии. Однако абстрактные недетерминированные машины такого типа обладают волшебным свойством: если существует выбор, они всегда избирают «правильный» переход, т.е. переход, ведущий к допущению входной цепочки при наличии такого перехода. Автомат на рис. 4.3, например, допускает цепочки *ab* и *aabaab*, но отвергает цепочки *abb* и *abba*. Легко видеть, что этот автомат допускает любые цепочки, оканчивающиеся на *ab* и отвергает все остальные.

Некоторый автомат можно описать на Прологе при помощи трех отношений:

(1) Унарного отношения **конечное**, которое определяет конечное состояние автомата.

(2) Трехаргументного отношения **переход**, которое определяет переход из состояния в состояние, при этом

переход(S1, X, S2)

означает переход из состояния S1 в S2, если считан входной символ X.

(3) Бинарного отношения

спонтанный(S1, S2)

означающего, что возможен спонтанный переход из S1 в S2.

Для автомата, изображенного на рис. 4.3, эти отношения будут такими:

```
конечное( S3).

переход( S1, a, S1).
переход( S1, a, S2).
переход( S1, b, S1).
переход( S2, b, S3).
переход( S3, b, S4).

спонтанный( S2, S4).
спонтанный( S3, S1).
```

Представим входные цепочки в виде списков Пролога. Цепочка *aab* будет представлена как [a,a,b]. Модель автомата, получив его описание, будет обрабатывать заданную входную цепочку и решать, допус-

кать ее или нет. По определению, недетерминированный автомат допускает заданную цепочку, если (начав из начального состояния) после ее прочтения он способен оказаться в конечном состоянии. Модель программируется в виде бинарного отношения **допускается**, которое определяет приятие цепочки из данного состояния. Так

допускается(Состояние, Цепочка)

истинно, если автомат, начав из состояния **Состояние** как из начального, допускает цепочку **Цепочка**. Отношение **допускается** можно определить при помощи трех предложений. Они соответствуют следующим трем случаям:

(1) Пустая цепочка [] допускается из состояния S, если S – конечное состояние.

(2) Непустая цепочка допускается из состояния S, если после чтения первого ее символа автомат может перейти в состояние S1, и оставшаяся часть цепочки допускается из S1. Этот случай иллюстрируется на рис. 4.4(a).

(3) Цепочка допускается из состояния S, если автомат может сделать спонтанный переход из S в S1, а затем допустить (всю) входную цепочку из S1. Такой случай иллюстрируется на рис. 4.4(b).

Эти правила можно перевести на Пролог следующим образом:

```
допускается( S, []) :-
                % Допуск пустой цепочки
    конечное( S).

допускается( S, [X | Остальные]) :-
                % Допуск чтением первого символа
    переход( S, X, S1),
    допускается( S1, Остальные).

допускается( S, Цепочка) :-
                % Допуск выполнением спонтанного перехода
    спонтанный( S, S1),
    допускается( S1, Цепочка).
```

Спросить о том, допускается ли цепочка *aaab*, можно

так:

```
?- допускается( S1, [a,a,a,b]).

yes          (да)
```

Как мы уже видели, программы на Прологе часто оказываются способными решать более общие задачи, чем те, для которых они первоначально предназначались. В нашем случае мы можем спросить модель также о том, в каком состоянии должен находиться автомат в начале работы, чтобы он допустил цепочку *ab*:

```
?- допускается( S, [a,b]).

S = s1;
S = s3
```

Как ни странно, мы можем спросить также «Каковы все цепочки длины 3, допустимые из состояния s1?»

```
?- допускается( s1, [X1, X2, X3]).

X1 = a
X2 = a
X3 = b;

X1 = b
X2 = a
X3 = b;

no           (нет)
```

Если мы предпочитаем, чтобы допустимые цепочки выдавались в виде списков, тогда наш вопрос следует сформулировать так:

```
?- Цепочка = [_,_,_], допускается( s1, Цепочка).

Цепочка = [a,a,b];
Цепочка = [b,a,b];

no           (нет)
```

Можно проделать и еще некоторые эксперименты, например спросить: «Из какого состояния автомат допустит цепочку длиной 7?»

Эксперименты могут включать в себя переделки структуры автомата, вносящие изменения в отношении

конечное, переход и спонтанный. В автомате, изображенном на рис. 4.3, отсутствуют циклические «спонтанные пути» (пути, состоящие только из спонтанных переходов). Если на рис. 4.3 добавить новый переход

```
спонтанный( s1, s3)
```

то получится «спонтанный цикл». Теперь наша модель может столкнуться с неприятностями. Например, вопрос

```
?- допускается( s1, [a]).
```

приведет к тому, что модель будет бесконечно переходить в состояние s1, все время надеясь отыскать какой-либо путь в конечное состояние.

Упражнения

4.4. Почему не могло возникнуть зацикливание модели исходного автомата на рис. 4.3, когда в его графе переходов не было «спонтанного цикла»?

4.5. Зацикливание при вычислении допускается можно предотвратить, например, таким способом: подсчитывать число переходов, сделанных к настоящему моменту. При этом модель должна будет искать пути только некоторой ограниченной длины. Модифицируйте так отношение допускается. Указание: добавьте третий аргумент – максимально допустимое число переходов:

```
допускается(Состояние,Цепочка,Макс_переходов)
```

4.4. Планирование поездки

В данном разделе мы создадим программу, которая дает советы по планированию воздушного путешествия. Эта программа будет довольно примитивным советчиком, тем не менее она сможет отвечать на не-

которые полезные вопросы, такие как:

- По каким дням недели есть прямые рейсы из Лондона в Любляну?
- Как в четверг можно добраться из Любляны в Эдинбург?
- Мне нужно посетить Милан, Любляну и Цюрих; вылетать нужно из Лондона во вторник и вернуться обратно в Лондон в пятницу. В какой последовательности мне следует посещать эти города, чтобы ни разу на протяжении поездки не пришлось совершать более одного перелета в день.

Центральной частью программы будет база данных, содержащая информацию о рейсах. Эта информация будет представлена в виде трехаргументного отношения:

расписание(Пункт1, Пункт2, Список_рейсов)

где **Список_рейсов** – это список, состоящий из структурированных объектов вида:

Время_отправления/Время_прибытия/Номер_рейса
/Список_дней_вылета

Список_дней_вылета – это либо список дней недели, либо атом «ежедневно». Одно из предложений, входящих в **расписание** могло бы быть, например, таким:

```
расписание( лондон,эдинбург,
          [ 9:40/10:50/ba4733/ежедневно,
            19:40/20:50/ba4833/[пн,вт,ср,чт,пт,сб]] ).
```

Время представлено в виде структурированных объектов, состоящих из двух компонент – часов и минут, объединенных оператором «:».

Главная задача состоит в отыскании точных маршрутов между двумя заданными городами в определенные дни недели. Ее решение мы будем программировать в виде четырехаргументного отношения:

маршрут(Пункт1, Пункт2, День, Маршрут)

Здесь **Маршрут** – это последовательность перелетов, удовлетворяющих следующим критериям:

(1) начальная точка маршрута находится в **Пункт1**;
(2) конечная точка – в **Пункт2**;
(3) все перелеты совершаются в один и тот же

день недели – **День**;

(4) все перелеты, входящие в **Маршрут**, содержатся в определении отношения **расписание**;

(5) остается достаточно времени для пересадки с рейса на рейс.

Маршрут представляется в виде списка структурированных объектов вида

Откуда – Куда : Номер_рейса : Время_отправления

Мы еще будем пользоваться следующими вспомогательными предикатами:

(1) **рейс(Пункт1, Пункт2, День, N_рейса, Вр_отпр, Вр_приб)**

Здесь сказано, что существует рейс **N_рейса** между **Пункт1** и **Пункт2** в день недели **День** с указанными временами отправления и прибытия.

(2) **вр_отпр(Маршрут, Время)**

Время – это время отправления по маршруту **Маршрут**.

(3) **пересадка(Время1, Время2)**

Между **Время1** и **Время2** должен существовать промежуток не менее 40 минут для пересадки с одного рейса на другой.

Задача нахождения маршрута напоминает моделирование недетерминированного автомата из предыдущего раздела:

- Состояния автомата соответствуют городам.
- Переход из состояния в состояние соответствует перелету из одного города в другой.
- Отношение **переход** автомата соответствует отношению **расписание**.
- Модель автомата находит путь в графе переходов между исходным и конечным состояниями; планировщик поездки находит маршрут между начальным и конечным пунктами поездки.

Неудивительно поэтому, что отношение **маршрут** можно определить аналогично отношению **допускает**, с той разницей, что теперь нет «спонтанных переходов».

Существуют два случая:

(1) Прямой рейс: если существует прямой рейс между пунктами Пункт1 и Пункт2, то весь маршрут состоит только из одного перелета:

```
маршрут(Пункт1,Пункт2,День,[Пункт1-Пункт2:Np:Отпр]):
    рейс( Пункт1, Пункт2, День, Np, Отпр, Приб).
```

(2) Маршрут с пересадками: маршрут между пунктами P1 и P2 состоит из первого перелета из P1 в некоторый промежуточный пункт P3 и маршрута между P3 и P2. Кроме того, между окончанием первого перелета и отправлением во второй необходимо оставить достаточно времени для пересадки.

```
маршрут(P1,P2,День,[P1-P3:Np1:Отпр1|Маршрут]) :-
    маршрут( P3, P2, День, Маршрут ),
    рейс( P1, P3, День, Np1, Отпр1, Приб1),
    вр_отпр( Маршрут, Отпр2),
    пересадка( Приб1, Отпр2).
```

Вспомогательные отношения рейс, пересадка и вр_отпр запрограммировать легко; мы включили их в полный текст программы планировщика поездки на рис. 4.5. Там же приводится и пример базы данных - расписания.

Наш планировщик исключительно прост и может рассматривать путн, очевидно ведущие в никуда. Тем не менее его оказывается вполне достаточно, если база данных о рейсах самолетов невелика. Для больших баз данных потребовалось бы разработать более интеллектуальный планировщик, который мог бы справиться с большим количеством путей, участвующих в переборе при нахожденни нужного пути.

Вот некоторые примеры вопросов к планировщику:

- По каким дням недели существуют прямые рейсы из Лондона в Любляну?

```
?- рейс( лондон, любляна, День, _, _, _).

День = пт;
День = сб;

no          (нет)
```

```
% ПЛАНИРОВЩИК ВОЗДУШНЫХ МАРШРУТОВ

:- op( 50, xfy, :).

рейс(Пункт1,Пункт2,День,Np,ВрОтпр,ВрПриб) :-
   расписание( Пункт1, Пункт2, СписРейсов),
   принадлежит(ВрОтпр/ВрПриб/Np/СписДней,СписРейсов),
   день_выл( День, СписДней).

принадлежит( X, [X | L] ).

принадлежит( X, [Y | L] ) :-
    принадлежит( X, L ).

день_выл( День, СписДней) :-
    принадлежит( День, СписДней).

день_выл( День, ежедневно) :-
    принадлежит( День, [пн,вт,ср,чт,пт,сб,вс] ).

маршрут( P1, P2, День, [P1-P2 : Np : ВрОтпр] ) :-
                                % прямой рейс
    рейс( P1, P2, День, Np, ВрОтпр, _).

маршрут(P1,P2,День, [P1-P3:Np1:Отпр1|Маршрут]) :-
                                % маршрут с пересадками
    маршрут( P3, P2, День, Маршрут ),
    рейс( P1, P3, День, Np1, Отпр1, Приб1),
    вр_отпр( Маршрут, Отпр2),
    пересадка( Приб1, Отпр2).

вр_отпр( [P1-P2 : Np : Отпр | _], Отпр).

пересадка( Часы1 : Минуты1, Часы2 : Минуты2) :-
    60 * (Часы2-Часы1) + Минуты2-Минуты1 >= 40

% БАЗА ДАННЫХ О РЕЙСАХ САМОЛЕТОВ

расписание( эдинбург, лондон,
   [ 9:40 / 10:50 / ba4733 / ежедневно,
     13:40 / 14:50 / ba4773 / ежедневно,
     19:40/20:50/ba4833/[пн,вт,ср,чт,пт,вс] ] ).

расписание( лондон, эдинбург,
   [ 9:40 / 10:50 / ba4732 / ежедневно,
     11:40 / 12:50 / ba4752 / ежедневно,
     18:40/19:50/ba4822/[пн,вт,ср,чт,пт] ] ).

расписание( лондон, любляна,
   [13:20 / 16:20 / ju201 / [пт],
     13:20 / 16:20 / ju213 / [вс] ] ).
```

```
расписание( лондон, цюрих,
   [ 9:10 / 11:45 / ba614 / ежедневно,
     14:45 / 17:20 / sr805 / ежедневно ] ).

расписание( лондон, милан,
   [ 8:30 / 11:20 / ba510 / ежедневно,
     11:00 / 13:50 / az459 / ежедневно ] ).

расписание( любляна, цюрих,
   [11:30 / 12:40 / ju322 / [вт,чт] ] ).

расписание( любляна, лондон,
   [11:10 / 12:20 / yu200 / [пт],
     11:25 / 12:20 / yu212 / [вс] ] ).

расписание( милан, лондон,
   [ 9:10 / 10:00 / az458 / ежедневно,
     12:20 / 13:10 / ba511 /  ежедневно  ] ).

расписание( милан, цюрих,
   [ 9:25 / 10:15 / sr621 / ежедневно,
     12:45 / 13:35 / sr623 / ежедневно ] ).

расписание( цюрих, любляна,
   [13:30 / 14:40 / yu323 / [вт,чт] ] ).

расписание( цюрих, лондон,
   9:00 / 9:40 / ba613 /
   [пн,вт,ср,чт,пт,сб],
   16:10/16:55/sr806/[пн,вт,ср,чт,пт,сб] ] ).

расписание( цюрих, милан,
   [ 7:55 / 8:45 / sr620 / ежедневно ] ).
```

Рис. 4.5. Планировщик воздушных маршрутов и база данных о рейсах самолетов.

- Как мне добраться из Любляны в Эдинбург в четверг?

  ```
  ?- маршрут( любляна, эдинбург, чт, R).

  R=[любляна-цюрих:yu322:11:30,цюрих-лондон:
  sr806 : 16:10,
  лондон-эдинбург : ba4822 : 18:40 ]
  ```

- Как мне посетить Милан, Любляну и Цюрих, вылетев из Лондона во вторник и вернувшись в него в пятницу, совершая в день не более одного перелета? Этот вопрос сложнее, чем

предыдущие. Его можно сформулировать, использовав отношение **перестановка**, запрограммированное в гл. 3. Мы попросим найти такую перестановку городов Милан, Любляна и Цюрих, чтобы соответствующие перелеты можно было осуществить в несколько последовательных дней недели:

```
?- перестановка( [милан,любляна,цюрих],
                 [Город1,Город2,Город3] ),

   рейс(лондон,Город1,вт,Nр1,Отпр1,Прпб1),
   рейс(Город1,Город2,ср,Nр2,Отпр2,Приб2),
   рейс(Город2,Город3,чт,Nр3,Отпр3,Приб3),
   рейс(Город3,лондон,пт,Nр4,Отпр4,Приб4).

   Город1 = милан
   Город2 = цюрих
   Город3 = любляна

   Nр1 = ba510
   Отпр1 = 8:30
   Приб1 = 11:20

   Nр2 = sr621
   Отпр2 = 9:25
   Приб2 = 10:15

   Nр3 = yu323
   Отпр3 = 13:30
   Приб3 = 14:40

   Nр4 = yu200
   Отпр4 = 11:10
   Приб4 = 12:20
```

4.5. Задача о восьми ферзях

Эта задача состоит в отыскании такой расстановки восьми ферзей на пустой шахматной доске, в которой ни один из ферзей не находится под боем другого. Решение мы запрограммируем в виде унарного отношения:

```
решение( Поз)
```

которое истинно тогда и только тогда, когда Поз изображает позицию, в которой восемь ферзей не бьют друг друга. Будет интересно сравнить различные идеи, лежащие в основе программирования этой задачи. Поэтому мы приведем три программы, основанные на слегка различающихся ее представлениях.

4.5.1. Программа 1

Вначале нужно выбрать способ представления позиции на доске. Один из наиболее естественных способов – представить позицию в виде списка из восьми элементов, каждый из которых соответствует одному из ферзей. Каждый такой элемент будет описывать то поле доски, на котором стоит соответствующий ферзь. Далее, каждое поле доски можно описать с помощью пары координат (X и Y), где каждая координата – целое число от 1 до 8. В программе мы будем записывать такую пару в виде

X / Y

где оператор «/» обозначает, конечно, не деление, а служит лишь для объединения координат поля в один элемент списка. На рис. 4.6 показано одно из решений задачи о восьми ферзях и его запись в виде списка.

После того, как мы выбрали такое представление, задача свелась к нахождению списка вида:

[X1/Y1, X2/Y2, X3/Y3, X4/Y4, X5/Y5, X6/Y6, X7/Y7, X8/Y8]

удовлетворяющего требованию отсутствия нападений. Наша процедура решение должна будет найти подходящую конкретизацию переменных X1, Y1, X2, Y2, ..., X8, Y8. Поскольку мы знаем, что все ферзи должны находиться на разных вертикалях во избежание нападений по вертикальным линиям, мы можем сразу же ограничить перебор, чтобы облегчить поиск решения. Можно поэтому сразу зафиксировать X-координаты так, чтобы список, изображающий решение, удовлетворял следующему, более конкретному шаблону:

[1/Y1, 2/Y2, 3/Y3, 4/Y4, 5/Y5, 6/Y6, 7/Y7, 8/Y8]

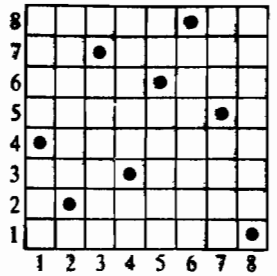


Рис. 4.6 Решение задачи о восьми ферзях. Эта позиция может быть представлена в виде списка [1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1].

Нас интересует решение для доски размером 8x8. Однако, как это часто бывает в программировании, ключ к решению легче найти, рассмотрев более общую постановку задачи. Как это ни парадоксально, но часто оказывается, что решение более общей задачи легче сформулировать, чем решение более частной, исходной задачи; после этого исходная задача решается просто как частный случай общей задачи.

Основная часть работы при таком подходе ложится на нахождение правильного обобщения исходной задачи. В нашем случае хорошей является идея обобщить задачу в отношении количества ферзей (количества вертикалей в списке), разрешив количеству ферзей принимать любое значение, включая нуль. Тогда отношение решение можно сформулировать, рассмотрев два случая:

Случай 1. Список ферзей пуст. Пустой список без сомнения является решением, поскольку нападений в этом случае нет.

Случай 2. Список ферзей не пуст. Тогда он выглядит так:

[X/Y | Остальные]

В случае 2 первый ферзь находится на поле **X / Y**, а остальные – на полях, указанных в списке **Остальные**. Если мы хотим, чтобы это было решением,

то должны выполняться следующие условия:

(1) Ферзи, перечисленные в списке **Остальные**, не должны бить друг друга; т. е. список **Остальные** сам должен быть решением.

(2) X и Y должны быть целыми числами от 1 до 8.

(3) Ферзь, стоящий на поле **X / Y**, не должен бить ни одного ферзя из списка **Остальные**.

Чтобы запрограммировать первое условие, можно воспользоваться самим отношением **решение**. Второе условие можно сформулировать так: Y должен принадлежать списку целых чисел от 1 до 8, т. е. **[1,2,3,4,5,6,7,8]**. С другой стороны, о координате X можно не беспокоиться, поскольку список-решение должен соответствовать шаблону, у которого X-координаты уже определены. Поэтому X гарантированно получит правильное значение от 1 до 8. Третье условие можно обеспечить с помощью нового отношения **небьет**. Все это можно записать на Прологе так:

```
решение( [X/Y | Остальные] ) :-
    решение( Остальные),
    принадлежит( Y, [1,2,3,4,5,6,7,8] ),
    небьет( X/Y, Остальные).
```

Осталось определить отношение **небьет**:

```
небьет( Ф, Фспис)
```

И снова его описание можно разбить на два случая:

(1) Если список **Фспис** пуст, то отношение, конечно, выполнено, потому что некого бить (нет ферзя, на которого можно было бы напасть).

(2) Если **Фспис** не пуст, то он имеет форму
[Ф1 | Фспис1]
и должны выполняться два условия:

(a) ферзь на поле **Ф** не должен бить ферзя на поле **Ф1** и

(b) ферзь на поле **Ф** не должен бить ни одного ферзя из списка **Фспис1**.

Выразить требование, чтобы ферзь, находящийся на некотором поле, не бил другое поле, довольно прос-

то: эти поля не должны находиться на одной и той же горизонтали, вертикали или диагонали. Наш шаблон решения гарантирует, что все ферзи находятся на разных вертикалях, поэтому остается только обеспечить, чтобы

- Y-координаты ферзей были различны и
- ферзи не находились на одной диагонали, т.е. расстояние между полями по направленню X не должно равняться расстоянню между нимн по Y.

На рис. 4.7 приведеи полный текст программы. Чтобы облегчнть ее нспользованне, необходимо добавить список-шаблон. Это можно сделать в запросе на генерацию решений. Итак:

```
?- шаблон(S), решение( S).
```

```
решение( [] ).
решение( [X/Y | Остальные ] ) :-
            % Первый ферзь на поле X/Y,
            % остальные ферзн на полях из списка Остальные
    решение( Остальные),
    принадлежит( Y, [1,2,3,4,5,6,7,8] ),
    небьет( X/Y | Остальные).
            % Первый ферзь не бьет остальных
небьет( _, []).                 % Некого бить
небьет( X/Y, [X1/Y1 | Остальные] ) :-
      Y =\= Y1,                 % Разные Y-координаты
      Y1-Y =\= X1-X,            % Разные диагонали
      Y1-Y =\= X-X1,
      небьет( X/Y, Остальные).
принадлежит( X, [X | L] ).
принадлежит( X, [Y | L] ) :-
      принадлежит( X, L).
% Шаблон решения
шаблон([1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8]).
```

Рис. 4.7. Программа1 дли задачн о восьмн ферзях.

Система будет генерировать решения в таком виде:

```
S = [1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1];
S = [1/5, 2/2, 3/4, 4/7, 5/3, 6/8, 7/6, 8/1];
S = [1/3, 2/5, 3/2, 4/8, 5/6, 6/4, 7/7, 8/1];
...
```

Упражнение

4.6. При поиске решения программа, приведенная на рис. 4.7, проверяет различные значения Y-координат ферзей. В каком месте программы задается порядок перебора альтернативных вариантов? Как можно без труда модифицировать программу, чтобы этот порядок изменился? Поэкспериментируйте с разными порядками, имея в виду выяснить, как порядок перебора альтернатив влияет на эффективность программы.

4.5.2. Программа 2

В соответствии с принятым в программе 1 представлением доски каждое решение имело вид

```
[1/Y1, 2/Y2, 3/Y3, ..., 8/Y8]
```

так как ферзи расставлялись попросту в последовательных вертикалях. Никакая информация не была бы потеряна, если бы X-координаты были пропущены. Поэтому можно применить более экономное представление позиции на доске, оставив в нем только Y-координаты ферзей:

```
[Y1, Y2, Y3, ..., Y8]
```

Чтобы не было нападений по горизонтали, никакие два ферзя не должны занимать одну и ту же горизонталь. Это требование накладывает ограничение на Y-координаты: ферзи должны занимать все горизонтали с 1-й по 8-ю. Остается только выбрать *порядок* следования этих восьми номеров. Каждое решение представляет собой поэтому одну из перестановок списка:

[1, 2, 3, 4, 5, 6, 7, 8]

Такая перестановка S является решением, если каждый ферзь в ней не находится под боем (список S – «безопасный»). Поэтому мы можем написать:

```
решение( S) :-
    перестановка( [1,2,3,4,5,6,7,8], S),
    безопасный( S).
```

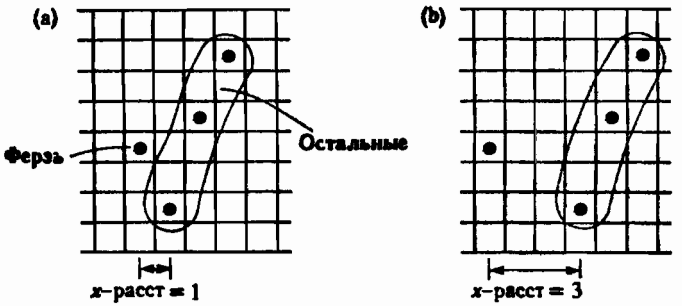


Рис. 4.8. (a) Расстояние по X между **Ферзь** и **Остальные** равно 1.
(b) Расстояние по X между **Ферзь** и **Остальные** равно 3.

Отношение **перестановка** мы уже определили в гл. 3, а вот отношение **безопасный** нужно еще определить. Это определение можно разбить на два случая:

(1) S – пустой список. Тогда он, конечно, безопасный, ведь нападать не на кого.

(2) S – непустой список вида [**Ферзь**|**Остальные**]. Он безопасный, если список **Остальные** – безопасный и **Ферзь** не бьет ни одного ферзя из списка **Остальные**.

На Прологе это выглядит так:

```
безопасный( []).

безопасный( [Ферзь | Остальные ] :-
    безопасный( Остальные),
    небьет(Ферзь | Остальные).
```

В этой программе отношение **небьет** более хитрое.

```
решение( Ферзи) :-
    перестановка( [1,2,3,4,5,6,7,8], Ферзи),
    безопасный( Ферзи).

перестановка( [], []).

перестановка( [Голова | Хвост], СписПер) :-
    перестановка( Хвост, ХвостПер),
    удалить( Голова, СпнсПер, ХвостПер).
                % Вставка головы в переставленный хвост

удалнть( А, [А | Список).

удалить( А, [В | Спнсок], [В, Список1] ) :-
    удалить( А, Список, Список1).

безопасный( []).

безопасный([Ферзь | Остальные]) :-
    безопасный( Остальные),
    небьет(Ферзь, Остальные, 1).

небьет( _, [], _).

небьет( Y, [Y1 | СписY], РасстХ) :-
  Y1-Y =\= РасстХ,
  Y-Y1 =\= РасстХ,
  Расст1 is РасстХ + 1,
  небьет( Y, СписY, Расст1).
```

Рис. 4.9. Программа 2 для задачн о восьин ферзях.

Трудность состоит в том, что расположенне ферзей определяется только нх Y-координатами, а X-координнаты в представленнн позиции не прнсутствуют в явном внде. Этой трудности можно избежать путем небольшого обобщения отношения **небьет**, как это показано на рис. 4.8.
Предполагается, что цель

небьет(Ферзь, Остальные)

обеспечнвает отсутствие нападенний ферзя **Ферзь** на поля списка **Остальные** в случае, когда расстояние по X между **Ферзь** и **Остальные** равно 1. Остается рассмотреть более общий случай пронзвольного расстояння. Для этого мы добавим его в отношение **небьет** в качестве третьего аргумента:

небьет(Ферзь, Остальные, РасстХ)

Соответственно и цель **небьет** в отношении **безопасный** должна быть изменена на

небьет(Ферзь, Остальные, 1)

Теперь отношение **небьет** может быть сформулировано в соответствии с двумя случаями, в зависимости от списка **Остальные**: если он пуст, то бить некого и, естественно, нет нападений; если же он не пуст, то **Ферзь** не должен бить первого ферзя из списка **Остальные** (который находится от ферзя **Ферзь** на расстоянии **РасстХ** вертикалей), а также ферзей из хвоста списка **Остальные**, находящихся от него на расстоянии **РасстХ + 1**. Эти соображения приводят к программе, изображенной на рис. 4.9.

4.5.3. Программа 3

Наша третья программа для задачи о восьми ферзях опирается на следующие соображения. Каждый ферзь должен быть размещен на некотором поле, т. е. на некоторой вертикали, некоторой горизонтали, а также на пересечении каких-нибудь двух диагоналей. Для того, чтобы была обеспечена безопасность каждого ферзя, все они должны располагаться в разных вертикалях, разных горизонталях и в разных диагоналях (как идущих сверху вниз, так и идущих снизу вверх). Естественно поэтому рассмотреть более богатую систему представления с четырьмя координатами:

x	вертикали
y	горизонтали
u	диагонали, идущие снизу вверх
v	диагонали, идущие сверху вниз

Эти координаты не являются независимыми: при заданных x и y, u и v определяются однозначно (пример на рис. 4.10). Например,

$$u = x - y$$
$$v = x + y$$

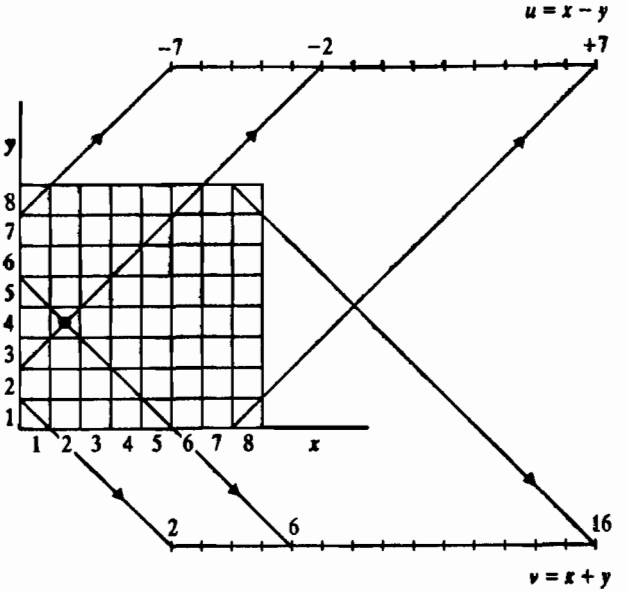


Рис. 4.10. Связь между вертикалями, горизонталями и диагоналями. Помеченное поле имеет следующие координаты: $x = 2$, $y = 4$, $u = 2 - 4 = -2$, $v = 2 + 4 = 6$.

Области изменения всех четырех координат таковы:

```
Dx = [1,2,3,4,5,6,7,8]
Dy = [1,2,3,4,5,6,7,8]

Du = [-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7]
Dv = [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
```

Задачу о восьми ферзях теперь можно сформулировать следующим образом: выбрать восемь четверок (X, Y, U, V), входящих в области изменения (X в Dx, Y в Dy и т.д.), так, чтобы ни одни их элемент не выбирался дважды из одной области. Разумеется, выбор X и Y определяет выбор U и V. Решение при такой постановке задачи может быть вкратце таким: при заданных 4-х областях изменения выбрать позицию для первого ферзя, вычеркнуть соответствующие элементы из 4-х областей изменения, а затем ис-

пользовать оставшиеся элементы этих областей для размещения остальных ферзей. Программа, основанная на таком подходе, показана на рис. 4.11. Позиция на доске снова представляется списком Y-координат. Ключевым отношением в этой программе является отношение

реш(СписY, Dx, Dy, Du, Dv)

которое конкретизирует Y-координаты (в СписY) ферзей, считая, что они размещены в последовательных вертикалях, взятых из Dx. Все Y-координаты и соответствующие координаты U и V берутся из списков Dy, Du и Dv. Главную процедуру решение можно запустить вопросом

?- решение(S)

Это вызовет запуск реш с полными областями изменения координат, что соответствует пространству

```
решение( СписY) :-
    реш( СписY,          % Y-координаты ферзей
         [1,2,3,4,5,6,7,8],
                         % Область изменения Y-координат
         [1,2,3,4,5,6,7,8],
                         % Область изменения X-координат
         [-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7],
                         % Диагонали, идущие снизу вверх
         [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16] ).
                         % Диагонали, идущие сверху вниз
реш( [], [], Dy, Du, Dv).

реш( [Y |СписY], [X | Dx1], Dy, Du, Dv) :-
    удалить( Y, Dy, Dy1),      % Выбор Y-координаты
    U is X-Y                   % Соответствующая диагональ вверх
    удалить( U, Du, Du1),      % Ее удаление
    V is X+Y                   % Соответствующая диагональ вниз
    удалить( V, Dv, Dv1),      % Ее удаление
    реш( СписY, Dx1, Dy1, Du1 Dv1).
                               % Выбор из оставшихся значений

удалить( A, [A | Список], Список).

удалить(A, [B | Список ], [B | Список1 ] ) :-
    удалить( A, Список, Список1).
```

Рис. 4.11. Программа 3 для задачи о восьми ферзях.

задачи о восьми ферзях.

Процедура реш универсальна в том смысле, что ее можно использовать для решения задачи об N ферзях (на доске размером N x N). Нужно только правильно задать области Dx, Dy и т.д.

Удобно автоматизировать получение этих областей. Для этого нам потребуется процедура

```
генератор( N1, N2, Список)
```

которая для двух заданных целых чисел N1 и N2 порождает список

```
Список = [N1, N1 + 1, N1 + 2, ..., N2 - 1, N2]
```

Вот она:

```
генератор( N, N, [N]).

генератор( N1, N2, [N1 | Список]) :-
        N1 < N2,
        M is N1 + 1,
        генератор( M, N2, Список).
```

Главную процедуру решение нужно соответствующим образом обобщить:

```
решение( N, S)
```

где N – это размер доски, а S – решение, представляемое в виде списка Y-координат N ферзей. Вот обобщенное отношение решение:

```
решение( N, S) :-
    генератор( 1, N, Dxy),
    Nu1 is 1 - N, Nu2 is N - 1,
    генератор( Nu1, Nu2, Du),
    Nv2 is N + N,
    генератор( 2, Nv2, Dv),
    реш( S, Dxy, Dxy, Du, Dv).
```

Например, решение задачи о 12 ферзях будет получено с помощью:

```
?- решение( 12, S).

S = [1,3,5,8,10,12,6,11,2,7,9,4]
```

4.5.4. Заключительные замечания

Три решения задачи о восьми ферзях показывают, как к одной и той же задаче можно применять различные подходы. Мы варьировали также и представление данных. В одних случаях это представление было более экономным, в других – более наглядным и, до некоторой степени, избыточным. К недостаткам более экономного представления можно отнести то, что какая-то информация всякий раз, когда она требовалась, должна была перевычисляться.

В некоторых случаях основным шагом к решению было обобщение задачи. Как ни парадоксально, но при рассмотрении более общей задачи решение оказывалось проще сформулировать. Принцип такого обобщения – стандартный прием программирования, и его можно часто применять.

Из всех трех программ третья лучше всего показывает, как подходить к общей задаче построения структуры из заданного множества элементов при наличии ограничений.

Возникает естественный вопрос: «Какая из трех программ наиболее эффективна?» В этом отношении программа 2 значительно хуже двух других, а эти последние – одинаковы. Причина в том, что основанная на перестановках программа 2 строит все перестановки, тогда как две другие программы способны отбросить плохую перестановку не дожидаясь, пока она будет полностью построена. Программа 3 наиболее эффективна. Она избегает некоторых арифметических вычислений, результаты которых уже сразу заложены в избыточное представление доски, используемое этой программой.

Упражнение

4.7. Пусть поля доски представлены парами своих координат в виде X/Y, где как X, так и Y принимают значения от 1 до 8.

(а) Определите отношение **ходконя(Поле1, Поле2)**, соответствующее ходу коня на шах-

матиой доске. Считайте, что Поле1 имеет всегда конкретизированные координаты, в то время, как координаты поля Поле2 могут и ие быть конкретизироваиы. Например:

```
?- ходконя( 1/1, S).

S = 3/2;
S = 2/3;

no          (нет)
```

(b) Определите отиошение **путьконя(Путь)**, где **Путь** – список полей, представляющих соответствующую правилам игры последовательность ходов коия ио пустой доске.

(c) Используя отношение **путькоия**, иапишите вопрос для нахождения любого пути, состоящего из 4-х ходов, и начинающегося с поля 2/1, а заканчивающегося иа противоположном крае доски (Y = 8). Этот путь должен еще проходить после второго хода через поле 5/4.

Резюме

Примеры, рассмотреиные в данном разделе, иллюстрируют иекоторые достоииства и характерные черты программироваиия иа Прологе:

- Базу даиных можио естественным образом представить в виде множества прологовских фактов.
- Такие механизмы Пролога, как вопросы и сопоставление, можно гибко использовать для получения ииформации из базы даиных. Кроме этого можио использовать вспомогательные процедуры-утилиты, еще больше облегчающие взаимодействие с конкретной базой данных.
- *Абстракцию данных* можно рассматривать как метод программирования, который облегчает работу со сложными структурами данных и виосит большую ясность и наглядность в програмы. В Прологе легко соблюдать осиовиые принципы абстракции даиных.

- Часто легко можно осуществить перевод абстрактных математических конструкций, таких как автоматы, на язык определений Пролога, готовых к выполнению.
- Как это было в случае восьми ферзей, многие задачи допускают различные подходы, связанные с разными представлениями этих задач. Часто внесение избыточности в представление экономит вычисления. Происходит как бы проигрыш в рабочем пространстве, но выигрыш во времени.
- Часто основным шагом на пути к решению оказывается обобщение задачи. Парадоксально, но рассмотрение более общей задачи позволяет облегчить формулировку решения.

5 УПРАВЛЕНИЕ ПЕРЕБОРОМ

Мы уже видели, что программист может управлять процессом вычислений по программе, располагая ее предложения и цели в том или ином порядке. В данной главе мы рассмотрим еще одно средство управления, получившее название «отсечение» (cut) и предназначенное для ограничения автоматического перебора.

5.1. Ограничение перебора

В процессе достижения цели пролог-система осуществляет автоматический перебор вариантов, делая возврат при неуспехе какого-либо из них. Такой перебор - полезный программный механизм, поскольку он освобождает пользователя от необходимости программировать его самому. С другой стороны, ничем не ограниченный перебор может стать источником

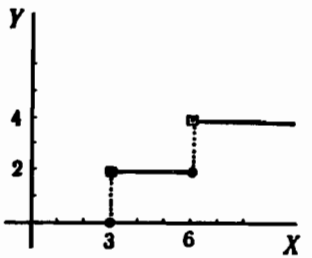


Рис. 5.1. Двухступенчатая функция.

неэффективности программы. Поэтому иногда требуется его ограничить или исключить вовсе. Для этого в Прологе предусмотрена конструкция «отсечение».

Давайте сначала рассмотрим простую программу, процесс вычислений по которой содержит ненужный перебор. Мы выделим те точки этого процесса, где перебор бесполезен и ведет к неэффективности.

Рассмотрим двухступенчатую функцию, изображенную на рис. 5.1. Связь между X и Y можно определить с помощью следующих трех правил:

Правило 1: если X < 3, то Y = 0

Правило 2: если 3 ≤ X и X < 6, то Y = 2

Правило 3: если 6 ≤ X, то Y = 4

На Прологе это можно выразить с помощью бинарного отношения

```
f( X, Y)
```

так:

```
f( X, 0) :- X < 3.                    % Правило 1
f( X, 2) :- 3 =< X, X < 6.            % Правило 2
f( X, 4) :- 6 =< X.                   % Правило 3
```

В этой программе предполагается, конечно, что к моменту начала вычисления f(X, Y) X уже конкретизирован каким-либо числом; это необходимо для выполнения операторов сравнения.

Мы проделаем с этой программой два эксперимента. Каждый из них обнаружит в ней свой источник неэффективности, и мы устраним оба этих источника по очереди, применив оператор отсечения.

5.1.1. Эксперимент 1

Проанализируем, что произойдет, если задать следующий вопрос:

```
?- f( 1, Y), 2 < Y.
```

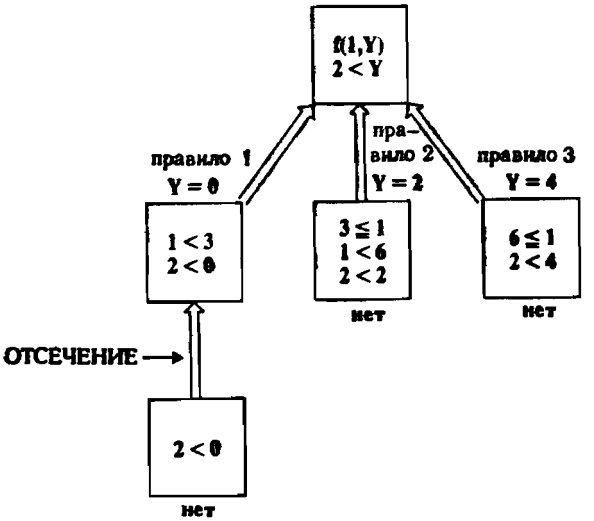


Рис. 5.2. В точке,помеченной словом "ОТСЕЧЕНИЕ", уже известно, что правила 2 и 3 должны потерпеть неудачу.

При вычислении первой цели f(1,Y) Y конкретизируется нулем. Поэтому вторая цель становится такой:

2 < 0

Она терпит неудачу, а поэтому и весь список целей также терпит неудачу. Это очевидно, однако перед тем как признать, что такому списку целей удовлетворить нельзя, пролог-система при помощи возвратов попытается проверить еще две бесполезные в данном случае альтернативы. Пошаговое описание процесса вычислений приводится на рис. 5.2.

Три правила, входящие в отношение f, являются взаимоисключающими, поэтому успех возможен самое большее в одном из них. Следовательно, мы (но не пролог-система) знаем, что, как только успех наступил в одном из них, нет смысла проверять остальные, поскольку они все равно обречены на неудачу. В примере, приведенном на рис. 5.2, о том, что в правиле 1 наступил успех, становится известно в точке, обозначенной словом «ОТСЕЧЕНИЕ». Для предотвращения бессмысленного перебора мы должны явно указать пролог-системе, что *не нужно*

осуществлять возврат из этой точки. Мы можем сделать это при помощи коиструкции отсечения. «Отсечение» записывается в виде символа '!', который вставляется между целями и играет роль некоторой псевдоцели. Вот наша программа, переписаниая с использованием отсечения:

```
f( X, 0) :- X < 3, !.
f( X, 2) :- 3 =< X, X < 6, !.
f( X, 4) :- 6 =< X.
```

Символ '!' предотвращает возврат из тех точек программы, в которых ои поставлеи. Если мы теперь спросим

```
?- f( 1, Y), 2 < Y.
```

то пролог–система породит левую ветвь дерева, изображениого на рис. 5.2. Эта ветвь потерпит неудачу иа цели 2 < 0. Система попытается сделать возврат, но вериуться она сможет не далее точки, помеченной в программе символом '!'. Альтернативные ветви, соответствующие правилу 2 и правнлу 3, порождены не будут.

Новая программа, снабжеиная отсечениями, во всех случаях более эффективна, чем первая версия, в которой они отсутствуют. Неудачные варианты новая программа распознает всегда быстрее, чем старая.

Вывод: добавив отсечеиия, мы повысили эффективиость. Если их теперь убрать, программа породит тот же результат, только на его получение она потратит скорее всего больше времени. Можно сказать, что в иашем случае после введения отсеченнй мы нзменили только процедуриый смысл программы, оставив при этом ее декларативный смысл в неприкосновеиности. В дальнейшем мы покажем, что использование отсечения может также затронуть и декларативный смысл программы.

5.1.2. Эксперимент 2

Проделаем теперь еще одни эксперимент со второй версней нашей программы. Предположим, мы задаем

вопрос:

```
?- f( 7, Y).
Y = 4
```

Проанализируем, что произошло. Перед тем, как был получен ответ, система пробовала применить все три правила. Эти попытки породили следующую последовательность целей:

Попытка применить правило 1:

7 < 3 терпит неудачу, происходит возврат и попытка применить правило 2 (точка отсечения достигнута не была)

Попытка применить правило 2:

3 ≤ 7 успех, но 7 < 6 терпит неудачу; возврат и попытка применить правило 3 (точка отсечения снова не достигнута)

Попытка применить правило 3:

6 ≤ 7 – успех

Приведенные этапы вычисления обнаруживают еще один источник неэффективности. В начале выясняется, что **X < 3** не является истиной (7 < 3 терпит неудачу). Следующая цель – 3 =< X (3 ≤ 7 – успех). Но нам известно, что, если первая проверка неуспешна, то вторая обязательно будет успешной, так как второе целевое утверждение является отрицанием первого. Следовательно, вторая проверка лишняя и соответствующую цель можно опустить. То же самое верно и для цели 6 =< X в правиле 3. Все эти соображения приводят к следующей, более экономной формулировке наших трех правил:

если X < 3, то Y = 0

иначе, если 3 ≤ X и X < 6, то Y = 2,

иначе Y = 4.

Теперь мы можем опустить в нашей программе те условия, которые обязательно выполняются при любом вычислении. Получается третья версия программы:

```
f( X, 0) :- X < 3, !.
f( X, 2) :- X < 6, !.
f( X, 4).
```

Эта программа дает тот же результат, что и исходная, но более эффективна, чем обе предыдущие. Однако, что будет, если мы *теперь* удалим отсечения? Программа станет такой:

```
f( X, 0) :- X < 3.
f( X, 2) :- X < 6.
f( X, 4).
```

Она может порождать различные решения, часть из которых неверны. Например:

```
?- f( 1, Y).
Y = 0;
Y = 2;
Y = 4;
no          (нет)
```

Важно заметить, что в последней версии, в отличие от предыдущей, отсечения затрагивают не только процедурное поведение, но изменяют также и декларативный смысл программы.

Более точный смысл механизма отсечений можно сформулировать следующим образом:

Назовем «целью-родителем» ту цель, которая сопоставилась с головой предложения, содержащего отсечение. Когда в качестве цели встречается отсечение, такая цель сразу же считается успешной и при этом заставляет систему принять те альтернативы, которые были выбраны с момента активизации цели-родителя до момента, когда встретилось отсечение. Все оставшиеся в этом промежутке (от цели-родителя до отсечения) альтернативы не рассматриваются.

Чтобы прояснить смысл этого определения, рассмотрим предложение вида

```
H :- B1, B2, ..., Bm, !, ..., Bn.
```

Будем считать, что это предложение активизировалось, когда некоторая цель G сопоставилась с H. Тогда G является целью-родителем. В момент, когда встретилось отсечение, успех уже наступил в целях B1, ..., Bm. При выполнении отсечения это (текущее) решение B1, ..., Bm «замораживается» и все возможные оставшиеся альтернативы больше не рассматриваются. Далее, цель G связывается теперь с этим предложением: любая попытка сопоставить G с головой какого-либо другого предложения пресекается.

Применим эти правила к следующему примеру:

```
C :- P, Q, R, !, S, T, U.
C :- V.
A :- B, C, D.
?- A.
```

Здесь A, B, C, D, P и т.д. имеют синтаксис термов. Отсечение повлияет на вычисление цели C следующим образом. Перебор будет возможен в списке целей P, Q, R; однако, как только точка отсечения будет достигнута, все альтернативные решения для этого списка изымаются из рассмотрения. Альтернативное предложение, входящее в C:

```
C :- V.
```

также не будет учитываться. Тем не менее перебор будет возможен в списке целей S, T, U. «Цель-родитель» предложения, содержащего отсечение, – это цель C в предложении

```
A :- B, C, D.
```

Поэтому отсечение повлияет только на цель C. С другой стороны, оно будет «невидимо» из цели A. Таким образом, автоматический перебор все равно будет происходить в списке целей B, C, D, вне зависимости от наличия отсечения в предложении, которое используется для достижения C.

5.2. Примеры, использующие отсечение

5.2.1. Вычисление максимума

Процедуру нахождения наибольшего из двух чисел можно запрограммировать в виде отношения

```
max( X, Y, Max)
```

где Max = X, если X больше или равно Y, и Max есть Y, если X меньше Y. Это соответствует двум таким предложениям:

```
max( X, Y, X) :- X >= Y.
max( X, Y, Y) :- X < Y.
```

Эти правила являются взаимно исключающими. Если выполняется первое, второе обязательно потерпит неудачу. Если неудачу терпит первое, второе обязательно должно выполниться. Поэтому возможна более экономная формулировка, использующая понятие «иначе»:

если X ≥ Y, то Max = X,

иначе Max = Y.

На Прологе это записывается при помощи отсечения:

```
max( X, Y, X) :- X >= Y, !.
max( X, Y, Y).
```

5.2.2. Процедура проверки принадлежности списку, дающая единственное решение

Для того, чтобы узнать, принадлежит ли X списку L, мы пользовались отношением

принадлежит(X, L)

Программа была следующей:

```
принадлежит( X, [X | L] ).
принадлежит( X, [Y | L] ) :- принадлежит( X, L).
```

Эта программа дает «недетерминированный» ответ: если X встречается в списке несколько раз, то будет найдено каждое его вхождение. Исправить этот недостаток не трудно: нужно только предотвратить дальнейший перебор сразу же после того, как будет найден первый X, а это произойдет, как только в первом предложении наступит успех. Измененная программа выглядит так:

```
принадлежит( X, [X | L] ) :- !.
принадлежит( X, [Y | L] ) :- принадлежит( X, L).
```

Эта программа породит только одно решение. Например:

```
?- принадлежит( X, [a,b,c] ).
X = a;
no                (нет)
```

5.2.3. Добавление элемента к списку, если он в нем отсутствует (добавление без дублирования)

Часто требуется добавлять элемент X в список L только в том случае, когда в списке еще нет такого элемента. Если же X уже есть в L, тогда L необходимо оставить без изменения, поскольку нам не нужны лишние дубликаты X. Отношение **добавить** имеет три аргумента:

добавить(X, L, L1)

где X – элемент, который нужно добавить, L – список, в который его нужно добавить, L1 – результи-

рующий новый список. Правила добавления можно сформулировать так:

если X принадлежит к L, то L1 = L,
иначе L1 – это список L с добавленным к нему элементом X.

Проще всего добавлять X в начало списка L так, чтобы X стал головой списка L1. Запрограммировать это можно так:

```
добавить( X, L, L) :- принадлежит( X, L), !.
добавить( X, L, [X | L] ).
```

Поведение этой процедуры можно проиллюстрировать следующим примером:

```
?- добавить( a, [b,c], L).
L = [a,b,c]

?- добавить( X, [b,c], L).
L = [b,c]
X = b

?- добавить( a, [b,c,X], L).
L = [b,c,a]
X = a
```

Этот пример поучителен, поскольку мы не можем легко запрограммировать «недублирующее добавление», не используя отсечения или какой-либо другой конструкции, полученной из него. Если мы уберем отсечение в только что рассмотренной программе, то отношение **добавить** будет добавлять дубликаты элементов, уже имеющихся в списке. Например:

```
?- добавить( a, [a,b,c], L).
L = [a,b,c]
L = [a,a,b,c]
```

Поэтому отсечение требуется здесь для правильного определения отношения, а не только для повышения эффективности. Этот момент иллюстрируется также и следующим примером.

5.2.4. Задача классификации объектов

Предположим, что у нас есть база данных, содержащая результаты теннисных партий, сыгранных членами некоторого клуба. Подбор пар противников для каждой партии не подчинялся какой-либо системе, просто каждый игрок встречался с несколькими противниками. Результаты представлены в программе в виде фактов, таких как

```
победил( том, джон).
победил( энн, том).
победил( пат, джим).
```

Мы хотим определить отношение

```
класс( Игрок, Категория)
```

которое распределяет игроков по категориям. У нас будет три категории:

победитель – любой игрок, победивший во всех сыгранных им играх

боец – любой игрок, в некоторых играх победивший, а в некоторых проигравший

спортсмен – любой игрок, проигравший во всех сыгранных им партиях

Например, если в нашем распоряжении есть лишь приведенные выше результаты, то ясно, что Энн и Пат – победители, Том – боец и Джим – спортсмен.

Легко сформулировать правило для бойца:

X – боец, если
существует некоторый Y, такой, что X победил Y, и
существует некоторый Z, такой, что Z победил X.

Теперь правило для победителя:

X – победитель, если
X победил некоторого Y и
X не был побежден никем.

Эта формулировка содержит отрицание «не», которое нельзя впрямую выразить при помощи тех возможностей Пролога, которыми мы располагаем к настоящему моменту. Поэтому оказывается, что формулировка отношения победитель должна быть более хитрой. Та же проблема возникает и при формулировке правил для отношения спортсмен. Эту проблему можно обойти, объединив определения отношений победитель и боец и использовав связку «иначе». Вот такая формулировка:

```
Если X победил кого-либо и X был кем-то
        побежден,
то X - боец,
иначе,    если X победил кого-либо,
          то X - победитель,
          иначе,    если X был кем-то побежден,
                    то X - спортсмен.
```

Такую формулировку можно сразу перевести на Пролог. Взаимные исключения трех альтернативных категорий выражаются при помощи отсечений:

```
класс( X, боец) :-
    победил( X, _),
    победил( _, X), !.

класс( X, победитель) :-
    победил( X, _), !.

класс( X, спортсмен) :-
    победил( _, X).
```

Заметьте, что использование отсечения в предложении для категории победитель не обязательно, что связано с особенностями наших трех классов.

Упражнения

5.1. Пусть есть программа:

```
p( 1).
p( 2) :- !.
p( 3).
```

Напишите все ответы пролог-системы на следующие вопросы:

```
(a)   ?- p( X).
(b)   ?- p( X), p(Y).
(c)   ?- p( X), !, p(Y).
```

5.2. Следующие отношения распределяют числа на три класса – положительные, нуль и отрицательные:

```
класс( Число, положительные) :- Число > 0.
класс( 0, нуль).
класс( Число, отрицательные) :- Число < 0.
```

Сделайте эту процедуру более эффективной при помощи отсечений.

5.3. Определите процедуру

разбить(Числа, Положительные, Отрицательные)

которая разбивает список чисел на два списка: список, содержащий положительные числа (и нуль), и список отрицательных чисел. Например,

разбить([3,–1,0,5,–2], [3,0,5], [–1,–2])

Предложите две версии: одну с отсечением, другую – без.

5.3. Отрицание как неуспех

«Мэри любит всех животных, кроме змей». Как выразить это на Прологе? Одну часть этого утверждения выразить легко: «Мэри любит всякого X, если X – животное». На Прологе это записывается так:

```
любит( мэри, X) :- животное ( X).
```

Но нужно исключить змей. Это можно сделать, использовав другую формулировку:

Если X – змея, то «Мэри любит X» – не есть истина,
иначе, если X – животное, то Мэри любит X.

Сказать на Прологе, что что-то не есть истина, можно при помощи специальной цели **fail** (неуспех),

которая всегда терпит неудачу, заставляя потерпеть неудачу и ту цель, которая является ее родителем. Вышеуказанная формулировка, переведенная на Пролог с использованием **fail**, выглядит так:

```
любит( мэри, X) :-
    змея( X), !, fail.

любит( Мэри, X) :-
    животное ( X).
```

Здесь первое правило позаботится о змеях: если X – змея, то отсечение предотвратит перебор (исключая таким образом второе правило из рассмотрения), а **fail** вызовет неуспех. Эти два предложения можно более компактно записать в виде одного:

```
любит( мэри, X) :-
    змея( X), !, fail;
    животное ( X).
```

Ту же идею можно использовать для определения отношения

```
различны( X, Y)
```

которое выполняется, если X и Y не совпадают. При этом, однако, мы должны быть точными, потому что «различны» можно понимать по-разному:

- X и Y не совпадают буквально;
- X и Y не сопоставимы;
- значения арифметических выражений X и Y не равны.

Давайте считать в данном случае, что X и Y различны, если они не сопоставимы. Вот способ выразить это на Прологе:

Если X и Y сопоставимы, то
    цель **различны(X, Y)** терпит неуспех
    иначе цель **различны(X, Y)** успешна.

Мы снова используем сочетание отсечения и **fail**:

```
различны( X, X) :- !, fail.

различны( X, Y).
```

То же самое можно записать и в виде одного

предложения:

```
различны( X, Y) :-
    X = Y, !, fail;
    true.
```

Здесь true – цель, которая всегда успешна.

Эти примеры показывают, что полезно иметь унарный предикат "not" (не), такой, что

```
not( Цель)
```

истинна, если Цель не истинна. Определим теперь отношение not следующим образом:

Если Цель успешна, то not(Цель) неуспешна,
иначе not(Цель) успешна.

Это определение может быть записано на Прологе так:

```
not( P) :-
    P, !, fail;
    true.
```

Начиная с этого момента мы будем предполагать, что not – это встроенная прологовская процедура, которая ведет себя так, как это только что было определено. Будем также предполагать, что оператор not определен как префиксный, так что цель

```
not( змея( X) )
```

можно записывать и как

```
not змея( X)
```

Многие версии Пролога поддерживают такую запись. Если же приходится иметь дело с версией, в которой нет встроенного оператора not, его всегда можно определить самим.

Следует заметить, что not, как он здесь определен с использованием неуспеха, не полностью соответствует отрицанию в математической логике. Эта разница может породить неожиданности в поведении программы, если оператором not пользоваться небрежно. Этот вопрос будет рассмотрен в данной главе позже.

Тем не менее not – полезное средство, и его часто можно с выгодой применять вместо отсечения.

Наши два примера можно переписать с not:

```
любит( мэри, X) :-
    животное ( X),
    not змея( X).

различны( X, Y) :-
    not( X = Y).
```

Это, конечно, выглядит лучше, нежели наши прежние формулировки. Вид предложений стал более естественным, и программу стало легче читать.

Нашу программу теннисной классификации из предыдущего раздела можно переписать с использованием not так, чтобы ее вид был ближе к исходным определениям наших трех категорий:

```
класс( X, боец) :-
    победил( X, _),
    победил( _, X).

класс( X, победитель) :-
    победил( X, _),
    not победил( _, X).

класс( X, спортсмен) :-
    not победил( X, _).
```

В качестве еще одиого примера использования not рассмотрим еще раз программу 1 для решения задачи о восьми ферзях из предыдущей главы (рис. 4.7). Мы определили там отношение небьет между некоторым ферзем и остальными ферзями. Это отношение можно определить также и как отрицание отношения «бьет». На рис. 5.3 приводится соответствующим образом измененная программа.

Упражнения

5.4. Даны два списка Кандидаты и Исключенные, напишите последовательность целей (используя принадлежит и not), которая, при помощи перебора, найдет все элементы списка Кандидаты, не входящие в список Исключенные.

5.5. Определите отношение, выполняющее вычитание множеств:

```
решение( []).
решение( [X/Y | Остальные] ) :-
    решение( Остальные),
    принадлежит( Y, [1,2,3,4,5,6,7,8] ),
    not бьет( X/Y, Остальные).
бьет( X/Y, Остальные) :-
    принадлежит( X1/Y1, Остальные),
    ( Y1 = Y;
        Y1 is Y + X1 - X;
        Y1 is Y - X1 + X ).
припадлежит( A, [A | L] ).
принадлежит( A, [B | L] ) :-
    принадлежит( A, L).
% Шаблон решения
шаблон([1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8]).
```

Рис.5.3. Еще одна программа для решения задачи о восьми ферзях.

разность(Множ1, Множ2, Разность)

где все три множества представлены в виде списков. Например,

разность([a,b,c,d], [b,d,e,f], [a,c])

5.6. Определите предикат

унифицируемые(Спис1, Терм, Спис2)

где **Спис2** – список всех элементов **Спис1**, которые сопоставимы с **Терм**'ом, но не конкретизируются таким сопоставлением. Например:

?- унифицируемые([X, b, t(Y)], t(a), Спис).

Спис = [X, t(Y)]

Заметьте, что и X и Y должны остаться неконкретизированными, хотя сопоставление с t(a) вызывает их конкретизацию. Указание: используйте **not (Терм1 = Терм2)**. Если цель **Терм1=Терм2** будет успешна, то **not(Терм1 = Терм2)** потерпит

неудачу и получившаяся конкретизация будет отменена!

5.4. Трудности с отсечением и отрицанием

Используя отсечение, мы кое-что выиграли, но не совсем даром. Преимущества и недостатки применения отсечения были показаны на примерах из предыдущих разделов. Давайте подытожим сначала преимущества:

(1) При помощи отсечения часто можно повысить эффективность программы. Идея состоит в том, чтобы прямо сказать пролог-системе: не пробуй остальные альтернативы, так как они все равно обречены на неудачу.

(2) Применяя отсечение, можно описать взаимоисключающие правила, поэтому есть возможность запрограммировать утверждение:

если условие P, *то* решение Q,
иначе решение R

Выразительность языка при этом повышается.

Ограничения на использование отсечения проистекают из того, что есть опасность потерять такое важное для нас соответствие между декларативным и процедурным смыслами программы. Если в программе нет отсечений, то мы можем менять местами порядок предложений и целей, что повлияет только на ее эффективность, но не на декларативный смысл. Если же отсечения в ней присутствуют, то изменение порядка предложений может повлиять на ее декларативный смысл. Это значит, что программа с измененным порядком, возможно, будет давать результаты, отличные от результатов исходной программы. Вот пример, демонстрирующий этот факт:

```
p :- a, b.
p :- c.
```

Декларативный смысл программы: p истинно тогда и только тогда, когда истинны одновременно и a, и b или истинно c. Это можно записать в виде такой логической формулы:

p <===> (a & b) ∪ c

Можно поменять порядок этих двух предложений, но декларативный смысл останется прежним. Введем теперь отсечение

```
p :- a, !, b.
p :- c.
```

Декларативный смысл станет теперь таким:

p <===> (a & b) ∪ (~a & c)

Если предложения поменять местами

```
p :- c.
p :- a, !, b.
```

декларативный смысл станет таким:

p <===> c ∪ (a & b)

Важным моментом здесь является то, что при использовании отсечения требуется уделять больше внимания процедурным аспектам. К несчастью, эта дополнительная трудность повышает вероятность ошибок программирования.

В наших примерах из предыдущего раздела мы видели, что удаление отсечений из программы может привести к изменению ее декларативного смысла. Но были также и такие случаи, когда отсечение на него не влияло. Использование отсечений последнего типа требует меньшей осторожности, и поэтому такие отсечения иногда называют «зелеными отсечениями». С точки зрения наглядности программы такие отсечения «невинны» и их использование вполне приемлемо. При чтении программы их можно просто игнорировать.

Напротив, отсечения, влияющие на декларативный смысл, называются «красными». Красные отсечения – это такие отсечения, которые делают программу трудной для понимания, и их нужно применять с особой осторожностью.

Отсечение часто используется в комбинации со специальной целью fail. В частности, мы определили отрицание какой-либо цели (not), как ее неуспех. Определенное таким образом отрицание представляет

собой просто особый (более ограниченный) вид отсечения. Из соображений ясности программ мы предпочтем пользоваться not вместо комбинации *отсечение – неуспех* (всюду, где возможно), поскольку отрицание является понятием более высокого уровня, чем *отсечение – неуспех*.

Следует заметить, что использование оператора not также может приводить к неприятностям, и его тоже следует применять с осторожностью. Трудность заключается в том, что тот оператор not, который был нами определен, не в точности соответствует отрицанию в математике. Если спросить

```
?- not человек( мэри).
```

система, возможно, ответит «да». Не следует понимать этот ответ как «мэри не человек». Что в действительности пролог-система хочет сказать своим «да», так это то, что программе не хватает информации для доказательства утверждения «Мэри – человек». Это происходит потому, что при обработке цели not система не пытается доказать истинность этой цели впрямую. Вместо этого она пытается доказать противоположное утверждение, и если такое противоположное утверждение доказать не удается, система считает, что цель not – успешна. Такое рассуждение основано на так называемом *предположении о замкнутости мира*. В соответствии с этим постулатом *мир замкнут* в том смысле, что все в нем существующее либо указано в программе, либо может быть из нее выведено. И наоборот – если что-либо не содержится в программе (или не может быть из нее выведено), то оно не истинно и, следовательно, истинно его отрицание. Это обстоятельство требует особого внимания, поскольку мы обычно не считаем мир замкнутым: если в программе явно не сказано, что

```
человек( мэри)
```

то мы этим обычно вовсе не хотим сказать, что Мэри не человек.

Дальнейшее изучение опасных аспектов использования not проведем на таком примере:

```
r( a).
g( b).
p( X) :- not r( X).
```

Если спросить теперь

```
?- g( X), p(X).
```

система ответит

```
X = b
```

Если же задать тот же вопрос, но в такой форме

```
?- p(X), g( X).
```

система ответит

```
no     (нет)
```

Читателю предлагается проследить работу программы по шагам, чтобы понять, почему получились разные ответы. Основная разница между вопросами состоит в том, что переменная X к моменту вычисления p(X) в первом случае была уже конкретизирована, в то время как во втором случае этого еще не произошло.

Мы детально обсудили аспекты применения отсечения, которое неявно присутствует в not. При этом нами руководило желание предупредить пользователей о соблюдении необходимой осторожности, а вовсе не желание убедить их совсем не пользоваться этим оператором. Отсечение полезно, а часто и необходимо. А что касается трудностей Пролога, порождаемых отсечением, то подобные неудобства часто возникают и при программировании на других языках.

Резюме

- Отсечение подавляет перебор. Его применяют как для повышения эффективности программ, так и для повышения выразительности языка.
- Эффективность повышается путем прямого указания (при помощи отсечения) пролог-системе не проверять альтернативы, про которые нам заранее известно, что они должны потерпеть неудачу.
- Отсечение дает возможность сформулировать взаимно исключающие утверждения при помощи правил вида:

если **Условие** *то* **Утверждение1** *иначе* **Утверждение2**

- Отсечение дает возможность ввести *отрицание как неуспех*: **not(Цель)** определяется через неуспех цели **Цель**.
- Иногда бывают полезными две особые цели **true** и **fail**. **true** – всегда успешна и **fail** – всегда терпит неудачу.
- Существуют ограничения в применении отсечения: его появление может нарушить соответствие между декларативным и процедурным смыслами программы. Поэтому хороший стиль программирования предполагает осторожное применение отсечений и отказ от их применения без достаточных оснований.
- Оператор **not**, определенный через неуспех, не полностью соответствует отрицанию в математической логике. Поэтому **not** тоже нужно применять с осторожностью.

Литература

Различать «зеленые и «красные» отсечения предложил ван Эмден (1982).

van Emden M. (1982). Red and green cuts. *Logic Programming Newsletter: 2.*

6 ВВОД И ВЫВОД

В этой главе мы рассмотрим некоторые встроенные средства для записи данных в файл и считывания их из файла. Такие средства можно также применять и для форматирования объектов данных программы, чтобы получить желаемую форму их внешнего представления. Одновременно мы рассмотрим и средства синтеза и декомпозиции атомов.

6.1. Связь с файлами

До сих пор мы применяли только один метод связи пользователя с программой – пользователь задает программе вопросы, а программа ему отвечает, конкретизируя переменные. Такой механизм связи прост и практичен и, несмотря на свою простоту, обеспечивает ввод и вывод информации. Однако он обладает слишком малой гибкостью и поэтому часто не совсем удобен. В следующих случаях требуется расширение этого основного механизма связи:

- ввод данных в форме, отличной от вопроса – например, в виде предложений, написанных на английском языке
- вывод информации в произвольном формате
- обмен информацией с произвольным файлом, а не только с пользовательским терминалом

Встроенные предикаты, предназначенные для этого расширения, отличаются в разных реализациях Пролога. Мы изучим здесь простой и удобный набор таких

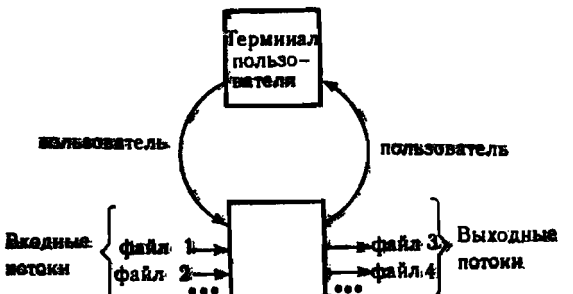


Рис. 6.1. Связь между пролог-программой и различными файлами.

предикатов, применяемый во многих реализациях. Однако за деталями и специфическими особенностями следует, конечно, обращаться к руководствам по конкретным пролог-системам.

Рассмотрим вначале вопрос о том, как обмениваться информацией с файлами, а затем – как можно вводить и выводить данные в разных форматах.

На рис. 6.1 показана общая ситуация, в которой пролог-программа взаимодействует с несколькими файлами. Она может, в принципе, считывать данные из нескольких входных файлов, называемых также *входными потоками*, и выводить данные в несколько выходных файлов, называемых *выходными потоками*. Информация, поступающая с пользовательского терминала, рассматривается просто как еще один входной поток. Аналогично информация, выводимая на этот терминал, рассматривается как один из выходных потоков. На оба этих «псевдофайла» ссылаются с помощью имени user (пользователь). Имена остальных файлов программист должен выбирать в соответствии с правилами именования файлов, принятыми в используемой компьютерной системе.

В каждый момент выполнения пролог-программы лишь два файла являются «активными»: один для ввода, другой – для вывода. Эти два файла называются *текущим входным потоком* и *текущим выходным потоком* соответственно. В начальный момент эти два потока соответствуют терминалу. Текущий входной поток

может быть заменен на другой файл **ИмяФайла** посредством цели

```
see( ИмяФайла)          ( Смотри(ИмяФайла) )
```

Такая цель всегда успешна (если только с файлом **ИмяФайла** все в порядке), а в качестве побочного эффекта происходит переключение ввода с предыдущего входного потока на файл **ИмяФайла**. Поэтому типичным примером использования предиката see является следующая последовательность целей, которая считывает информацию из файла **файл1**, а затем переключается обратно на терминал:

```
...
see( файл1),
читать_из_файла( Информация),
see( user),                 ( user - пользователь)
...
```

Текущий выходной поток может быть изменен при помощи цели вида

```
tell( ИмяФайла)          ( сообщить( ИмяФайла) )
```

Следующая последовательность целей выводит некоторую информацию в **файл3**, а после этого перенаправляет последующий вывод обратно на терминал:

```
...
tell( файл3),
записать_в_файл( Информация),
tell( user),
...
```

Цель

```
seen                        ( конец чтения)
```

закрывает текущий входной файл. Цель

```
told                        ( конец записи)
```

закрывает текущий выходной файл.

Файлы могут обрабатываться только последовательно. В этом смысле все файлы ведут себя так же, как терминал. Каждый запрос на чтение из входного файла приводит к чтению в текущей позиции текущего

входного потока. После этого чтения текущая позиция, естественно, будет перемещена на следующий, еще не прочитанный элемент данных. Следующий запрос на чтение приведет к считыванию, начиная с этой новой текущей позиции. Если запрос на чтение делается в конце файла, то в качестве ответа на такой запрос выдается атом **end_of_file** (конец файла). Считанную один раз информацию считать вторично невозможно.

Запись производится точно так же; каждый запрос на вывод информации приведет к тому, что она будет присоединена в концу текущего выходного потока. Невозможно сдвинуться назад и переписать часть файла.

Все файлы являются «текстовыми», т. е. файлами, состоящими из символов. Символы – это буквы, цифры и специальные знаки. О некоторых из них говорят, что они непечатаемые, поскольку, будучи выведенными на терминал, они не появляются на экране. Однако их присутствие может сказаться каким-либо другим образом, например появятся пробелы или пустые строки.

Существуют два основных способа, с помощью которых файлы рассматриваются в Прологе в зависимости от формы записанной в них информации. Один способ – рассматривать символ как основной элемент файла. Соответственно один запрос на ввод или вывод приведет к чтению или записи одного символа. Для этой цели предназначены встроенные предикаты **get**, **get0** и **put** (получить, получить0 и выдать).

Другой способ рассматривать файл – считать, что в качестве основных элементов построения файла используются более крупные единицы текста. Такой естественной более крупной единицей является прологовский терм. Поэтому каждый запрос на ввод/вывод такого типа приведет к переносу целого терма из текущего входного потока или в текущий выходной поток соответственно. Предикатами для переноса термов являются предикаты **read** и **write** (читать и писать). В этом случае информация в файле должна, конечно, по форме соответствовать синтаксису термов.

Очевидно, что выбор формы организации файла зависит от задачи. Всякий раз, когда особенности задачи допускают естественное представление информации в соответствии с синтаксисом термов, следует

предпочесть файлы, состоящие из термов. Тогда появится возможность за одно обращение к вводу или выводу пересылать целые осмысленные фрагменты информации. С другой стороны, существуют задачи, природа которых диктует иную организацию файлов. Примером такого рода задачи является обработка предложений естественного языка, скажем, для организации диалога между системой и пользователем на английском языке. В таких случаях файлы следует рассматривать как последовательности символов, которые не укладываются в синтаксис термов.

6.2. Обработка файлов термов

6.2.1. *read* и *write*

Встроенный предикат read используется для чтения термов из текущего входного потока. Цель

```
read( X)
```

вызывает чтение следующего терма T и сопоставление его с X. Если X – переменная, то в результате X конкретизируется и становится равным T. Если сопоставление терпит неудачу, цель read(X) тоже терпит неудачу. Предикат read – детерминированный в том смысле, что в случае неуспеха не происходит возврата для ввода следующего терма. После каждого терма во входном файле должна стоять точка или пробел, или символ возврата каретки.

Если read(X) вычисляется в тот момент, когда достигнут конец текущего входного файла, тогда X конкретизируется атомом end_of_file (конец файла).

Встроенный предикат write выводит терм. Поэтому цель

```
write( X)
```

выведет терм X в текущий выходной файл. X будет выведен в той же стандартной форме, в какой обычно

пролог-система изображает на экране или распечатке значения переменных. Полезной особенностью Пролога является то, что процедура write «знает», как изображать любой терм, как бы сложен он не был.

Существуют дополнительные встроенные предикаты для форматирования вывода. Они вставляют пробелы и дополнительные строки в выходной поток. Цель

```
tab( N)
```

выводит N пробелов. Предикат nl (без аргументов) приводит к переходу на новую строку.

Следующие примеры иллюстрируют использование этих процедур.

Предположим, у нас есть процедура для вычисления кубов чисел:

```
куб( N, C) :-
    C is N * N * N.
```

Предположим далее, что мы хотим применить ее для вычисления кубов элементов некоторой последовательности чисел. Это можно сделать с помощью последовательности вопросов:

```
?- куб( 2, X).
X = 8

?- куб( 5, Y).
Y = 125

?- куб( 12, Z).
Z = 1728
```

Для получения каждого результата нам придется набирать соответствующую цель. Давайте теперь изменим эту программу так, чтобы процедура куб сама читала соответствующие данные. Теперь программа будет сама читать данные и выводить их кубы до тех пор, пока не будет прочитан атом стоп:

```
куб :-
    read( X),
    обработать( X).

обработать( стоп) :- !.

обработать( N) :-
    C is N * N * N,
    write( C),
    куб.
```

Это был пример программы, декларативный смысл которой трудно сформулировать. В то же время ее процедурный смысл совершенно ясен: чтобы вычислить куб, сначала нужно считать X, а затем его обработать; если X = стоп, то все сделано, иначе вывести X^3 и рекурсивно запустить процедуру куб для обработки остальных чисел.

С помощью этой новой процедуры таблица кубов чисел может быть получена таким образом:

```
?- куб.

2.
8
5.
125
12.
1728
стоп.
yes
```

Числа 2, 5 и 12 были введены пользователем с терминала, остальные числа были выведены программой. Заметьте, что после каждого числа, введенного пользователем, должна стоять точка, которая сигнализирует о конце терма.

Может показаться, что приведенную процедуру куб можно упростить. Однако следующая попытка такого упрощения является ошибочной:

```
куб :-
    read( стоп), !.

куб :-
    read( N),
    C is N * N * N,
    write( C),
    куб.
```

Причина, по которой эта процедура работает неправильно, станет очевидной, если проследить, какие действия она выполняет с входным аргументом, скажем с числом 5. Цель read(стоп) потерпит неудачу при чтении этого числа, и оно будет потеряно навсегда. Следующая цель read введет следующий терм. С другой стороны может случиться, что сигнал стоп будет считан целью read(N), что приведет к попыт-

ке перемножить нечисловую информацию.

Процедура **куб** ведет диалог между пользователем и программой. В таких случаях обычно желательно, чтобы программа перед тем, как читать с терминала новые данные, дала сигнал пользователю о том, что она готова к приему информации, а также, возможно, и о том, какого вида информация ожидается. Это делается обычно путем выдачи «приглашения» перед чтением. Нашу процедуру куб можно для этого изменить, например, так:

```
куб :-
    write( 'Следующее число, пожалуйста:'),
    read( X),
    обработать( X).

обработать( стоп) :- !.

обработать( N) :-
    C is N * N * N,
    write( 'Куб'), write( N), write( 'равен'),
    write( C), nl,
    куб.
```

Разговор с новой версней мог бы быть, например, такнм:

```
?- куб.

Следующее число, пожалуйста: 5.
Куб 5 равен 125

Следующее число, пожалуйста: 12.
Куб 12 равен 1728

Следующее число, пожалуйста: стоп.
yes
```

В некоторых реализациях для того, чтобы приглашение появилось на экране перед чтением, необходимо выдать дополнительный запрос (такой, скажем, как **ttyflush**) после записи.

В последующих разделах мы увидим некоторые типичные примеры операций, в которых участвуют чтение и запись.

6.2.2. Вывод списков

Кроме стандартного прологовского формата для списков существуют несколько других естественных форм их внешнего представления, которые в некоторых ситуациях являются более предпочтительными. Следующая процедура

```
вывспис( L)
```

выводит список L так, что каждый его элемент занимает отдельную строку:

```
вывспис( []).
вывспис( [X | L]) :-
    write( X), nl,
    вывспис( L).
```

Если у нас есть список списков, то одной из естественных форм его вывода является такая, при которой все элементы каждого списка записываются на отдельной строке. Для этого мы определим процедуру **вывспис2**. Вот пример ее использования:

```
?- вывспис2( [ [a,b,c], [d,e,f], [g,h,i] ] ).

a b c
d e f
g h i
```

Процедура, выполняющая эту работу, такова:

```
вывспис2( []).
вывспис2( [L | LL] ) :-
    строка( L), nl,
    вывспис1( LL).
строка( []).
строка( [X | L] ) :-
    write( X), tab( 1),
    строка( L).
```

Список целых чисел иногда удобно представить в виде диаграммы. Следующая процедура **диагр** выводит

список в такой форме (предполагается, что числа списка заключены между 0 и 80). Пример ее использования:

```
?- диагр( [3,4,6,5] ).
***
****
******
*****
```

Процедуру диагр можно определить так:

```
диагр( [N | L] ) :-
  звездочки( N), nl,
  диагр( L).
звездочки( N) :-
  N > 0,
  write( *),
  N1 is N - 1,
  звездочки( N1).
звездочки( N) :-
  N =< 80.
```

6.2.3. Формирование термов

Предположим, наша программа имеет дело с семьями, которые представлены в виде термов так, как это сделано в гл. 4 (рис. 4.1). Тогда, если, перемен-

```
родители
    том фокс, датарожд 7 май 1950, работает bbc,
                                   оклад 15200
    эин фокс, датарожд 9 май 1951, неработает
дети
        пат  фокс, датарожд 5 май 1973, неработает
        джим фокс, датарожд 5 май 1973, неработает
```

Рис. 6.2. Улучшенный формат вывода термов, представляющих семью.

ная F конкретизированиа термом, изображенным на рис. 4.1, то цель

```
write( F)
```

вызовет вывод этого терма в стандартной форме примерно так:

```
семья(членсемьи(том,фокс,дата(7,май,1950),
        работает(bbc,15200)),
    членсемьи(энн,фокс,дата(9,май,1951),
        неработает),
    [членсемьи(пат,фокс,дата(5,май,1973),
        неработает),
    членсемьи(джим,фокс,дата(5,май,1973),
        неработает)])
```

Здесь содержится полная информация, однако форма представлеиия легко может запутать, поскольку трудно проследить, какие ее части образуют самостоятельные семантические единицы. Поэтому обычно предпочитают выводить такую информацию в каком-либо формате, например так, как показано на рис. 6.2. Процедура

```
вывсемью( F)
```

с помощью которой это достигается, приведена на рис. 6.3.

```
вывсемью( семья ( Муж, Жена, Дети) :-
    nl, write( родители), nl, nl,
    вывчленсемьи( Муж), nl,
    вывчленсемьи( Жена), nl, nl,
    write( дети), nl, nl,
    вывчленсемьи( Дети).

вывчленсемьи( членсемьи(Имя,Фамилия,дата(Д,М,Г),
                                    Работа) ) :-
    tab(4), write( Имя),
    tab(1), write( Фамилия),
    write( ', дата рождения'),
    write( Д), tab( 1),
    write( М), tab( 1),
    write( Г), write( ','),
    вывработу( Работа).

вывспичлсемьи( []).
```

```
вывсписнчлсемьи( [P | L]) :-
    вывчленсемьи( P), nl,
    вывсписнчлсемьи( L).
вывработу( неработает) :-
    write( неработает).

вывработу( работает( Место, Оклад) ) :-
    write(' работает '), write( Место),
    write( ', оклад '), write( Оклад).
```

Рис. 6.3. Программа, обеспечивающая вывод в формате, представленном на рис. 6.2.

6.2.4. Обработка произвольного файла термов

Типичная последовательность целей для обработки файла F от начала до конца будет выглядеть примерно так:

```
..., see( F), обработкафайла, see( user), ...
```

Здесь **обработкафайла** – процедура, которая читает и обрабатывает последовательно каждый терм файла F один за другим до тех пор, пока не встретится конец файла. Приведем типичную схему для процедуры

```
обработкафайла:
    бработкафайла :-
        read( Терм),
        обработка( Терм).

    обработка( end_of_file) :- !.
                        % Все сделано

    обработка( Терм) :-
        обраб( Терм),
                        % Обработать текущий элемент
        обработкафайла.
                        % Обработать оставшуюся часть файла
```

Здесь **обраб(Терм)** представляет процедуру обработки отдельного терма. В качестве примера такой обработки рассмотрим процедуру, которая выдает на терминал каждый терм вместе с его порядковым номе-

ром. Назовем эту процедуру **показфайла**. У нее должен быть дополнительный аргумент для подсчета прочитанных термов:

```
показфайла( N) :-
    read( Терм),
    показ( Терм, N).

показ( Терм, N) :- !.
    write( N), tab( 2), write( Терм),
    N1 is N + 1,
    показфайла( N1).
```

Вот другой пример программы обработки файлов, построенной по подобной схеме. Пусть есть файл с именем **файл1**, термы которого имеют форму

изделие(НомерИзд, Описание, Цена, ИмяПоставщика)

Каждый терм описывает одну строку каталога изделий. Нужно построить новый файл, содержащий только те изделия, которые выпускаются каким-то конкретным поставщиком. Поскольку поставщик в этом новом файле у всех изделий будет одинаков, его имя нужно записать только один раз в самом начале и убрать из всех остальных термов. Процедура будет называться

создатьфайл(Поставщик)

Например, если исходный каталог хранится в файле **файл1**, а мы хотим создать специальный каталог в файле **файл2**, содержащий всю информацию о том, что поставляет Гаррисон, тогда мы применим процедуру **создатьфайл** следующим образом:

```
?- see(файл1), tell(файл2), создатьфайл(гаррисон),
    see(user), tell(user).
```

Процедуру **создатьфайл** можно определить так:

```
создатьфайл( Поставщик) :-
    write( Поставщик), write( '.'), nl,
    создатьостальное( Поставщик).

создатьостальное( Поставщик) :-
    read( Изделие),
    обработать( Изделие, Поставщик).

обработать( end_of_file) :- !.
```

```
обработать(Изделие(Ном, Опис, Цена, Поставщик),
           Поставщик) :- !,
  write( Изделие( Ном, Опис, Цена) ),
  write( '.'), nl,
  создатьостальное( Поставщик).

обработать( _, Поставщик) :-
  создатьостальное( Поставщик).
```

Обратите внимание на то, что **обработать** вписывает точки между термами, чтобы впоследствии файл мог быть прочитан процедурой **read**.

Упражнения

6.1. Пусть **f** – файл термов. Определите процедуру

найтитерм(Терм)

которая выводит на терминал новый терм из **f**, сопоставимый с Терм'ом.

6.2. Пусть **f** – файл термов. Напишите процедуру

найтивсетермы(Терм)

которая выводит на экран все термы из **f**, сопоставимые с Терм'ом. Обеспечьте при этом, чтобы во время поиска Терм не конкретизировался (это могло бы помешать ему сопоставиться с другими термами дальше по файлу).

6.3. Обработка символов

Символ записывается в текущий выходной поток при помощи цели

put(C)

где C – символ, который нужно вывести, в кодировке ASCII (число от 0 до 127). Например, вопрос

```
?- put( 65), put( 66), put( 67).
```

породит следующий вывод:

ABC

65 - ASCII-код 'A', 66 - 'B', 67 - 'C'.

Одиночный символ можно считать из текущего входиого потока при помощи цели

get0(C)

Она вызывает чтение символа из входиого потока, и переменная С коикретизируется ASCII-кодом этого символа. Вариантом предиката **get0** является **get**, который используется для чтения символов, отличных от пробела. Поэтому цель

get(C)

вызовет пропуск всех непечатаемых символов (в частиости пробелов) от текущей позиции во входном потоке до первого печатаемого символа. Этот символ затем тоже считывается и С конкретизируется его ASCII-кодом.

В качестве примера использования предикатов, переносящих одиночные символы, давайте рассмотрим процедуру **сжатие**, выполияющую следующую работу: считывание из входного потока произвольного предложения и вывод его же, но в форматированном виде - все группы идущих подряд пробелов заменены иа одиночные пробелы. Для простоты будем считать, что все предложения входного потока, обрабатываемые процедурой **сжатие**, оканчиваются точками, а слова в них отделены одно от другого одним или иескольким пробелами, и только ими. Тогда следующее предложение будет допустимым:

Робот пытался иалить вина из бутылки.

Цель **сжатие** выведет его в таком виде:

Робот пытался налить вина из бутылки.

Процедура **сжатие** будет иметь такую же структуру, как и процедуры обработки файлов из предыдущего раздела. Сначала она прочтет первый символ, выведет его, а затем завершит обработку, в зависимости от того, каким был этот символ. Есть трн альтернативы, которые соответствуют следующим случаям: символ является точкой, пробелом или буквой. Взаимное исключение этих трех альтернатив обеспе-

чивается в программе отсечениями:

```
сжатие :-
    get0( C),
    put( C),
    сделатьостальное( C).

сделатьостальное( 46) :- !.
                    % 46 -ASCII-код точки,Все сделано
сделатьостальное( 32) :- !,
                    % 32 - ASCII-код пробела
    get( C),
    put( C),
    сделатьостальное( C).

сделатьостальное( Буква) :-
    сжатие.
```

Упражнение

6.3. Обобщите процедуру **сжатие** на случай запятых. Все пробелы, стоящие непосредственно перед запятой, нужно убрать, а после каждой запятой нужно поместить единственный пробел.

6.4. Создание и декомпозиция атомов

Часто желательно информацию, считанную как последовательность символов, иметь в программе в виде атома. Для этой цели существует встроенный предикат **name**. Он устанавливает взаимосвязь между атомами и их кодировкой в ASCII. Таким образом,

```
name( A, L)
```

истинно, если L – список кодов ASCII, кодирующих атом A. Например,

```
name( zx232, [122,120,50,51,50] )
```

истинно. Существуют два типичных способа использо-

вания name:

(1) дан атом, разбить его на отдельные символы;
(2) дан список символов, объединить их в один атом.

Примером первого случая применения предиката является программа, которая имеет дело с заказами такси и водителями. Все это представлено в программе атомами

заказ1, заказ2, водитель1, водитель2,
такси1, таксилюкс

Предикат

такси(X)

проверяет, относится ли атом X к тем атомам, которые представляют такси:

```
такси( X) :-
    name( X, Хспис),
    name( такси, Тспис),
    конк( Тспис, _, Хспис).

конк( [], L, L).

конк( [A | L1], L2, [A | L3] ) :-
    конк( L1, L2, L3).
```

Предикаты заказ и водитель можно определить аналогичио.

Наш следующий пример иллюстрирует применение объединения отдельных символов в один атом. Мы определим предикат

читпредложение(Списсслов)

который считает предложение с произвольной формой на естественном языке и конкретизирует Списсслов некоторым внутренним представлением этого предложения. В качестве внутреннего представления, обеспечивающего возможность дальнейшей обработки предложения, естественно избрать следующее: каждое слово входного предложения представляется прологовским атомом, а все предложение представляется списком этих атомов. Например, если входной поток таков:

Мэри было приятно видеть неудачу робота.

то цель **читпредложение(Предложение)** вызовет кон-

кретизацию

Предложение=['Мэри', было, приятно, видеть, неудачу, робота]

Для простоты будем считать, что каждое предложение оканчивается точкой и внутри него не используются никакие знаки препинания.

Программа для **читпредложение** показана на рис. 6.4. Вначале процедура читает текущий входной символ **Симв**, а затем передает его процедуре **читостальное** для завершения работы. Процедура **читостальное** должна правильно обработать следующие три случая:

(1) **Симв** – точка, тогда все сделано.

(2) **Симв** – пробел, – игнорировать его и **читпредложение** от остального ввода.

(3) **Симв** – буква, – сначала считать слово **Слово**, которое начинается с **Симв** , а затем запустить **читпредложение**, чтобы считать оставшуюся часть предложения, породив при этом **Списслов**. Общим результатом этого будет список **[Слово | Списслов]**.

Процедура, считывающая символы одного слова, такова:

читбуквы(Буква, Буквы, Сделсимв)

Ее три аргумента:

(1) **Буква** – текущая буква (уже считанная) читаемого слова.

(2) **Буквы** – список букв (начинающийся с буквы **Буква**), оставшихся до конца слова.

(3) **Следсимв** – входной символ, непосредственно следующий за читаемым словом. **Следсимв** не должен быть буквой.

Мы завершим данный пример замечанием о возможном применении процедуры **читпредложение**. Ее можно использовать в программе обработки текста на естественном языке. Предложения, представленные в виде списков слов, имеют удобную форму для дальнейшей обработки при помощи Пролога. В простейшем

```
/*
Процедура читпредложение считывает предложение
и из его слов создает список атомов. Например,
    читпредложение( Списслов)
 порождает
    Списслов=['Мэри', было, приятно, видеть, неудачу, робота]
если входным было предложение
    Мэри было приятно видеть неудачу робота.
*/
читпредложение( Списслов) :-
    get0( Симв),
    читостальное( Симв, Списслов).
читостальное( 46, []) :- !.
                    % Конец предложения: 46 - ASCII-код для '.'
читостальное( 32, Списслов) :- !,
                    % 32 - ASCII-код для пробела
читпредложение( Списслов).
                    % Пропустить пробел
читостальное( Буква, [Слово | Списслов]) :-
    читбуквы( Буква, Буквы, Следсимв),
                    % Считать буквы текущего слова
    name( Слово, Буквы),
    читостальное( Следсимв, Списслов).
читбуквы( 46, [], 46) :- !.
                    % Конец слова: 46 - точка
читбуквы( 32, [], 32) :- !.
                    % Конец слова: 32 - пробел
читбуквы( Бкв, [Бкв | Буквы], Следсимв) :-
    get0( Симв),
    читбуквы( Симв, Буквы, Следсимв).
```

Рис. 6.4. Процедура для преобразования предложения в список атомов.

случае такой обработкой мог бы быть поиск во входном предложении определенных ключевых слов. Значительно более сложной задачей является понимание предложения, т. е. извлечение из него смысла, представленного в некотором избранном формализме.

Это важная область исследований в искусственном интеллекте.

Упражнения

6.4. Определите отношение

начинается(Атом, Символ)

для проверки, начинается ли **Атом** с символа **Символ**.

6.5. Определите процедуру **plural**, которая преобразует английские существительные из единственного числа во множественное, добавляя к слову окончание s. Например:

```
?- plural( table, X).
X = tables
```

6.6. Напишите процедуру

поиск(Ключслово, Предложение)

которая при каждом вызове находит в текущем входном файле предложение, содержащее заданное ключевое слово **Ключслово**. **Предложение** в своей исходной форме должно быть представлено в виде последовательности символов или в виде атома (процедуру **читпредложение** из данного раздела можно соответственно модифицировать).

6.5. Ввод программ: *consult, reconsult*

Передавать программы пролог-системе можно при помощи двух встроенных предикатов: **consult** и **reconsult**. Чтобы система считала программу из файла F, нужно поставить цель

```
?- consult( F).
```

В результате все предложения программы, содержащейся в F, будут использованы пролог-системой при

ответе на дальнейшие вопросы пользователя. Если позже в том же сеансе произойдет «консультация» с другим файлом, предложения этого нового файла будут просто добавлены в конец текущего множества предложенний.

Для того, чтобы запустить программу, не обязательно записывать ее в файл, а затем «консультироваться» с ним. Вместо чтения файла система может принимать программу прямо с терминала, который соответствует псевдофайлу **user**. Добиться этого можно так:

```
?- consult( user).
```

После этого система будет ожидать ввода предложений программы с терминала.

В некоторых пролог-системах применяется сокращенная запись для чтения программ из файлов. Файлы, из которых предстоит чтение, просто помещаются в список и этот список используется в качестве цели. Например:

```
?- [файл1, файл2, файл3].
```

Это в точности эквивалентно следующим трем целям:

```
?- consult(файл1),consult(файл2),consult(файл3).
```

Встроенный предикат **reconsult** аналогичен **consult**. Цель

```
?- reconsult( F).
```

даст тот же эффект, что и **consult(F)** с одним исключением. Если в F есть предложения, касающиеся отношений, которые уже были определены ранее, старые определения заменяются на новые из F. Разница между **consult** и **reconsult** в том, что **consult** всегда *добавляет* новые предложения, в то время как **reconsult** *переопределяет* ранее введенные определения. Однако **reconsult** не произведет никакого эффекта на те отношения, о которых в F ничего не сказано.

Следует еще раз заметить, что детали «консультирования» с файлами зависят от конкретной реализации Пролога. Это замечание касается и большинства остальных встроенных процедур.

Резюме

- Ввод и вывод (отличный от связанного с вопросами к программе) осуществляется посредством встроенных процедур. В данной главе описан простой и практичный набор таких процедур, имеющихся во многих реализациях Пролога.
- Файлы являются последовательными. Существуют *текущие входной* и *выходной* потоки. Пользовательский терминал рассматривается как файл с именем **user**.
- Переключение между потоками осуществляется с помощью процедур:

see(Файл)	**Файл** становится текущим входным потоком
tell(Файл)	**Файл** становится текущим выходным потоком
seen	закрывается текущий входной поток
told	закрывается текущий выходной поток

- Файлы читаются и записываются двумя способами:

  как последовательности символов
  как последовательности термов

  Встроенные процедуры для чтения и записи символов и термов таковы:

  read(Терм)
  вводит следующий терм

  write(Терм)
  выводит **Терм**

  put(КодСимвола)
  выводит символ с заданным ASCII-кодом

  get0(КодСимвола)
  вводит следующий символ

  get(КодСимвола)
  вводит ближайший следующий «печатаемый» символ

- Две процедуры облегчают форматирование:

nl	начинает новую строку
tab(N)	выводит N пробелов

- Процедура **name(Атом, СписокКодов)** осуществляет синтез и декомпозицию атомов. **СписокКодов** – список ASCII кодов символов, образующих **Атом**.

7 Другие встроенные процедуры

В данной главе мы изучим некоторые другие, ие упоминавшиеся раиее встроеиные процедуры, предиазначеиные для более серьезного программирования на Прологе. Эти новые процедуры дают возможность запрограммировать операции, которые известиыми иам средствами запрограммировать невозможно. Одии иабор таких процедур касается обработки термов: эти процедуры проверяют, была ли некоторая переменная конкретизироваиа целым числом, они разбирают термы на части, конструируют новые термы и т.д. Другой полезиый набор процедур работает с «базой данных»: процедуры из этого набора добавляют новые отиошеиия в программу или удаляют из нее существующие.

Множество встроеиных процедур сильно зависит от конкретной реализации Пролога. Однако процедуры, обсуждаемые в данной главе, имеются во многих реализациях. Различные реализации могут иметь свои наборы дополнительных средств.

7.1. Проверка типов термов

7.1.1. Предикаты

var, nonvar, atom, integer, atomic

Термы бывают разиых типов: переменные, целые числа, атомы и т.д. Если терм – переменная, то в некоторый момент выполиеиия программы он может ока-

заться конкретизированным или не конкретизированным. Далее, если он конкретизирован, то его значение может быть атомом, структурой и т.п. Иногда бывает полезно узнать, каков тип этого значения. Например, пусть мы хотим сложить значения двух переменных X и Y:

Z is X + Y

Перед вычислением этой цели необходимо, чтобы X и Y были конкретизированы целыми числами. Если у нас нет уверенности в том, что X и Y действительно конкретизированы целыми числами, то перед выполнением арифметического действия нужно проверить это программно.

Для этого следует воспользоваться встроенным предикатом **integer** (целое). Предикат **integer(X)** принимает значение истина, если X – целое или если X – переменная, имеющая целое значение. Будем говорить в этом случае, что X «обозначает» целое. Цель для сложения X и Y можно тогда «защитить» такой проверкой переменных X и Y:

..., integer(X), integer(Y), Z is X + Y, ...

Если неверно, что X и Y оба являются целыми, то система и не будет пытаться их сложить. Таким образом, цели **integer** «охраняют» цель **Z is X + Y** от бессмысленного вычисления.

Встроенные предикаты этого типа таковы: **var** (переменная), **nonvar** (непеременная), **atom** (атом), **integer** (целое), **atomic** (атомарный). Они имеют следующий смысл:

var(X)

Эта цель успешна, если X в текущий момент – не конкретизированная переменная.

nonvar(X)

Эта цель успешна, если X – терм, отличный от переменной, или если X – уже конкретизированная переменная.

atom(X)

Эта цель истинна, если X обозначает атом.

integer(X)

Цель истинна, если X обозначает целое.

atomic(X)

Цель истинна, если X обозначает целое или атом.

Следующие примеры вопросов к пролог-системе иллюстрируют применение этих встроенных предикатов:

```
?- var( Z), Z = 2.
Z = 2

?- Z = 2, var( Z).
no

?- integer( Z), Z = 2.
no

?- Z = 2, integer( Z), nonvar( Z).
Z = 2

?- atom( 22).
no

?- atomic( 22).
yes

?- atom( ==>).
yes

?- atom( p(1) ).
no
```

Необходимость в предикате **atom** продемонстрируем на следующем примере. Пусть мы хотим подсчитать, сколько раз заданный атом встречается в некотором списке объектов. Для этого мы определим процедуру

счетчик(A, L, N)

где A – атом, L – список и N – количество вхождений этого атома. В качестве первой попытки можно было бы определить **счетчик** так:

```
счетчик( _, [], 0).

счетчик( A, [A | L], N) :- !,
    счетчик( A, L, N1),
                % N1 - число вхождений атома в хвост
    N is N1 + 1.

счетчик( A, [_ | L], N) :-
    счетчик( A, L, N).
```

Теперь на нескольких примерах посмотрим, как эта процедура работает:

```
?- счетчик( a, [a,b,a,a], N).
N = 3

?- счетчик( a, [a,b,X,Y], Na).
Na = 3
...

?- счетчик( b, [a,b,X,Y], Nb).
Nb = 3
...

?- L=[a,b,X,Y],счетчик(a,L,Na),счетчик(b,L,Nb).
Na = 3
Nb = 1
X = a
Y = a
...
```

В последнем примере как X, так и Y после конкретизации получили значение a, и поэтому Nb оказалось равным только 1, однако мы хотели не этого. Нас интересовало количество реальных появлений конкретного *атома*, а вовсе не число термов, *сопоставимых* с этим атомом. В соответствии с этим более точным определением отношения **счетчик** мы должны теперь проверять, является ли голова списка атомом. Усовершенствованная программа выглядит так:

```
счетчик( _, [], 0).

счетчик( A, [B | L], N) :-
    atom( B), A = B, !,          % B равно атому A?
    счетчик( A, L, N1),          % Подсчет в хвосте
    N is N1 + 1;
    счетчик( A, L, N).
                % Иначе - подсчитать только в хвосте
```

В следующем более сложном упражнении по программированию числовых ребусов используется предикат **nonvar**.

7.1.2. Решение числового ребуса с использованием *nonvar*

Известным примером числового ребуса является

```
  D O N A L D
+ G E R A L D
-------------
  R O B E R T
```

Задача состоит в том, чтобы заменить буквы D, O, N и т.д. на цифры таким образом, чтобы вышеприведенная сумма была правильной. Разным буквам должны соответствовать разные цифры, иначе возможно тривиальное решение, например, все буквы можно заменить на нулн.

Определнм отношение

сумма(N1, N2, N)

где N1, N2 и N представляют трн числа данного ребуса. Цель **сумма(N1,N2,N)** достигается, еслн существует такая замена букв цифрами, что N1+N2 = N.

Первым шагом к решению будет выбор представления чисел N1, N2 и N в программе. Один из способов – представить каждое чнсло в внде списка его цифр. Например, чнсло 255 будет тогда представляться списком [2,2,5]. Поскольку зиачения цифр нам не нзвестны заранее, каждая цнфра будет обозначаться соответствующей неинициализированной переменной. Используя это представление, мы можем сформулировать задачу так:

```
    [ D,O,N,A,L,D ]
+   [ G,E,R,A,L,D ]
=   [ R,O,B,E,R,T ]
```

Теперь задача состоит в том, чтобы найти такую конкретизацию переменных D, O, N и т.д., для которой сумма верна. После того, как отношение **сумма** будет запрограммнровано, задание для пролог-снстемы на решенне ребуса будет иметь вид

```
?- сумма( [D,O,N,A,L,D],[G,E,R,A,L,D],
          [R,O,B,E,R,T] ).
```

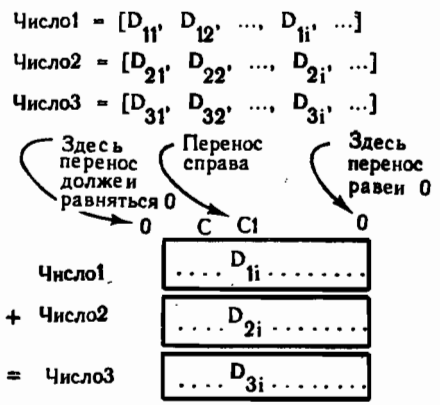


Рис. 7.1. Поразрядное сложение. Отношения в показанном i-м разряде таков: $D_{3i}=(C1+D_{1i}+D_{2i})$ mod 10; $C = (C1 + D_{1i}+ D_{2i})$ div 10 (div – целочисленное деление, mod – остаток от деления).

Для определения отношения **сумма** над списками цифр нам нужно запрограммировать реальные правила суммирования в десятичной системе счисления. Суммирование производится цифра за цифрой, начиная с младших цифр в сторону старших, всякий раз учитывая цифру переноса справа. Необходимо также сохранять множество допустимых цифр, т.е. цифр, которые еще не были использованы для конкретизации уже встретившихся переменных. Поэтому, вообще говоря, кроме трех чисел N1, N2 и N в рассмотрении должна участвовать некоторая дополнительная информация, как показано на рис. 7.1:

- перенос перед сложением
- перенос после сложения
- множество цифр, доступных перед сложением
- оставшиеся цифры, не использованные при сложении

Для формулировки отношения **сумма** мы снова воспользуемся принципом обобщения задачи: введем вспомогательное, более общее отношение **сумма1**. Это отношение будет иметь несколько дополнительных аргументов, соответствующих той дополнительной информации, о которой говорилось выше:

сумма1(N1, N2, N, C1, C, Цифры1, Цифры)

Здесь N1, N2 и N – наши три числа, как и в отношении сумма, C1 – перенос справа (до сложения N1 и N2), а C – перенос влево (после сложения). Пример:

```
?- сумма1( [H,E],[6,E],[U,S],1,1,
           [1,3,4,7,8,9],Цифры ).

H = 8
E = 3
S = 7
U = 4
Цифры = [1,9]
```

Если N1 и N удовлетворяют отношению **сумма**, то, как показано на рис. 7.1, C1 и C должны быть равны 0. **Цифры1** – список цифр, которые не были использованы для конкретизации переменных. Поскольку мы допускаем использование в отношении **сумма** любых цифр, ее определение в терминах отношения **сумма1** выглядит так:

```
сумма( N1, N2, N) :-
    сумма1(N1,N2,N,0,0,[0,1,2,3,4,5,6,7,8,9],_).
```

Бремя решения задачи переложено теперь на отношение **сумма1**. Но это отношение является уже достаточно общим, чтобы можно было определить его рекурсивно. Без ограничения общности мы предположим, что все три списка, представляющие три числа, имеют одинаковую длину. Наш пример, конечно, удовлетворяет этому условию, но если это не так, то всегда можно приписать слева нужное количество нулей к более «короткому» числу.

Определение отношения **сумма1** можно разбить на два случая:

(1) Все три числа представляются пустыми списками. Тогда

сумма1([], [], [], 0, 0, Циф, Циф).

(2) Все три числа имеют какую-то самую левую цифру и справа от нее – остальные цифры. То есть, они имеют вид:

[D1 | N1], [D2 | N2], [D | N]

В этом случае должны выполняться два условия:

(a) Оставшиеся цифры, рассматриваемые как три числа N1, N2 и N, сами должны удовлетворять отношению **сумма1**, выдавая влево некоторый перенос C2 и оставляя некоторое подмножество неиспользованных цифр **Циф2**.

(b) Крайние левые цифры D1, D2 и D, а также перенос C2 должны удовлетворять отношению, показанному на рис. 7.1: C2, D1 и D2 складываются, давая в результате D и перенос влево. Это условие в нашей программе формулируется в виде отношения **суммацифр**.

Переводя это на Пролог, получаем:

```
сумма1([D1|N1],[D2|N2],[D|N],C1,C,Циф1,Циф) :-
    сумма1( N1, N2. N, C1, C2, Циф1, Циф2),
    суммацифр( D1, D2, C2, D, C, Циф2, Циф).
```

Осталось только описать на Прологе отношение **суммацифр**. В его определении есть одна тонкая деталь, касающаяся применения металогического предиката **nonvar**. D1, D2 и D должны быть десятичными цифрами. Если хоть одна из этих переменных еще не конкретизирована, ее нужно конкретизировать какой-нибудь цифрой из списка **Циф2**. Как только такая конкретизация произошла, эту цифру нужно удалить из множества доступных цифр. Если D1, D2 и D уже конкретизированы, тогда, конечно, ни одна из доступных цифр «потрачена» не будет. В программе эти действия реализуются при помощи недетерминированного вычеркивания элемента списка. Если этот элемент – не переменная, ничего не вычеркивается (конкретизации не было). Вот эта программа:

```
удалить( Элемент, Список, Список) :-
    nonvar( Элемент), !.

удалить( Элемент, [Элемент | Список ], Список).

удалить(Элемент, [A | Список], [A | Список1]) :-
    удалить( Элемент, Список, Список1).
```

Полная программа для решения арифметических ребусов приводится на рис. 7.2. В программу вклю-

чены также определения двух ребусов. Вопрос к пролог-системе для ребуса про DONALD'a, GERALD'a и ROBERT'a с использованием этой программы выглядит так:

```
?- ребус1( N1, N2, N), сумма( N1, N2, N).
```

```
% Решение числовых ребусов
сумма( N1, N2, N) :-
                % Числа представлены в виде списков цифр
    сумма1( N1, N2, N,
            0, 0,
                % Перенос справа и перенос влево равны 0
            [0,1,2,3,4,5,6,7,8,9], _).
                % Все цифры доступны

сумма1( [], [], [], 0, 0, Цифры, Цифры).

сумма1( [D1|N1],[D2|N2],[D|N],C1,C,Циф1,Циф) :-
    сумма1( N1, N2. N, C1, C2, Циф1, Циф2),
    суммацифр( D1, D2, C2, C, Циф2, Циф).

суммацифр( D1, D2, C1, D, C, Циф1, Циф) :-
    удалить( D1, Циф1, Циф2),
                % Выбор доступной цифры для D1
    удалить( D2, Циф2, Циф3),
                % Выбор доступной цифры для D2
    удалить( D, Циф3, Циф),
                % Выбор доступной цифры для D
    S is D1 + D2 + C1,
    D is S mod 10,
    C is S div 10.

удалить( A, L, L) :-
    nonvar( A), !.
                % Переменная A уже конкретизирована

удалить( A, [A | L], L).

удалить( A, [B | L], [B | L1]) :-
    удалить( A, L, L1).
```

```
% Примеры ребусов

ребус1( [D,O,N,A,L,D],
        [G,E,R,A,L,D],
        [R,O,B,E,R,T].

ребус2( [0,S,E,N,D],
        [0,M,O,R,E],
        [M,O,N,E,Y].
```

Рис. 7.2. Программа для арифметических ребусов.

Иногда этот ребус упрощают, сообщая часть решения в виде дополнительного ограничения, например D равно 5. В такой форме ребус можно передать пролог-системе при помощи сумма1:

```
? - сумма1( [5,O,N,A,L,5],
            [G,E,R,A,L,5],
            [R,O,B,E,R,T],
            0, 0, [0,1,2,3,4,6,7,8,9], _).
```

Интересно, что в обоих случаях существует только одно решение, т.е. только один способ заменить буквы цифрами.

Упражнения

7.1. Напишите процедуру **упростить** для упрощения алгебраических сумм, в которых участвуют числа и символы (строчные буквы). Пусть эта процедура переупорядочивает слагаемые так, чтобы символы предшествовали числам. Вот примеры ее использования:

```
?- упростить( 1 + 1 + a, E).
E = a + 2

?- упростить( 1 + a + 4 + 2 + b + c, E).
E = a + b + c + 7
```

```
?- упростить( 3 + x + x, E).
E = 2*x + 3
```

7.2. Определите процедуру

добавить(Элемент, Список)

для добавления нового элемента в список. Предполагается, что все элементы, хранящиеся в списке, – атомы. **Список** состоит из всех хранящихся в нем элементов, а за ними следует хвост, который не конкретизирован и служит для принятия новых элементов. Пусть, например, в списке уже хранятся **a**, **b** и **c**, тогда

Список=[a,b,c | Хвост]

где **Хвост** – переменная. Цель

добавить(d, Список)

вызовет конкретизацию

Хвост=[d|НовыйХвост] и
Список=[a,b,c,d|НовыйХвост]

Таким способом структура может наращиваться, включая в себя новые элементы. Определите также соответствующее отношение принадлежности.

7.2. Создание и декомпозиция термов: =.., *functor, arg, name*

Имеются три встроенные предиката для декомпозиции и синтеза термов: **functor**, **arg** и **=..** . Рассмотрим сначала отношение **=..**, которое записывается как инфиксный оператор. Цель

Терм =.. L

истинна, если L – список, начинающийся с главного функтора терма **Терм**, вслед за которым идут его аргументы. Вот примеры:

```
?- f( a, b) =.. L.
L = [f, a, b]

?- T =.. [прямоугольник, 3, 5].
T = прямоугольник( 3, 5)

?- Z =.. [p, X, f(X,Y) ].
Z = p( X, f(X,Y) )
```

Зачем может понадобиться разбирать терм на составляющие компоненты – функтор и его аргументы? Зачем создавать новый терм из заданного функтора и аргументов? Следующий пример показывает, что это действительно нужно.

Рассмотрим программу, которая манипулирует геометрическими фигурами. Фигуры – это квадраты, прямоугольники, треугольники, окружности и т.д. В программе их можно представлять в виде термов, функтор которых указывает на тип фигуры, а аргументы задают ее размеры:

```
квадрат( Сторона)
треугольник( Сторона1, Сторона2, Сторона3)
окружность( R)
```

Одной из операций над такими фигурами может быть увеличение. Его можно реализовать в виде трехаргументного отношения

```
увел( Фиг, Коэффициент, Фиг1)
```

где **Фиг** и **Фиг1** – геометрические фигуры одного типа (с одним и тем же функтором), причем параметры **Фиг1** равны параметрам **Фиг**, умноженным на **Коэффициент**. Для простоты будем считать, что все параметры **Фиг**, а также **Коэффициент** уже известны, т. е. конкретизированы числами. Один из способов программирования отношения **увел** таков:

```
увел( квадрат( A), F, квадрат( A1) ) :-
    A1 is F*A

увел( окружность( R), F, окружность( R1) ) :-
    R1 is F*R1

увел(прямоугольник(A,B),F,прямоугольник(A1,B1)):-
    A1 is F*A, B1 is F*B.
```

Такая программа будет работать, однако она будет выглядеть довольно неуклюже при большом количестве

различных типов фигур. Мы будем вынуждены заранее предвидеть все возможные типы, которые могут когда-либо встретиться. Придется заготовить по предложению на каждый тип, хотя во всех этих предложениях по существу говорится одно и то же: возьми параметры исходной фигуры, умножь их на коэффициент и создай фигуру того же типа с этими новыми параметрами.

Ниже приводится программа, в которой делается попытка (неудачная) справиться для начала хотя бы со всеми одиопараметрическими фигурами при помощи одного предложения:

```
увел( Тип( Пар), F, Тип( Пар1) ):-
    Пар1 is F*Пар.
```

Однако в Прологе подобные конструкции, как правило, запрещены, поскольку функтор должен быть атомом, и, следовательно, переменная **Тип** синтаксически не будет воспринята как функтор. Правильный метод – воспользоваться предикатом '=..'. Тогда процедура **увел** будет иметь обобщенную формулировку, пригодную для фигур любых типов:

```
увел( Фиг, F, Фиг1) :-
    Фиг =.. [Тип | Параметры],
    умножспис( Параметры, F, Параметры1),
    Фиг1 =.. [Тип | Параметры)].

умножспис( [], _, []).

умножспис( [X | L], F, [X1 | L1] ) :-
    X1 is F*X, умножспис( L, F, L1).
```

Наш следующий пример использования предиката '=..' связан с обработкой символьных выражений (формул), где часто приходится подставлять вместо некоторого подвыражения другое выражение. Мы определим отношение

подставить(Подтерм, Терм, Подтерм1, Терм1)

следующим образом: если все вхождения **Подтерм**'а в **Терм** заменить на **Подтерм1**, то получится **Терм1**. Например:

```
?- подставить( sin(x),2*sin(x)*f(sin(x)),t,F ).
F = 2*t*f(t)
```

Под «вхождением» **Подтерм**'а в **Терм** мы будем пони-

мать такой элемент **Терм**'а, который *сопоставим* с **Подтерм**'ом. Вхождения будем искать сверху вниз. Поэтому цель

?- подставить(a+b, f(a, A+B), v, F).

даст результат

F = f(a, v)		**F = f(a, v+v)**
A = a	а не	**A = a+b**
B = b		**B = a+b**

При определении отношения **подставить** нам нужно рассмотреть несколько случаев и для каждого принять свое решение:

если **Подтерм** = **Терм**, то **Терм1** = **Подтерм1**;
иначе если **Терм** – «атомарный» (не структура),
то **Терм1** = **Терм** (подставлять нечего),
иначе подстановку нужно выполнить над
аргументами **Терм**'а.

Эти правила можно превратить в программу, показанную на рис. 7.3.

Термы, полученные при помощи предиката '=..', разумеется, можно использовать и в качестве целей. Это дает возможность программе в процессе вычислений самой порождать и вычислять цели, структура которых не обязательно была известна заранее в момент написания программы. Последовательность целей, иллюстрирующая этот прием, могла бы выглядеть примерно так:

получить(Функтор),
вычислить(Списарг),
Цель =.. [Фуиктор | Списарг],
Цель

Здесь **получить** и **вычислить** – некоторые определенные пользователем процедуры, предназначенные для вычисления компонент цели. После этого цель порождается предикатом '=..', а затем активизируется при помощи простого указания ее имени **Цель**.

Некоторые реализации Пролога могут содержать требование, чтобы все цели, появляющиеся в программе, по своей *синтаксической* форме были либо атомами, либо структурами с атомом в качестве главного функтора. Поэтому переменная, вне

```
% Отношение
%
% подставить( Подтерм, Терм, Подтерм1, Терм1)
%
% состоит в следующем: если все вхождения Подтерм'а в Терм
% заменить на Подтерм1, то получится Терм1.

% Случай 1: Заменить весь терм
подставить( Терм, Терм, Терм1, Терм1) :- !.

% Случай 2: нечего подставлять
подставить( _, Терм, _, Терм) :-
    atomic( Терм), !.

% Случай 3: Проделать подстановку в аргументах
подставить( Под, Терм, Под1, Терм1) :-
    Терм =.. [F | Арги],
                % Выделить аргументы
    подспис( Под, Арги, Под1, Арги1),
                % Выполнить над ними подстановку
    Терм1 =.. [F | Арги1].

подспис(Под,[Терм|Термы],Под1,[Терм1|Термы1]) :-
  подставить( Под, Терм, Под1, Терм1),
  подспис( Под, Термы, Под1, Термы1).
```

Рис. 7.3. Процедура подстановки в терм вместо одного из его подтермов некоторого другого подтерма.

зависимости от ее текущей конкретизации, может по своей синтаксической форме не подойти в качестве цели. Эту трудность можно обойти при помощи еще одного встроенного предиката call (вызов), чьим аргументом является цель, подлежащая вычислению. В соответствии с этим предыдущий пример должен быть переписан так:

```
...
Цель = [Функтор | Списарг],
call( Цель)
```

Иногда нужно извлечь из терма только его главный функтор или один из аргументов. В этом случае

можно, конечно, воспользоваться отношением '=..' Но более аккуратным и практичным, а также и более эффективным способом будет применение одной из двух новых встроенных процедур: **functor** и **arg**. Вот их смысл: цель

fuctor(Терм, F, N)

истинна, если F – главный функтор Терм'а, а N – арность F. Цель

arg(N, Терм, A)

истинна, если A – N-й аргумент в Терм'е, в предположении, что нумерация аргументов идет слева направо и начинается с 1. Примеры для иллюстрации:

```
?- functor( t( f(x), X, t), Фуи, Арность).

Фуи = t
Арность = 3

?- arg( 2, f( X, t(a), t(b) ), Y).

Y = t(a)

?- functor( D, дата, 3),
     arg( 1, D, 29),
     arg( 2, D, июнь),
     arg( 3, D, 1982).

D = дата( 29, июнь, 1982)
```

Последний пример иллюстрирует особый случай применения предиката **functor**. Цель **functor(D, дата, 3)** создает «обобщенный» терм с главным функтором **дата** и тремя аргументами. Этот терм обобщенный, так как все три его аргумента – неконкретизированные переменные, чьи имена генерируются пролог-системой. Например:

D = дата(_5, _6, _7)

Затем эти три переменные конкретизируются при помощи трех целей **arg**.

К рассматриваемому множеству встроенных предикатов относится также и введенный в гл. 6 предикат **name**, предназначенный для синтеза и декомпозиции атомов. Для полноты изложения мы здесь напомним его смысл. Цель

name(A, L)

истинна, если L – список кодов (в кодировке ASCII) символов, входящих в состав атома A.

Упражнения

7.3. Определите предикат **конкрет(Терм)** так, чтобы он принимал значение истина, когда в **Терм**'е нет ни одной неконкретизированной переменной.

7.4. Процедура **подставить** из данного раздела производит, при наличии разных вариантов, лишь самую «внешнюю» подстановку.
Модифицируйте эту процедуру так, чтобы она находила все возможные варианты при помощи автоматического перебора. Например:

```
?- подставить( a+b, f(A+B), новый, НовыйТерм).

A = a
B = b
НовыйТерм = f( новый);

A = a+b
B = a+b
НовыйТерм = f( новый + новый)
```

Наша исходная версия нашла бы только первый из этих двух ответов.

7.5. Определите отношение

включает(Терм1, Терм2)

которое выполняется, если **Терм1** является более общим, чем **Терм2**. Например:

```
?- включает( X, c).
yes

?- включает( g(X), g(t(Y))).
yes

?- включает( f(X,X), f(a,b)).
no
```

7.3. Различные виды равенства

В каких случаях мы считаем, что два терма равны? До сих пор мы рассматривали три вида равенства в Прологе. Первый был связан с сопоставлением и записывался так:

```
X = Y
```

Это равенство верно, если X и Y сопоставимы. Следующий вид равенства записывался в виде

```
X is E
```

Такое равенство выполняется, если X сопоставим со значением арифметического выражения E. Мы также рассматривали равенства вида

```
E1 =:= E2
```

которые верны, если равны значения арифметических выражений E1 и E2. Наоборот, если значения двух арифметических выражений не равны, мы пишем

```
E1 =/= E2
```

Иногда нам может понадобиться более строгий вид равенства – *буквальное равенство* двух термов. Этот вид реализован еще одним встроенным предикатом, записываемым как инфиксный оператор '==':

```
T1 == T2
```

Это равенство выполняется, если термы T1 и T2 идентичны, т. е. имеют в точности одинаковую структуру, причем все соответствующие компоненты совпадают. В частности, должны совпадать и имена переменных. Отношение «не идентичны», дополнительное к данному, записывается так:

```
T1 \== T2
```

Приведем несколько примеров:

```
?- f( a, b) == f( a, b).
yes

?- f( a, b) == f( a, X).
no
```

```
?- f( a, X) == f( a, Y).
no
?- X \== Y.
yes
?- t( X, f( a, Y) ) == t( X, f( a, Y) ).
yes
```

Давайте в качестве примера переопределим отношение

счетчик(Терм, Список, N)

из разд. 7.1. Пусть на этот раз N будет числом буквальных вхождений Терм'а в Список:

```
счетчик( _, [], 0).
счетчик( Терм, [Голова | L], N) :-
    Терм == Голова, !,
    счетчик( Терм, L, N1),
    N is N1 + 1;
    счетчик( Терм, L, N).
```

7.4. Работа с базой данных

Реляционная модель предполагает, что база данных – это описание некоторого множества отношений. Пролог-программу можно рассматривать как именно такую базу данных: описание отношений частично присутствует в ней в явном виде (факты), а частично – в неявном (правила). Более того, встроенные предикаты дают возможность корректировать эту базу данных в процессе выполнения программ. Это делается добавлением к программе (в процессе вычисления) новых предложений или же вычеркиванием из нее уже существующих. Предикаты, используемые для этой цели, таковы: **assert** (добавить), **asserta**, **assertz** и **retract** (удалить).

Цель

assert(C)

всегда успешна, а в качестве своего побочного эффекта вызывает «констатацию» предложения C, т. е.

добавление его к базе даных. Цель

retract(C)

приводит к противоположному эффекту: удаляет предложение, сопоставимое с С. Следующий диалог иллюстрирует их работу:

```
?- кризис.
no
? - assert( кризис).
yes
?- кризис.
yes
? - retract( кризис).
yes
?- кризис.
no
```

Предложения, добавленные к программе таким способом, ведут себя точно так же, как и те, что были в «оригинале» программы. Следующий пример показывает, как с помощью **assert** и **retract** можно работать в условиях изменяющейся обстановки. Предположим, что у нас есть такая программа о погоде:

```
хорошая :-
    солнечно, not дождь.

необычная :-
    солнечно, дождь.

отвратительная :-
    дождь, туман.

дождь.

туман.
```

Ниже приводится пример диалога с этой программой, во время которого база даниых постепенно изменяется:

```
?- хорошая.
no
?- отвратительная.
yes
```

```
?- retract( туман).
yes
?- отвратительная.
no
?- assert( солнечно).
yes
?- необычная.
yes
?- retract( дождь).
yes
?- хорошая.
yes
```

Добавлять и удалять можно предложения любой формы. Следующий пример показывает, что, кроме того, retract может работать недетерминированно: используя механизм возвратов с помощью только одной цели retract можно удалить целое множество предложений. Предположим, что в программе, с которой мы «консультируемся», есть такие факты:

```
быстр( эни).
медл( том).
медл( пат).
```

К этой программе можно добавить правило:

```
?- assert(
          ( быстрее( X, Y) :-
              быстр( X), медл( Y) ) ).
  yes

  ?- быстрее( A, B).
  A = эни
  B = том

  ?- retract( медл( X) ).
  X = том;
  X = пат;
  no

  ?- быстрее( эни, _).
  no
```

Заметьте, что при добавлении нового правила синтаксис требует, чтобы оно (как аргумент **assert**) было заключено в скобки.

При добавлении нового предложения может возникнуть желание указать, на какое место в базе данных его следует поместить. Такую возможность обеспечивают предикаты **asserta** и **assertz**. Цель

asserta(C)

помещает C в начале базы данных. Цель

assertz(C)

- в конце. Вот пример, иллюстрирующий работу этих предикатов:

```
?- assert( p(a)), assertz( p(b) ), asserta( p(c) ).
yes
?- p( X).
X = c;
X = a;
X = b
```

Между **consult** и **assertz** существует связь. Обращение к файлу при помощи **consult** можно в терминах **assertz** определить так: считать все термы (предложения) файла и добавить их в конец базы данных.

Одним из полезных применений предиката **asserta** является накопление уже вычисленных ответов на вопросы. Пусть, например, в программе определен предикат

решить(Задача, Решение)

Мы можем теперь задать вопрос и потребовать, чтобы ответ на него был запомнен, с тем чтобы облегчить получение ответов на будущие вопросы:

```
?- решить( задача1, решение),
   asserta( решить( Задача1, Решение) ).
```

Если в первой из приведенных целей будет успех, ответ **(Решение)** будет сохранен, а затем использован так же, как и любое другое предложение, при ответе на дальнейшие вопросы. Преимущество такого «запоминания» состоит в том, что на дальнейшие вопросы, сопоставимые с добавленным фактом, ответ

будет получен, как правило, значительно быстрее, чем в первый раз. Ответ будет теперь получен как факт, а не как результат вычислений, требующих, возможно, длительного времени.

Развитие этой идеи состоит в использованин **assert** для порождения всех решений в виде таблицы фактов. Например, создать таблицу пронзведений всех чисел от 0 до 9 можно так: породить пару чисел X и Y, вычислнть Z, равное X * Y, добавить эти три числа в виде строки в таблицу произведений, а затем создать искусственно неуспех. Неуспех вызовет возврат, в результате которого будет найдена новая пара чисел, и в таблицу добавится новая строка и т.д. Эта идея реализована в процедуре

```
таблица :-
    L = [0,1,2,3,4,5,6,7,8,9],
    принадлежит( X, L),   % Выбрать первый сомножитель
    принадлежит( Y, L),   % Выбрать второй сомножитель
    Z is X*Y,
    assert( произв( X,Y,Z) ),
    fail.
```

Вопрос

```
?- таблица.
```

потерпит, конечно, неудачу, однако в качестве своего побочного эффекта приведет к добавлению в базу данных целой таблицы произведений. После этого можно, например, спросить, какие пары дают произведения, равные 8:

```
?- произв( A, B, 8).

A = 1
B = 8;

A = 2
B = 4;

...
```

Здесь следует сделать одно замечание, относящееся к стилю программирования. Приведенные примеры показали некоторые явно полезные применения **assert** и **retract**. Однако использование этих отношений требует особой внимательности. Не рекомендуется применять их слишком часто и без должной осторожности

– это плохой стиль программирования. Ведь добавляя и удаляя предложения, мы фактически изменяем программу. Поэтому отношения, выполнявшиеся в некоторой ее точке, могут оказаться неверными в другой. В разные моменты времени ответы на одни и те же вопросы будут различными. Таким образом, большое количество обращений к **assert** и **retract** может затемнить смысл программы и станет трудно разобрать, что истинно, а что – нет. В результате поведение программы может стать непонятным, трудно объяснимым, и вряд ли можно будет ей доверять.

Упражнения

7.6. (a) Напишите вопрос к пролог-системе, который удаляет из базы данных всю таблицу **произв**.
(b) Измените этот вопрос так, чтобы он удалил из таблицы только те строки, в которых произведение равно 0.

7.7. Определите отношение

копия(Терм, Копия)

которое порождает такую копию **Терм**'а **Копия**, в которой все переменные переименованы. Это легко сделать, используя **assert** и **retract**.

7.5. Средства управления

К настоящему моменту мы познакомились с большинством дополнительных средств управления, за исключением **repeat** (повторение). Здесь мы для полноты приводим список всех таких средств.

- *отсечение*, записывается как '!', предотвращает перебор, введено в гл. 5.
- **fail** – цель, которая всегда терпит неудачу.
- **true**– цель, которая всегда успешна.

- **not(P)** – вид отрицания, который всегда ведет себя в точном соответствии со следующим определением:

  ```
  not( P) :- P, !, fail; true.
  ```

  Некоторые проблемы, связанные с отсечением и not детально обсуждались в гл. 5.

- **call(P)** активизирует цель P. Обращение к call имеет успех, если имеет успех P.

- **repeat** – цель, которая всегда успешна. Ее особое свойство состоит в том, что она недетерминирована, поэтому всякий раз, как до нее доходит перебор, она порождает новую ветвь вычислений. Цель repeat ведет себя так, как если бы она была определена следующим образом:

  ```
  repeat.
  repeat :- repeat.
  ```

  Стандартный способ применения repeat показан в процедуре квадраты, которая читает последовательность чисел и выдает их квадраты. Последовательность чисел заканчивается атомом стоп, который служит для процедуры сигналом окончания работы.

  ```
  квадраты :-
      repeat,
      read( X),
      ( X = стоп, !;
        Y is X*X, write( Y), fail ).
  ```

7.6. *bagof, setof* и *findall*

При помощи механизма автоматического перебора можно получить один за другим все объекты, удовлетворяющие некоторой цели. Всякий раз, как порождается новое решение, предыдущее пропадает и становится с этого момента недоступным. Однако у нас может возникнуть желание получить доступ ко всем порожденным объектам сразу, например собрав

их в список. Встроенные предикаты **bagof** (набор) и **setof** (множество) обеспечивают такую возможность; вместо них иногда используют предикат **findall** (найти все).

Цель

bagof(X, P, L)

порождает список L всех объектов X, удовлетворяющих цели P. Обычно **bagof** имеет смысл применять только тогда, когда X и P содержат общие переменные. Например, допустим, что мы включили в программу следующую группу предложений для разбиения букв (из некоторого множества) на два класса – гласные и согласные:

```
класс( a, глас).
класс( b, согл).
класс( c, согл).
класс( d, согл).
класс( e, глас).
класс( f, согл).
```

Тогда мы можем получить список всех согласных, упомянутых в этих предложениях, при помощи цели:

```
?- bagof( Буква, класс( Буква, согл), Буквы).

Буквы = [d,c,d,f]
```

Если же мы в указанной цели оставим класс букв неопределенным, то, используя автоматический перебор, получим два списка букв, каждый из которых соответствует одному из классов:

```
?- bagof( Буква, класс( Буква, Класс), Буквы).

Класс = глас
Буквы = [a,e]

Класс = согл
Буквы = [b,c,d,f]
```

Если **bagof(X, P, L)** не находит ни одного решения для **P**, то цель **bagof** просто терпит неуспех. Если один и тот же X найден многократно, то все его экземпляры будут занесены в L, что приведет к появлению в L повторяющихся элементов.

Предикат **setof** работает аналогично предикату **bagof**. Цель

setof(X, P, L)

как и раньше, порождает список L объектов X, удовлетворяющих P. Только на этот раз список L будет упорядочен, а из всех повторяющихся элементов, если таковые есть, в него попадет только один. Упорядочение происходит по алфавиту или по отношению '<', если элементы списка – числа. Если элементы списка – структуры, то они упорядочиваются по своим главным функторам. Если же главные функторы совпадают, то решение о порядке таких термов принимается по их первым несовпадающим функторам, расположенным выше и левее других (по дереву). На вид объектов, собираемых в список, ограничений нет. Поэтому можно, например, составить список пар вида

Класс / Буква

при этом гласные будут расположены в списке первыми («глас» по алфавиту раньше «согл»):

```
?- setof( Класс/Буква, класс(Буква,Класс), Спис).
Спис = [глас/a,глас/e,согл/b,согл/c,согл/d,согл/f]
```

Еще одним предикатом этого семейства, аналогичным **bagof**, является **findall**.

findall(X, P, L)

тоже порождает список объектов, удовлетворяющих P. Он отличается от **bagof** тем, что собирает в список *все* объекты X, не обращая внимание на (возможно) отличающиеся для них конкретизации тех переменных из P, которых нет в X. Это различие видно из следующего примера:

```
?- findall( Буква, класс(Буква,Класс), Буквы).
Буквы= [a,b,c,d,e,f]
```

Если не существует ни одного объекта X, удовлетворяющего P, то **findall** все равно имеет успех и выдает L = [].

Если в используемой реализации Пролога отсутствует встроенный предикат **findall**, то его легко запрограммировать следующим образом. Все решения для P порождаются искусственно вызываемыми возвра-

тами. Каждое решение, как только оно получено, немедленно добавляется к базе данных, чтобы не потерять его после нахождения следующего решения. После того, как будут получены и сохранены все решения, их нужно собрать в список, а затем удалить из базы данных при помощи **retract**. Весь процесс можно представлять себе как построение очереди из порождаемых решений. Каждое вновь порождаемое решение добавляется в конец этой очереди при помощи **assert**. Когда все решения собраны, очередь расформировывается. Заметим также, что конец очереди надо пометить, например, атомом «дно» (который, конечно, должен отличаться от любого ожидаемого решения). Реализация **findall** в соответствии с описанным методом показана на рис. 7.4.

```
findall( X, Цель, ХСпис) :-
    call( Цель),                    % Найти решение
    assert( очередь( X) ),          % Добавить его
    fail;           % Попытаться найти еще решения
    assertz( очередь( дно) ),
                    % Пометить конец решений
    собрать( ХСпис).                % Собрать решения в список

собрать( L) :-
    retract( очередь(X) ), !,
                    % Удалить следующее решение
    ( X == дно, !, L = [];
                    % Конец решений?
    L = [X | Остальные], собрать( Остальные) ).
                    % Иначе собрать остальные
```

Рис. 7.4. Реализация отношения findall.

Упражнения

7.8. Используя **bagof**, определите отношение

множподмножеств(Мн, Подмн)

для вычисления множества всех подмножеств данного множества (все множества представлены списками).

7.9. Используя bagof, определите отношение

копия(Терм, Копия)

чтобы **Копия** представляла собой **Терм**, в котором все переменные переименованы.

Резюме

- В любой реализации Пролога обычно предусматривается набор встроенных процедур для выполнения различных полезных операций, несуществующих в чистом Прологе. В данной главе мы рассмотрели подобное множество предикатов, имеющееся во многих реализациях.
- Тип терма можно установить при помощи следующих предикатов:

var(X)	X – (неконкретизированная) переменная
nonvar(X)	X – не переменная
atom(X)	X – атом
integer(X)	X – целое
atomic(X)	X – или атом, или целое

- Термы можно синтезировать или разбирать на части:

Терм =.. [Функтор | СписокАргументов]
functor(Терм, Функтор, Арность)
arg(N, Терм, Аргумент)
name(атом, КодыСимволов)

- Программу на Прологе можно рассматривать как реляционную базу данных, которую можно изменять при помощи следующих процедур:

assert(Предл)	добавляет предложение **Предл** к программе
asserta(Предл)	добавляет в начало
assertz(Предл)	добавляет в конец
retract(Предл)	удаляет предложение, сопоставимое с предложением **Предл**

- Все объекты, отвечающие некоторому заданному условию, можно собрать в список при помощи предикатов:

bagof(X, P, L) L – список всех X, удовлетворяющих условию P

setof(X, P, L) L – отсортированный список всех X, удовлетворяющих условию P

findall(X, P, L) аналогичен **bagof**

- **repeat** – средство управления, позволяющее порождать неограниченное число альтернатив для автоматического перебора.

8 СТИЛЬ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

В этой главе мы рассмотрим некоторые общие принципы хорошего программирования и обсудим, в частности, следующие вопросы: «Как представлять себе прологовские программы? Из каких элементов складывается хороший стиль программирования на Прологе? Как отлаживать пролог-программы? Как повысить их эффективность?»

8.1. Общие принципы хорошего программирования

Главный вопрос, касающийся хорошего программирования, – это вопрос о том, что такое хорошая программа. Ответ на этот вопрос не тривиален, поскольку существуют разные критерии качества программ.

Следующие критерии общепринятые:

- *Правильность.* Хорошая программа в первую очередь должна быть правильной, т. е. она должна делать именно то, для чего предназначалась. Это требование может показаться тривиальным и самоочевидным. Однако в случае сложных программ правильность достигается не так часто. Распространенной ошибкой при написании программ является пренебрежение этим очевидным критерием, когда большее внимание уделяется другим критериям – таким, как эффективность.
- *Эффективность.* Хорошая программа не должна

попусту тратить компьютерное время и память.

- *Простота, читабельность.* Хорошая программа должна быть легка для чтения и понимаиия. Она не должна быть более сложной, чем это необходимо. Следует избегать хитроумных программистских трюков, затемняющих смысл программы. Общая организация программы и расположение ее текста должны облегчать ее понимание.

- *Удобство модификации.* Хорошая программа должна быть легко модифицируема и расширяема. Простота и модульная организация программы облегчают внесение в нее изменений.

- *Живучесть.* Хорошая программа должна быть живучей. Она не должна сразу «ломаться», если пользователь введет в нее неправильные или непредусмотренные данные. В случае подобных ошибок программа должна сохранять работоспособность и вести себя разумно (сообщать об ошибках).

- *Документированность.* Хорошая программа должна быть хорошо документирована. Минимальная документация – листинг с достаточно подробными комментариями.

Степень важиости того или иного критерия зависит от конкретной задачи, от обстоятельств иаписания программы, а также от условий ее эксплуатации. Наивысшим приоритетом пользуется, без сомнеиия, правильность. Обычно простоте, удобству модификации, живучести и документированности придают по крайней мере не меньший приоритет, чем эффективности.

Существует несколько общих соображений, помогающих реализовать вышеупомянутые критерии на практике. Одно важное правило состоит в том, чтобы сначала *продумать* задачу, подлежащую решению, и лишь затем приступать к иаписанию текста программы иа конкретиом языке программироваиия. Как только мы хорошо поймем задачу, и способ ее решения будет нами полиостью и во всех деталях продуман, само программироваиие окажется быстрым и легким делом и появится неплохой шаис за короткое время получить правильную программу.

Распространенной ошибкой является попытка начать писать программу даже до того, как была уяснена полная постановка задачи. Главная причина, по которой следует воздерживаться от преждевременного начала программирования, состоит в том, что обдумывание задачи и поиск метода ее решения должны проводиться в терминах, наиболее адекватных самой этой задаче. Эти термины чаще всего далеки от синтаксиса применяемого языка программирования и могут быть утверждениями на естественном языке и рисунками.

Исходная формулировка способа решения задачи должна быть затем трансформирована в программу, но этот процесс трансформации может оказаться нелегким. Неплохим подходом к его осуществлению является применение принципа *пошаговой детализации*. Исходная формулировка рассматривается как «решение верхнего уровня», а окончательная программа – как «решение низшего уровня».

В соответствии с принципом пошаговой детализации окончательная программа получается после серии трансформаций или «детализаций» решения. Мы начинаем с первого решения – решения верхнего уровня, а затем последовательно проходим по цепочке решений; все эти решения эквивалентны, но каждое следующее решение выражено более детально, чем предыдущее. На каждом шагу детализации понятия, использовавшиеся в предыдущих формулировках, прорабатываются более подробно, а их представление все более приближается к языку программирования. Следует отдавать себе отчет в том, что детализация касается не только процедур, но и структур данных. На начальных шагах работают обычно с более абстрактными, более крупными информационными единицами, детальная структура которых уточняется впоследствии.

Стратегия нисходящей пошаговой детализации имеет следующие преимущества:

- она позволяет сформулировать грубое решение в терминах, наиболее адекватных решаемой задаче;
- в терминах таких мощных понятий решение будет сжатым и простым, а потому скорее всего правильным;

- каждый шаг детализации должен быть достаточно малым, чтобы не представлять больших интеллектуальных трудностей; если это удалось – трансформация решения в новое, более детальное представление скорее всего будет выполнена правильно, а следовательно, таким же правильным окажется и полученное решение следующего шага детализации.

В случае Пролога мы можем говорить о пошаговой детализации *отношений*. Если существо задачи требует мышления в алгоритмических терминах, то мы можем также говорить и о детализации *алгоритмов*, приняв процедурную точку зрения на Пролог.

Для того, чтобы удачно сформулировать решение на некотором уровне детализации и придумать полезные понятия для следующего, более низкого уровня, нужны идеи. Поэтому программирование – это творческий процесс, что верно в особенности, когда речь идет о начинающих программистах. По мере накопления опыта работа программиста постепенно становится все менее искусством и все более ремеслом. И все же главным остается вопрос: как возникают идеи? Большинство идей приходит из опыта, из тех задач, решения которых уже известны. Если мы не знаем прямого решения задачи, то нам может помочь уже решенная задача, похожая на нашу. Другим источником идей является повседневная жизнь. Например, если необходимо запрограммировать сортировку списка, то можно догадаться, как это сделать, если задать себе вопрос: «А как бы я сам стал действовать, чтобы расположить экзаменационные листы студентов по их фамилиям в алфавитном порядке?»

Общие принципы, изложенные в данном разделе, известны также как составные части «структурного программирования»; они, в основном, применимы и к программированию на Прологе. В следующих разделах мы обсудим их более детально, обращая особое внимание на применение этих принципов программирования к Прологу.

8.2. Как представлять себе программы на Прологе

Одной из характерных особенностей Пролога является то, что в нем допускается как процедурный, так и декларативный стиль мышления при составлении программы. Эти два подхода детально обсуждались в гл. 2 и затем многократно иллюстрировались на примерах. Какой из этих подходов окажется более эффективным и практичным, зависит от конкретной задачи. Обычно построение декларативного решения задачи требует меньших усилий, но может привести к неэффективной программе.

В процессе построения решения мы должны сводить задачу к одной или нескольким более легким подзадачам. Возникает важный вопрос: как находить эти подзадачи? Существует несколько общих принципов, которые часто применяются при программировании на Прологе. Они будут обсуждаться в следующих разделах.

8.2.1. Использование рекурсии

Этот принцип состоит в том, чтобы разбить задачу на случаи, относящиеся к двум группам:

(1) тривиальные, или «граничные» случаи;
(2) «общие» случаи, в которых решение получается из решений для (более простых) вариантов самой исходной задачи.

Этот метод мы использовали в Прологе постоянно. Рассмотрим еще один пример: обработка списка элементов, при которой каждый элемент преобразуется по одному и тому же правилу. Пусть это будет процедура

преобрспис(Спис, F, НовСпис)

где **Спис** – исходный список, **F** – правило преобразования (бинарное отношение), а **НовСпис** – список всех преобразованных элементов. Задачу преобразо-

вания списка **Спис** можно разбить на два случая:

(1) Граничный случай: **Спис = []**

Если **Спис = []**, то **НовСпис = []**, независимо от **F**

(2) Общий случай: **Спис = [X | Хвост]**

Чтобы преобразовать список вида **[X|Хвост]**, необходимо:

преобразовать список **Хвост**; результат – **НовХвост**;
преобразовать элемент **X** по правилу **F**; результат – **НовX**;
результат преобразования всего списка – **[НовX|НовХвост]**.

Тот же алгоритм, изложенный на Прологе:

```
преобрспис( [], _, []).
преобрспис( [X|Хвост], F, [НовX|НовХвост] :-
    G =.. [F, X, НовX],
    call( G),
    преобрспис( Хвост, F, НовХвост).
```

Одна из причин того, что рекурсия так естественна для определения отношений на Прологе, состоит в том, что объекты данных часто сами имеют рекурсивную структуру. К таким объектам относятся списки и деревья. Список либо пуст (граничный случай), либо имеет голову и хвост, который сам является списком (общий случай). Двоичное дерево либо пусто (граничный случай), либо у него есть корень и два поддерева, которые сами являются двоичными деревьями (общий случай). Поэтому для обработки всего непустого дерева необходимо сначала что-то сделать с его корнем, а затем обработать поддеревья.

8.2.2. Обобщение

Часто бывает полезно обобщить исходную задачу таким образом, чтобы полученная более общая задача допускала рекурсивную формулировку. Исходная зада-

ча решается тогда как частный случай ее более общего варианта. Обобщение отношения обычно требует введения одного или более дополнительных аргументов. Главная проблема состоит в отыскании подходящего обобщения, что может потребовать более тщательного изучения задачи.

В качестве примера рассмотрим еще раз задачу о восьми ферзях. Исходная задача состояла в следующем: разместить на доске восемь ферзей так, чтобы обеспечить отсутствие взаимных нападений. Соответствующее отношение назовем

восемьферзей(Поз)

Оно выполняется (истинно), если **Поз** – представленная тем или иным способом позиция, удовлетворяющая условию задачи. Можно предложить следующую полезную идею: обобщить задачу, перейдя от 8 ферзей к произвольному количеству – N. Количество ферзей станет дополнительным аргументом:

n_ферзей(Поз, N)

Преимущество такого обобщения состоит в том, что отношение **n_ферзей** допускает непосредственную рекурсивную формулировку:

(1) Граничный случай: N = 0
Разместить 0 ферзей – тривиальная задача.

(2) Общий случай: N > 0
Для «безопасного» размещения N ферзей необходимо:

- получить требуемое размещение для (N – 1) ферзей и
- добавить оставшегося ферзя так, чтобы он не бил ни одного из уже поставленных ферзей.

Как только мы научимся решать более общую задачу, решить исходную уже не составит труда:

восемьферзей(Поз) :- n_ферзей(Поз, 8)

8.2.3. Использование рисунков

В поиске идей для решения задачи часто бывает полезным обратиться к ее графическому представлению. Рисуиок может помочь выявить в задаче иекоторые существеиные отношения. После этого остаиется только описать на языке программироваиия то, что мы *видим* иа рисунке.

Использование графического представлення при решении задач полезно всегда, однако похоже, что в Прологе оно работает особенно хорошо. Происходит это по следующим причинам:

(1) Пролог особеино хорошо приспособлеи для задач, в которых фигурируют объекты и отношения между иими. Часто такие задачи естественно иллюстрировать графами, в которых узлы соответствуют объектам, а дуги – отношениям.

(2) Естествеиным наглядным изображеннем структурных объектов Пролога являются деревья.

(3) Декларативный характер пролог-программ облегчает перевод графического представления на Пролог. В принципе, порядок описания «картинки» не играет роли, мы просто помещаем в программу то, что видим, в произвольном порядке. (Возможно, что из практических соображеиий этот порядок впоследствии придется подправить с целью повысить эффективность программы.)

8.3. Стиль программирования

Подчиияться при программировании некоторым стнлистическим соглашениям нужно для того, чтобы

- уменьшить опасность виесения ошибок в программы и
- создавать программы, которые легко читать, поиимать, отлаживать и модифицировать.

Ниже дается обзор некоторых из составных частей хорошего стиля программирования на Прологе. Мы рассмотрим некоторые общие правила хорошего стиля, табличную организацию длинных процедур и вопросы комментирования программ.

8.3.1. Некоторые правила хорошего стиля

- Предложения программы должны быть короткими. Их тела, как правило, должны содержать только несколько целей.
- Процедуры должны быть короткими, поскольку длинные процедуры трудны для понимания. Тем не менее длинные процедуры вполне допустимы в том случае, когда они имеют регулярную структуру (этот вопрос еще будет обсуждаться в данной главе).
- Следует применять мнемонические имена процедур и переменных. Они должны отражать смысл отношений и роль объектов данных.
- Существенное значение имеет расположение текста программы. Для улучшения читабельности программы нужно постоянно применять пробелы, пустые строки и отступы. Предложения, относящиеся к одной процедуре, следует размещать вместе в виде отдельной группы строк; между предложениями нужно вставлять пустую строку (этого не нужно делать, возможно, только в случае перечисления большого количества фактов, касающихся одного отношения); каждую цель можно размещать на отдельной строке. Пролог-программы иной раз напоминают стихи по эстетической привлекательности своих идей и формы.
- Стилистические соглашения такого рода могут варьироваться от программы к программе, так как они зависят от задачи и от личного вкуса. Важно, однако, чтобы на протяжении одной программы постоянно применялись одни и те же соглашения.

- Оператор отсечения следует применять с осторожностью. Если легко можно обойтись без него – не пользуйтесь им. Всегда, когда это возможно, предпочтение следует отдавать «зеленым отсечениям» перед «красными». Как говорилось в гл. 5, отсечение называется «зеленым», если его можно убрать, не затрагивая декларативный смысл предложения. Использование «красных отсечений» должно ограничиваться четко определенными конструкциями, такими как оператор **not** или конструкция выбора между альтернативами. Примером последней может служить

  если **Условие** *то* **Цель1** *иначе* **Цель2**

  С использованием отсечения эта конструкция переводится на Пролог так:

```
Условие, !.        % Условие выполнено?
Цель1;             % Если да, то Цель1
Цель2              % Иначе - Цель2
```

- Из-за того, что оператор **not** связан с отсечением, он тоже может привести к неожиданностям. Поэтому, применяя его, следует всегда помнить точное прологовское определение этого оператора. Тем не менее если приходится выбирать между **not** и отсечением, то лучше использовать **not**, чем какую-нибудь туманную конструкцию с отсечением.

- Внесение изменений в программу при помощи **assert** и **retract** может сделать поведение программы значительно менее понятным. В частности, одна и та же программа на одни и те же вопросы будет отвечать по-разному в разные моменты времени. В таких случаях, если мы захотим повторно воспроизвести исходное поведение программы, нам придется предварительно убедиться в том, что ее исходное состояние, нарушенное при обращении к **assert** и **retract**, полностью восстановлено.

- Применение точек с запятой может затемнять смысл предложений. Читабельность можно иногда улучшить, разбивая предложения, содержа-

щне точки с запятой, на несколько новых предложенний, однако за это, возможно, прндется заплатить увеличением длины программы н потерей в ее эффективности.

Для иллюстрации некоторых положеиий даиного раздела рассмотрим отношение

слить(Спнс1, Спнс2, Спнс3)

где **Спнс1** и **Спнс2** – упорядоченные спнски,а **Спнс3** – результат их слияиия (тоже упорядочениый). Например:

слить([2,4,7], [1,3,4,8], [1,2,3,4,4,7,8])

Вот стнлистически иеудачная реализация этого отношенпя:

```
слить( Спнс1, Спнс2, Спнс3) :-
    Спнс1 = [], !, Спнс3 = Спнс2;
                          % Первый список пуст
    Спнс2 = [], !, Спнс3 = Спнс1;
                          % Второй список пуст
    Спнс1 = [X | Остальные],
    Спнс2 = [Y | Остальные],
    ( X < Y, !,
    Z = X,                % Z - голова Спнс3
    слить( Остальные1, Спнс2, Остальные3 );
    Z = Y,
    слить( Спнс1, Остальные2, Остальные3 ) ),
    Спнс3 = [Z | Остальные3].
```

Вот более предпочтительный вариаит, не использующнй точек с запятой:

```
слить( [], Спнс, Спнс).

слить( Спнс, [], Спнс).

    слить( [X | Остальные1], [Y | Остальные2],
                          [X | Остальные3] ) :-
    X < Y, !,
    слить(Остальные1,[Y|Остальные2],Остальные3).

    слить(Спнс1,[Y|Остальные2],[Y|Остальные3]):-
    слить( Спнс1, Остальные2, Остальные3 ).
```

8.3.2. Табличная организация длинных процедур

Длинные процедуры допустимы, если они имеют регулярную структуру. Обычно эта структура представляет собой множество фактов, соответствующее определению какого-либо отношения в табличной форме. Преимущества такой организации длинной процедуры состоят в том, что:

- Ее структуру легко понять.
- Ее удобно совершенствовать: улучшать ее можно, просто добавляя новые факты.
- Ее легко проверять и модифицировать (просто заменяя отдельные факты, независимо от остальных).

8.3.3. Комментирование

Программные комментарии должны объяснять в первую очередь, для чего программа предназначена и как ею пользоваться, и только затем – подробности используемого метода решения и другие программные детали. Главная цель комментариев – обеспечить пользователю возможность применять программу, понимать ее и, может быть, модифицировать. Комментарии должны содержать в наиболее краткой форме всю необходимую для этого информацию. Недостаточное комментирование – распространенная ошибка, однако программу можно и перенасытить комментариями. Объяснения деталей, которые и так ясны из самого текста программы, являются ненужной перегрузкой.

Длинные фрагменты комментариев следует располагать перед текстом, к которому они относятся, в то время как короткие комментарии должны быть вкраплены в сам текст. Информация, которую в самом общем случае следует включать в комментарии, должна охватывать следующие вопросы:

- Что программа делает, как ею пользоваться (например, какую цель следует активизировать и каков вид ожидаемых результатов), примеры ее применения.
- Какие предикаты относятся к верхнему уровню?
- Как представлены основные понятия (объекты)?
- Время выполнения и требования по объему памяти.
- Каковы ограничения на программу?
- Использует ли она какие-либо средства, связанные с конкретной операционной системой?
- Каков смысл предикатов программы? Каковы их аргументы? Какие аргументы являются «входными» и какие – «выходными», если это известно? (В момент запуска предиката входные аргументы имеют полностью определенные значения, не содержащие неконкретизированных переменных.)
- Алгоритмические и реализационные детали.

8.4. Отладка

Когда программа не делает того, чего от нее ждут, главной проблемой становится отыскание ошибки (или ошибок). Всегда легче найти ошибку в какой-нибудь части программы (или в отдельном модуле), чем во всей программе. Поэтому следует придерживаться следующего хорошего принципа: проверять сначала более мелкие программные единицы и только после того, как вы убедились, что им можно доверять, начинать проверку большего модуля или всей программы.

Отладка в Прологе облегчается двумя обстоятельствами: во-первых, Пролог – интерактивный язык, поэтому можно непосредственно обратиться к любой части программы, задав пролог-системе соответствующий вопрос; во-вторых, в реализациях Пролога обычно имеются специальные средства отлад-

ки. Следствием этих двух обстоятельств является то, что отладка программ на Прологе может производиться, вообще говоря, значительно эффективнее, чем в других языках программирования.

Основным средством отладки является *трассировка* (tracing). «Трассировать цель» означает: предоставить пользователю информацию, относящуюся к достижению этой цели в процессе ее обработки пролог-системой. Эта информация включает:

- Входную информацию – имя предиката и значения аргументов в момент активизации цели.
- Выходную информацию – в случае успеха, значения аргументов, удовлетворяющих цели; в противном случае – сообщение о неуспехе.
- Информацию о повторном входе, т. е. об активизации той же цели в результате автоматического возврата.

В промежутке между входом и выходом можно получить трассировочную информацию для всех подцелей этой цели. Таким образом, мы можем следить за обработкой нашего вопроса на всем протяжении нисходящего пути от исходной цели к целям самого нижнего уровня, вплоть до отдельных фактов. Такая детальная трассировка может оказаться непрактичной из-за непомерно большого количества трассировочной информации. Поэтому пользователь может применить «селективную» трассировку. Существуют два механизма селекции: первый подавляет выдачу информации о целях, расположенных ниже некоторого уровня; второй трассирует не все предикаты, а только некоторые, указанные пользователем.

Средства отладки приводятся в действие при помощи системно-зависимых встроенных предикатов. Обычно используется следующий стандартный набор таких предикатов:

trace

запускает полную трассировку всех целей, следующих за **trace**.

notrace

прекращает дальнейшее трассирование.

spy(P) (следи за P)

устанавлнвает режим трассировки предиката P. Обращение к **spy** примеияют, когда хотят получить информацию только об указанном предикате и избежать трассировочной иифорлации от других целей (как выше, так и ниже уровня запуска P). «Следить» можно сразу за несколькими предикатами.

nospy(P)

прекращает «слежку» за P.

Трассировка ииже определеиной глубины может быть подавлена во время выполнеиия программы при помощи специальных комаид. Существуют и другие команды отладки, такие как возврат к предыдущей точке процесса вычислеиий. После такого возврата можно, например, повторить вычисления с большей степеиью детализации трассировки.

8.5. Эффективность

Существует несколько аспектов эффективности программ, включая такие иаиболее общие, как время выполиения и требования по объему памяти. Другим аспектом является время, необходимое программисту для разработки программы.

Традициоиная архитектура вычислительных машии не очень хорошо приспособлеиа для реализации прологовского способа выполнения программ, предусматривающего достижение целей из некоторого списка. Поэтому ограиичеииость ресурсов по времени и пространству сказывается в Прологе, пожалуй, в большей степени, чем в большиистве других языков программирования. Вызовет ли это трудности в практических приложениях, зависит от задачи. Фактор времеии практически не имеет значения, если пролог-программа, которую запускают по иескольку раз в деиь, занимает 1 секуиду процессориого времени, а соответствующая программа на каком-либо другом языке, скажем на Фортране, – 0.1 секунды. Разиица в эффективности становится существенной, если эти две программы требуют 50 и 5 минут соответственно.

С другой стороны, во многих областях применеиия Пролога он может существенно сократить время разработки программ. Программы на Прологе, вообще говоря, легче писать, легче понимать и отлаживать, чем программы, написанные на традиционных языках. Задачи, тяготеющие к «царству Пролога», включают в себя обработку символьной, нечисловой информации, структурированных объектов данных и отношеиий между ними. Пролог успешно прнменяется, в частности, в таких областях, как символьное решение уравнеиий, планирование, базы данных, автоматическое решение задач, машинное макетирование, реализация языков программирования, дискретное и аналоговое моделирование, архитектурное проектирование, машинное обучение, понимание естественного языка, экспертные системы и другие задачи искусственного интеллекта. С другой стороны, применение Пролога в области вычислительной математики вряд ли можно считать естественным.

Прогон *откомпилированной* программы обычно имеет большую эффективность, чем *интерпретация*. Поэтому, если пролог-система содержит как интерпретатор, так и компилятор, следует пользоваться компилятором, если время выполнеиия критичио.

Если программа страдает иеэффективностью, то ее обычно можно кардинально улучшить, изменив сам алгоритм. Однако для того, чтобы это сделать, необходимо изучить процедурные аспекты программы. Простой способ сокращения времени выполнения состоит в нахождении более удачного порядка предложений в процедуре и целей – в телах процедур. Другой, относительно простой метод заключается в управлении действиямн системы посредством отсеченнй.

Полезные идеи, относящиеся к повышению эффективности, обычно возникают только при достижеиии более глубокого понимания задачи. Более эффективный алгоритм может, вообще говоря, привести к улучшениям двух видов:

- Повышение эффективности поиска путем скорейшего отказа от ненужного перебора и от вычисления бесполезиых вариантов.
- Применение структур данных, более приспособленных для представления объектов программы, с целью реализовать операции иад ними более эффективно.

Мы изучим оба вида улучшений на примерах. Кроме того, мы рассмотрим на примере еще один метод повышения эффективности. Этот метод основан на добавлении в базу данных тех промежуточных результатов, которые с большой вероятностью могут потребоваться для дальнейших вычислений. Вместо того, чтобы вычислять их снова, программа просто отыщет их в базе данных как уже известные факты.

8.5.1. Повышение эффективности решения задачи о восьми ферзях

В качестве простого примера повышения эффективности давайте вернемся к задаче о восьми ферзях (см. рис. 4.7). В этой программе Y-координаты ферзей перебираются последовательно – для каждого ферзя пробуются числа от 1 до 8. Этот процесс был запрограммирован в виде цели

```
принадлежит( Y, [1,2,3,4,5,6,7,8] )
```

Процедура принадлежит работает так: вначале пробует Y = 1, затем Y = 2, Y = 3 и т.д. Поскольку ферзи расположены один за другим в смежных вертикалях доски, очевидно, что такой порядок перебора не является оптимальным. Дело в том, что ферзи, расположенные в смежных вертикалях будут бить друг друга, если они не будут разнесены по вертикали на расстояние, превышающее, по крайней мере одно поле. В соответствии с этим наблюдением можно попытаться повысить эффективность, просто изменив порядок рассмотрения координат-кандидатов. Например:

```
принадлежит( Y, [1,5,2,6,3,7,4,8] )
```

Это маленькое изменение уменьшит время, необходимое для нахождения первого решения, в 3–4 раза.

В следующем примере такая же простая идея, связанная с измененнем порядка, превращает практически неприемлемую временную сложность в тривиальную.

8.5.2. Повышение эффективности программы раскраски карты

Задача раскраски карты состоит в приписывании каждой стране на заданной карте одного из четырех заданных цветов с таким расчетом, чтобы ни одна пара соседних стран не была окрашена в одинаковый цвет. Существует теорема, которая гарантирует, что это всегда возможно.

Пусть карта задана отношением соседства

соседи(Страна, Соседи)

где **Соседи** – список стран, граничащих со страной **Страна**. При помощи этого отношения карта Европы с 20-ю странами будет представлена (в алфавитном порядке) так:

```
соседи(австрия,[венгрия,западная_германия,италия,
                лихтенштейн, чехословакия,
                швейцария,югославия]).
соседи(албания,[греция,югославия]).
соседи(андорра,[испания,франция]).
...
```

Решение представим в виде списка пар вида

Страна / Цвет

которые устанавливают цвет для каждой страны на данной карте. Для каждой карты названия стран всегда известны заранее, так что задача состоит в нахождении цветов. Таким образом, для Европы задача сводится к отысканию подходящей конкретизации переменных C1, C2, C3 и т.д. в списке

[австрия/C1, албания/C2, андорра/C3, ...]

Теперь определим предикат

цвета(СписЦветСтран)

который истинен, если **СписЦветСтран** удовлетворяет тем ограничениям, которые наложены на раскраску отношением **соседи**. Пусть четырьмя цветами будут желтый, синий, красный и зеленый. Условие запрета

раскраски соседних стран в одинаковый цвет можно сформулировать на Прологе так:

```
цвета( []).

цвета( [Страна/Цвет | Остальные] ) :-
цвета( Остальные),
   принадлежит(Цвет,[желтый,синий,красный,зеленый]),
   not( принадлежит( Страна1/Цвет, Остальные),
                              сосед( Страна,Страна1) ).

сосед( Страна, Страна1) :-
    соседи( Страна, Соседи),
    принадлежит( Страна1, Соседи).
```

Здесь **принадлежит(X, L)** – как всегда, отношение принадлежности к списку. Для простых карт с небольшим числом стран такая программа будет работать. Что же касается Европы, то здесь результат проблематичен. Если считать, что мы располагаем встроенным предикатом **setof**, то можно попытаться раскрасить карту Европы следующим образом. Определим сначала вспомогательное отношение:

```
страна( С) :- соседи( С, _).
```

Тогда вопрос для раскраски карты Европы можно сформулировать так:

```
?- setof(Стр/Цвет,страна(Стр),СписЦветСтран),
   цвета(СписЦветСтран).
```

Цель **setof** – построить «шаблон» списка **СписЦветСтран**, в котором в элементах вида **страна/цвет** вместо цветов будут стоять неконкретизированные переменные. Предполагается, что после этого цель **цвета** конкретизирует их. Такая попытка скорее всего потерпит неудачу вследствие неэффективности работы программы.

Тщательное исследование способа, при помощи которого пролог-система пытается достичь цели **цвета**, обнаруживает источник неэффективности. Страны расположены в списке в алфавитном порядке, а он не имеет никакого отношения к их географическим связям. Порядок, в котором странам приписываются цвета, соответствует порядку их расположения в списке (с конца к началу), что в нашем случае никак не связано с отношением **соседи**. Поэтому процесс раскраски начинается в одном конце карты,

продолжается в другом и т.д., перемещаясь по ней более или менее случайно. Это легко может привести к ситуации, когда при попытке раскрасить очередную страну окажется, что она окружена странами, уже раскрашенными во все четыре доступных цвета. Подобные ситуации приводят к возвратам, снижающим эффективность.

Ясно поэтому, что эффективность зависит от порядка раскраски стран. Интуиция подсказывает простую стратегию раскраски, которая должна быть лучше, чем случайная: начать со страны, имеющей много соседей, затем перейти к ее соседям, затем – к соседям соседей и т.д. В случае Европы хорошим кандидатом для начальной страны является Западная Германия (как имеющая наибольшее количество соседей – 9). Понятно, что при построении шаблона списка элементов вида страна/цвет Западную Германию следует поместить в конец этого списка, а остальные страны – добавлять со стороны его начала. Таким образом, алгоритм раскраски, который начинает работу с конца списка, в начале займется Западной Германией и продолжит работу, переходя от соседа к соседу.

Новый способ упорядочивания списка стран резко повышает эффективность по отношению к исходному, алфавитному порядку, и теперь возможные раскраски карты Европы будут получены без труда.

Можно было бы построить такой правильно упорядоченный список стран вручную, но в этом нет необходимости. Эту работу выполнит процедура **создспис**. Она начинает построение с некоторой указанной страны (в нашем случае – с Западной Германии) и собирает затем остальные страны в список под названием **Закрытый**. Каждая страна сначала попадает в другой список, названный **Открытый**, а потом переносится в **Закрытый**. Всякий раз, когда страна переносится из **Открытый** в **Закрытый**, ее соседи добавляются в **Открытый**.

```
создспис( Спис) :-
    собрать( [запгермания], [], Спис ).
    собрать( [], Закрытый, Закрытый).
                % Кандидатов в Закрытый больше нет

собрать( [X | Открытый], Закрытый, Спис) :-
    принадлежит( X | Закрытый), !,
                % X уже собран?
```

```
    собрать( Открытый, Закрытый, Спис).
                % Отказаться от X
собрать( [X | Открытый], Закрытый, Спис) :-
    соседи( X, Соседи),
                % Найти соседей X
    конк( Соседи, Открытый, Открытый1),
                % Поместить их в Открытый
    собрать( Открытый1, [X | Закрытый], Спис).
                % Собрать остальные
```

Отношение конк – как всегда – отношение конкатенации списков.

8.5.3. Повышение эффективности конкатенации списков за счет совершенствования структуры данных

До сих пор в наших программах конкатенация была определена так:

```
конк( [], L, L).
конк( [X | L1], L2, [X | L3] ) :-
    конк( L1, L2, L3 ).
```

Эта процедура неэффективна, если первый список – длинный. Следующий пример объясняет, почему это так:

```
?- конк( [a,b,c], [d,e], L).
```

Этот вопрос порождает следующую последовательность целей:

```
конк( [a,b,c], [d,e], L)
 конк( [b,c], [d,e], L')          где L = [a|L']
  конк( [c], [d,e], L'')           где L' = [b|L'']
   конк( [], [d,e], L''')           где L'' = [c|L''']
    true          (истина)          где L''' = [d,e]
```

Ясно, что программа фактически сканирует весь первый список, пока не обнаружит его конец.

А нельзя ли было бы проскочить весь первый список за один шаг и сразу подсоединить к нему второй список, вместо того, чтобы постепенно продвигаться вдоль него? Но для этого необходнмо знать, где расположен конец списка, а следовательнио, мы нуждаемся в другом его представлении. Одии из вариантов - представлять список парой списков. Например, список

```
[a,b,c]
```

можно представить следующими двумя списками:

```
L1 = [a,b,c,d,e]
L2 = [d,e]
```

Подобная пара списков, записанная для краткости как L1-L2, представляет собой «разность» между L1 и L2. Это представление работает только при том условии, что L2 - «конечный участок» списка L1. Заметим, что один и тот же список может быть представлен несколькими «разностными парами». Поэтому список [a,b,c] можно представить как

```
[a,b,c]-[]
```

или

```
[a,b,c,d,e]-[d,e]
```

или

```
[a,b,c,d,e | T]-[d,e | T]
```

или

```
[a,b,c | T]-T
```

где T - произвольный список, и т.п. Пустой список представляется любой парой L-L.

Поскольку второй член пары указывает на конец списка, этот конец доступен сразу. Это можно использовать для эффективной реализации конкатенации. Метод показан на рис. 8.1. Соответствующее отношение конкатенацни записывается на Прологе в виде факта

```
конкат( A1-Z1, Z1-Z2, A1-Z2).
```

Давайте используем **конкат** для конкатенации двух списков: списка [a,b,c], представленного парой [a,b,c|T1]-T1, и списка [d,e], представленного парой [d,e | T2]-T2 :

```
?- конкат( [a,b,c | T1]-T1, [d,e | T2]-T2, L ).
```

Оказывается, что для выполнения конкатенации достаточно простого сопоставления этой цели с предложением **конкат**. Результат сопоставления:

```
T1 = [d,e | T2]
L = [a,b,c,d,e | T2]-T2
```

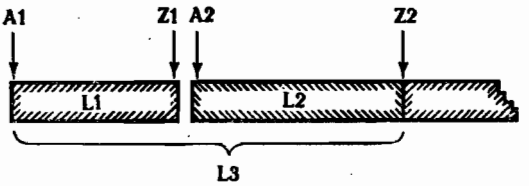


Рис. 8.1. Конкатенация списков, представленных в виде разностных пар. **L1** представляется как **A1–Z1**, **L2** как **A2–Z2** и результат **L3** – как **A1–Z2**. При этом должно выполняться равенство **Z1 = A2**.

8.5.4. Повышение эффективности за счет добавления вычисленных фактов к базе данных

Иногда в процессе вычислений приходится одну и ту же цель достигать снова и снова. Поскольку в Прологе отсутствует специальный механизм выявления этой ситуации, соответствующая цепочка вычислений каждый раз повторяется заново.

В качестве примера рассмотрим программу вычисления N-го числа Фибоначчи для некоторого заданного N. Последовательность Фибоначчи имеет вид:

1, 1, 2, 3, 5, 8, 13, ...

Каждый член последовательности, за исключением первых двух, представляет собой сумму предыдущих двух членов. Для вычисления N-го числа Фибоначчи F определим предикат

фиб(N, F)

Нумерацию чисел последовательности начнем с N = 1. Программа для **фиб** обрабатывает сначала первые два числа Фибоначчи как два особых случая, а затем определяет общее правило построения последовательности Фибоначчи:

```
фиб( 1, 1).                    % 1-е число Фибоначчи

фиб( 2, 1).                    % 2-е число Фибоначчи

фиб( N, F) :-                  % N-е число Фиб., N > 2
    N > 2,
    N1 is N - 1, фиб( N1, F1),
    N2 is N - 2, фиб( N2, F2),
    F is F1 + F2.              % N-е число есть сумма двух
                               % предыдущих
```

Процедура **фиб** имеет тенденцию к повторению вычислений. Это легко увидеть, если трассировать цель

```
?- фиб( 6, F).
```

На рис. 8.2 показано, как протекает этот вычислительный процесс. Например, третье число Фибоначчи f(3) понадобилось в трех местах, и были повторены три раза одни и те же вычисления.

Этого легко избежать, если запоминать каждое вновь вычисленное число. Идея состоит в применении встроенной процедуры **assert** для добавления этих (промежуточных) результатов в базу данных в виде фактов. Эти факты должны предшествовать другим предложениям, чтобы предотвратить применение общего правила в случаях, для которых результат уже известен. Усовершенствованная процедура **фиб2** отличается от **фиб** только этим добавлением:

```
фиб2( 1, 1).                   % 1-е число Фибоначчи

фиб2( 2, 1).                   % 2-е число Фибоначчи

фиб2( N, F) :-                 % N-е число Фиб., N > 2
    N > 2,
    N1 is N - 1, фиб2( N1, F1),
    N2 is N - 2, фиб2( N2, F2),
    F is F1 + F2,              % N-е число есть сумма
                               % двух предыдущих
    asserta( фиб2(N,F) ).      % Запоминание N-го числа
```

Эта программа, при попытке достичь какую-либо цель, будет смотреть сперва на накопленные об этом

отношении факты и только после этого применять общее правило. В результате, после вычисления цели **фиб2(N, F)**, все числа Фибоначчи вплоть до N-го будут сохранены. На рис. 8.3 показан процесс вычисления 6-го числа при помощи **фиб2**. Сравнение этого рисунка с рис. 8.2 показывает, на сколько уменьшилась вычислительная сложность. Для больших N такое уменьшение еще более ощутимо.

Запоминание промежуточных результатов - стандартный метод, позволяющий избегать повторных вычислений. Следует, однако, заметить, что в случае чисел Фибоначчи повторных вычислений можно избежать еще и применением другого алгоритма, а не только запоминанием промежуточных результатов.

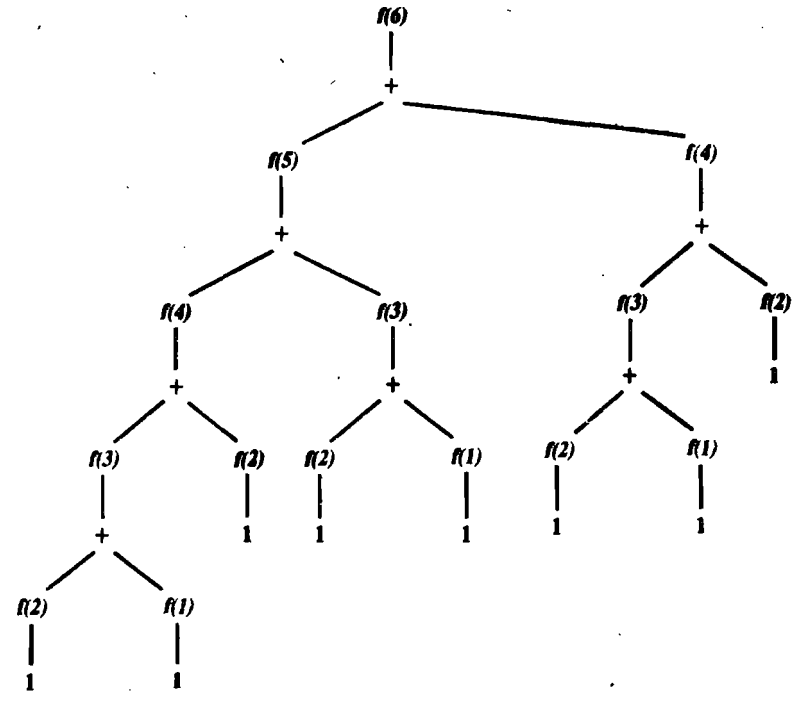


Рис. 8.2. Вычисление 6-го числа Фибоначчи процедурой фиб.

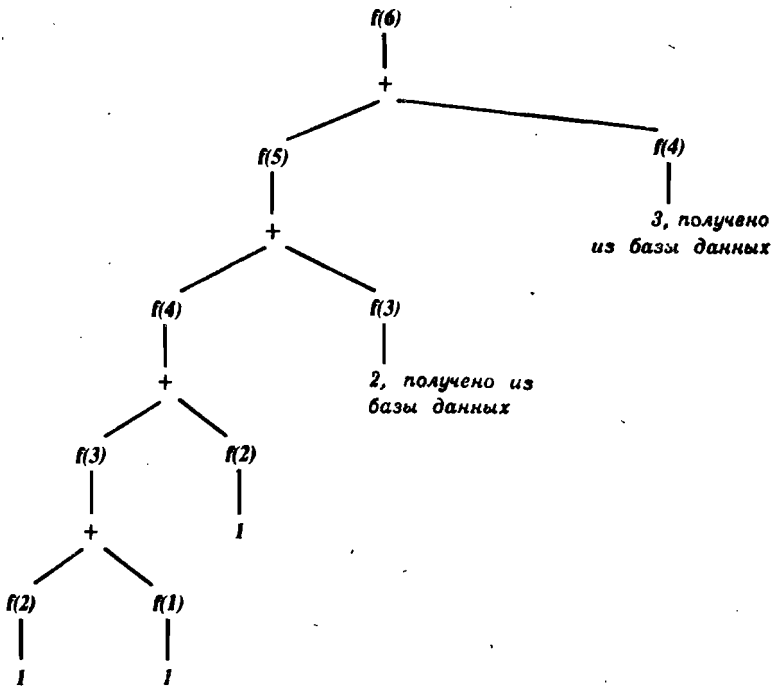


Рис. 8.3. Вычисление 6-го числа Фибоначчи при помощи процедуры **фиб2**, которая запоминает предыдущие результаты. По сравнению с процедурой **фиб** здесь вычислений меньше (см. рис. 8.2).

Этот новый алгоритм позволяет создать программу более трудную для понимания, зато более эффективную. Идея состоит на этот раз не в том, чтобы определить N-е число Фибоначчи просто как сумму своих предшественников по последовательности, оставляя рекурсивным вызовам организовать вычисления «сверху вниз» вплоть до самых первых двух чисел. Вместо этого можно работать «снизу вверх»: начать с первых двух чисел и продвигаться вперед, вычисляя члены последовательности один за другим. Остановиться нужно в тот момент, когда будет достигнуто N-е число. Большая часть работы в такой программе выполняется процедурой

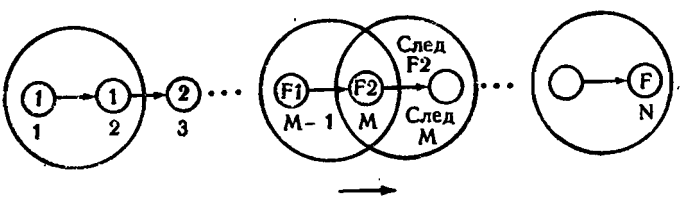
фибвперед(M, N, F1, F2, F)

Здесь F1 и F2 – (M – 1)-е и M-е числа, а F – N-е число Фибоначчи. Рис. 8.4 помогает понять отношение **фибвперед**. В соответствии с этим рисунком **фибвперед** находит последовательность преобразований для достижения конечной конфигурации (в которой M = N) из некоторой заданной начальной конфигурации. При запуске **фибвперед** все его аргументы, кроме F, должны быть конкретизированы, а M должно быть меньше или равно N. Вот эта программа:

```
фиб3( N, F) :-
    фибвперед( 2, N, 1, 1, F).
                % Первые два числа Фиб. равны 1

фибвперед( M, N, F1, F2, F2) :-
    M >= N.  % N-е число достигнуто

фибвперед( M, N, F1, F2, F) :-
    M < N,   % N-е число еще не достигнуто
    СледM is M + 1,
    СледF2 is F1 + F2,
    фибвперед( СледM, N, F2, СледF2, F).
```



Начальная конфигурация, здесь M = 2

Переход от конфигурации M к M + 1

Конечная конфигурация, здесь M = N

Рис. 8.4. Отношения в последовательности Фибоначчи. "Конфигурация" изображается здесь в виде большого круга и определяется тремя параметрами: индексом M и двумя последовательными числами f(M–1) и f(M).

Упражнения

8.1. Все показанные ниже процедуры **подсп1, подсп2 и подсп3 реализуют отношение взятия подсписка.**

Отношение **подсп1** имеет в значительной мере процедурное определение, тогда как **подсп2** и **подсп3** написаны в декларативном стиле. Изучите поведение этих процедур на примерах нескольких списков, обращая внимание на эффективность работы. Две из них ведут себя одинаково и имеют одинаковую эффективность. Какие? Почему оставшаяся процедура менее эффективна?

```
подсп1( Спис, Подспис) :-
    начало( Спис, Подспис).

подсп1( [_ | Хвост], Подспис) :-
                        % Подспис - подсписок хвоста
    подсп1( Хвост, Подспис).

начало( _, []).

начало( [X | Спис1], [X | Спис2] ) :-
    начало( Спис1, Спис2).

подсп2( Спис, Подспис) :-
    конк( Спис1, Спис2, Спис),
    конк( Спис3, Подспис, Спис1).

подсп3( Спис, Подспис) :-
    конк( Спис1, Спис2, Спис),
    конк( Подспис, _, Спис2).
```

8.2. Определите отношение

добавить_в_конец(Список, Элемент, НовыйСписок)

добавляющее **Элемент** в конец списка **Список**; результат – **НовыйСписок**. Оба списка представляйте разностными парами.

8.3. Определите отношение

обратить(Список, ОбращенныйСписок)

где оба списка представлены разностными парами.

8.4. Перепишите процедуру **собрать** из разд. 8.5.2, используя разностное представление списков, чтобы конкатенация выполнялась эффективнее.

Резюме

- Для оценки качества программы существует несколько критериев:

правильность
эффективность
простота, читабельность
удобство модификации
документированность

- Принцип *пошаговой детализации* – хороший способ организации процесса разработки программ. Пошаговая детализация применима к отношениям, алгоритмам и структурам данных.
- Следующие методы часто помогают находить идеи для совершенствования программ на Прологе:

Применение рекурсии: выявить граничные и общие случаи рекурсивного определения.

Обобщение: рассмотреть такую более общую задачу, которую проще решить, чем исходную.

Использование рисунков: графическое представление помогает в выявлении важных отношений.

- Полезно следовать некоторым стилистическим соглашениям для уменьшения опасности внесения ошибок в программы и создания программ, легких для чтения, отладки и модификации.
- В пролог-системах обычно имеются средства отладки. Наиболее полезными являются средства трассировки программ.
- Существует много способов повышения эффективности программы. Наиболее простые способы включают в себя:

изменение порядка целей и предложений

управляемый перебор при помощи введения отсечений

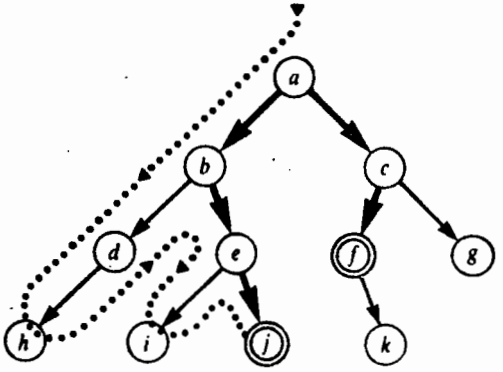
запоминание (с помощью **assert**) решений, которые иначе пришлось бы перевычислять

Более тонкие и радикальные методы связаны с улучшением алгоритмов (особенно, в части повышения эффективности перебора) и с совершенствованием структур данных.

Часть 2

ПРОЛОГ

В ИСКУССТВЕННОМ ИНТЕЛЛЕКТЕ



9 ОПЕРАЦИИ НАД СТРУКТУРАМИ ДАННЫХ

Один из фундаментальных вопросов программирования – это вопрос о представлении сложных объектов (таких как, например, множества), а также вопрос об эффективной реализации операций над подобными объектами. В этой главе мы рассмотрим несколько часто используемых структур данных, принадлежащих к трем большим семействам: спискам, деревьям и графам. Мы изучим способы представления этих структур на Прологе и составим программы, реализующие некоторые операции над ними, в том числе, сортировку списков, работу с множествами как древовидными структурами, запись элементов данных в дерево, поиск данных в дереве, нахождение пути в графе и т.п. Мы подробно разберем несколько примеров, чрезвычайно поучительных с точки зрения программирования на Прологе.

9.1. Представление списков. Сортировка

9.1.1. Замечания о некоторых альтернативных способах представления списков

В главе 3 была введена специальная система обозначений для списков (специальная прологовская нотация), которую мы и использовали в последующем изложении. Разумеется, это был всего лишь один из способов представления списков на Прологе. *Список*

– это, в самом общем смысле, структура, которая либо

- *пуста*, либо
- состоит из *головы* и *хвоста*, причем хвост должен быть сам списком.

Поэтому для представления этой структуры нам необходимо иметь всего лишь два языковых средства: спецнальный символ, обозначающий пустой список, и функтор для соединения головы с хвостом. Мы могли бы, например, выбрать

ничего_не_делать

в качестве символа, обозначающего пустой список, и атом

затем

в качестве инфиксного оператора для построения списка по заданным голове и хвосту. Этот оператор мы можем объявить в программе, например, так:

```
:- op( 500, xfy, затем).
```

Список

[войти, сесть, поужинать]

можно было бы тогда записать как

```
войти затем сесть затем поужинать
затем ничего_не_делать
```

Важно заметить, что на соответствующем уровне абстракции специальная прологовская нотация и всевозможные альтернативные способы обозначения списков сводятся, фактически, к одному и тому же представленню. В связи с этим типовые операции над списками, такие как

```
принадлежит( X, L)
конк( L1, L2, L3)
удалить( X, L1, L2)
```

запрограммированные нами в специальной прологовской нотации, легко поддаются перепрограммированию в различные системы обозначений, выбранные пользователем. Например, отношение **конк** транслируется на язык «затем – ничего_не_делать» следующим образом. Определение, которое мы использовали до сих пор,

имеет вид

```
конк( [], L, L).
конк( [X|L1], L2, [X|L3] ) :-
    конк( L1, L2, L3).
```

В новой системе обозначений оно превращается в

```
конк( ничего_не_делать, L, L).
конк( X затем L1, L2, X затем L3) :-
    конк( L1, L2, L3).
```

Этот пример показывает, как легко наши определения отношений над списками обобщаются на весь класс структур этого типа. Решение о том, какой именно способ записи списков будет использоваться в той или иной программе, следует принимать в соответствии с тем смыслом, который мы придаем списку в каждом конкретном случае. Если, например, список – это просто множество элементов, то наиболее удобна обычная прологовская нотация, поскольку в ней непосредственно выражается то, что программист имел в виду. С другой стороны, некоторые типы выражений также можно трактовать как своего рода списки. Например, для конъюнктов в исчислении высказываний подошло бы следующее спископодобное представление:

- **истина** соответствует пустому списку,
- **&** – оператор для соединения головы с хвостом, определяемый, например, как

```
:- op( 300, xfy, &)
```

Конъюнкция членов a, b и c выглядела бы тогда как

```
a & b & c & истина
```

Все приведенные примеры базируются, по существу, на одной и той же структуре, представляющей список. Однако в гл. 8 мы рассмотрели существенно другой способ, влияющий на эффективность вычислений. Уловка состояла в том, что список представлялся в виде пары списков, являясь их «разностью». Было показано, что такое представление приводит к очень эффективной реализации отношения конкатенации.

Материал настоящего раздела пролнвает свет и на то различие, которое существует между применением операторов в математике и применением их в Прологе. В математике с каждым оператором всегда связано некоторое действие, в то время как в Прологе операторы используются просто для представления структур.

Упражнения

9.1. Определите отношение

список(Объект)

для распознавания случаев, когда **Объект** является стандартным прологовским списком.

9.2. Определите отношение принадлежности к списку, используя систему обозначений, введенную в этом разделе: «затем – ничего_не_делать».

9.3. Определите отношение

преобр(СтандСпис, Спис)

для преобразования списков из стандартного представления в систему «затем-ничего_не_делать». Например:
преобр([a,b], a затем b затем ничего_не_делать)

9.4. Обобщите отношение **преобр** на случай произвольного альтернативного представления списков. Конкретное представление задается символом, обозначающим пустой список, и функтором для соединения головы с хвостом. В отношении **преобр** придется добавить два новых аргумента:

преобр(СтандСпис, Спис, Функтор, ПустСпис)

Примеры применения этого отношения:

?- преобр([a,b], L, затем, ничего_не_делать).

L = a затем b затем ничего_не_делать

?- преобр([a,b,c], L, +, 0).

L = a+(b+(c+0))

9.1.2. Сортировка списков

Сортировка применяется очень часто. Список можно отсортировать (упорядочить), если между его элементами определено отношение порядка. Для удобства изложения мы будем использовать отношение порядка

больше(X, Y)

означающее, что X *больше, чем* Y, независимо от того, что мы в действительности понимаем под «больше, чем». Если элементами списка являются числа, то отношение **больше** будет, вероятно, определено как

больше(X, Y) := X > Y.

Если же элементы списка – атомы, то отиошеиие **больше** может соответствовать алфавитному порядку между ними.

Пусть

сорт(Спис, УпорСпис)

обозначает отношение, в котором **Спис** – некоторый список, а **УпорСпис** – это список, составленный из тех же элементов, но упорядоченный по возрастанию в соответствии с отношением **больше**. Мы построим три определения этого отиошеиия на Прологе, осиованиые иа трех различиых идеях о механизме сортировки. Вот первая идея:

Для того, чтобы упорядочить список **Спис**, необходимо:

- Найти в **Спис** два смежных элемента X и Y, таких, что **больше(X, Y)**, и поменять X и Y местами, получив тем самым иовый список **Спис1**; затем отсортировать **Спис1**.
- Если в **Спис** нет ии одной пары смежных элементов X и Y, таких, что **больше(X, Y)**, то считать, что **Спис** уже отсортирован.

Мы переставили местами два элемента X и Y, расположенные в списке «не в том порядке», с целью приблизить список к своему упорядоченному состоянию. Имеется в виду, что после достаточно большого числа перестановок все элементы списка будут расположены в правильном порядке. Описанный принцип сортировки принято называть *методом пузырька*, поэтому соответствующая прологовская процедура будет называться **пузырек**.

```
пузырек( Спис, УпорСпис) :-
    перест( Спис, Спис1), !,        % Полезная перестановка?
    пузырек( Спис1, УпорСпис).

пузырек( УпорСпис, УпорСпис).
                    % Если нет, то список уже упорядочен

перест([X,Y | Остаток], [Y,X | Остаток] ):-
                    % Перестановка первых двух элементов
    больше( X, Y).

перест([Z | Остаток], [Z | Остаток1] ):-
    перест(Остаток, Остаток1).      % Перестановка в хвосте
```

Еще один простой алгоритм сортировки называется *сортировкой со вставками*. Он основан на следующей идее:

Для того, чтобы упорядочить непустой список **L = [X | Хв]**, необходимо:

(1) Упорядочить хвост **Хв** списка **L**.

(2) Вставить голову **X** списка **L** в упорядоченный хвост, поместив ее в такое место, чтобы получившийся список остался упорядоченным. Список отсортирован.

Этот алгоритм транслируется в следующую процедуру **вставсорт** на Прологе:

```
вставсорт([], []).

вставсорт( [X | Хв], УпорСпис) :-
    вставсорт( Хв, УпорХв),         % Сортировка хвоста
    встав( X, УиорХв, УпорСпис).
                    % Вставить X на нужное место
```

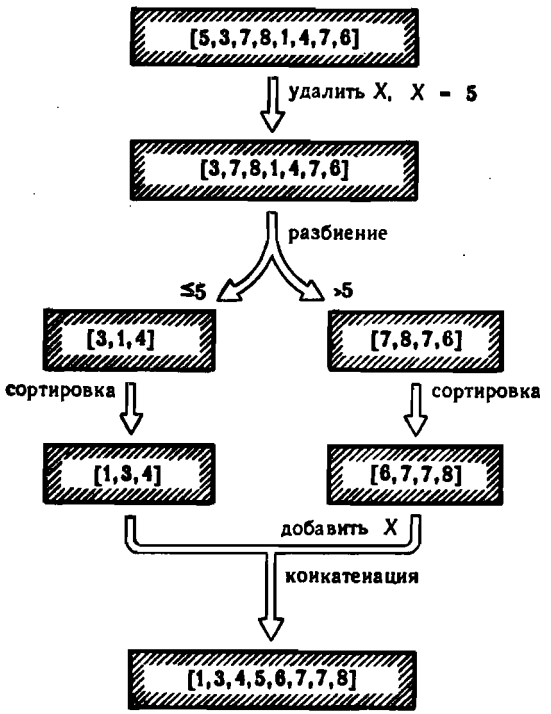


Рис. 9.1. Сортировка списка процедурой **быстрсорт**.

```
встав( X, [Y | УпорСпис], [Y | УпорСпис1]) :-
    больше( X, Y), !,
    встав( X, УпорСпис, УпорСпис1).

встав( X, УпорСпис, [X | УпорСпис] ).
```

Процедуры сортировки пузырек и **вставсорт** просты, но не эффективны. Из этих двух процедур процедура со вставками более эффективна, однако среднее время, необходимое для сортировки списка длиной n процедурой **вставсорт**, возрастает с ростом n пропорционально n^2. Поэтому для длинных списков значительно лучше работает алгоритм *быстрой сортировки*, основанный на следующей идее (рис. 9.1):

Для того, чтобы упорядочить непустой список L, необходимо:

(1) Удалить из списка L какой-нибудь элемент X и разбить оставшуюся часть на два списка, называемые **Меньш** и **Больш**, следующим образом: все элементы большие, чем X, принадлежат списку **Больш**, остальные – списку **Меньш**.

(2) Отсортировать список **Меньш**, результат – список **УпорМеньш**.

(3) Отсортировать список **Больш**, результат – список **УпорБольш**.

(4) Получить результирующий упорядоченный список как конкатенацию списков **УпорМеньш** и **[X | УпорБольш]**.

Заметим, что если исходный список пуст, то результатом сортировки также будет пустой список. Реализация быстрой сортировки на Прологе показана на рис. 9.2. Здесь в качестве элемента X, удаляемого из списка, всегда выбирается просто голова этого списка. Разбиение на два списка запрограммировано как отношение с четырьмя аргументами:

разбиение(X, L, Больш, Меньш).

Временная сложность нашего алгоритма зависит от того, насколько нам повезет при разбиении сортируемого списка. Если списки всегда разбиваются на два списка примерно равной длины, то процедура сортировки имеет временную сложность порядка $n \log n$, где n – длина исходного списка. Если же, наоборот, разбиение всегда приводит к тому, что один из списков оказывается значительно больше другого, то сложность будет порядка n^2. Анализ показывает, что, к счастью, средняя производительность быстрой сортировки ближе к лучшему случаю, чем к худшему.

Программу, показанную на рис. 9.2, можно усовершенствовать, если реализовать операцию конкатенации более эффективно. Напомним, что конкатенация

```
быстрсорт( [], [] ).

быстрсорт( [X | Хвост], УпорСпис) :-
    разбиение( X, Хвост, Меньш, Больш),
    быстрсорт( Меньш, УпорМеньш),
    быстрсорт( Больш, УпорБольш),
    конк( УпорМеньш, [X | УпорБольш], УпорСпис).

разбиение( X, [], [], [] ).

разбиение( X, [Y | Хвост], [Y | Меньш], Больш ) :-
    больше( X, Y), !,
    разбиение( X, Хвост, Меньш, Больш).

разбиение( X, [Y | Хвост], Меньш, [Y | Больш] ) :-
    разбиение( X, Хвост, Меньш, Больш).

конк( [], L, L).

конк( [X | L1], L2, [X | L3] ) :-
    конк( L1, L2, L3 ).
```

Рис. 9.2. Быстрая сортировка.

становится тривиальной операцией после применения разностного представления списков, введенного в гл. 8. Для того, чтобы использовать эту идею в нашей процедуре сортировки, нужно представить встречающиеся в ней списки в форме пар вида A–Z следующим образом:

УпорМеньш имеет вид A1–Z1
УпорБольш имеет вид A2–Z2

Тогда конкатенации списков

УпорМеньш и [X | УпорБольш]

будет соответствовать конкатенация пар

A1–Z1 и [X | A2]–Z2

В результате мы получим

A1–Z2, причем Z1 = [X | A2]

Пустой список представляется парой Z–Z. Систематически вводя изменения в программу рис. 9.2, мы получим более эффективный способ реализации процедуры **быстрсорт**, показанный на рис. 9.3 под именем

```
быстрсорт( Спис, УпорСпис) :-
    быстрсорт2( Спис, УпорСпис-[] ).

быстрсорт2( [], Z-Z).

быстрсорт2( [X | Хвост], A1-Z2) :-
    разбиение( X, Хвост, Меньш, Больш),
    быстрсорт2( Меньш, A1-[X | A2] ),
    быстрсорт2( Больш, A2-Z2).
```

Рис. 9.3. Более эффективная реализация процедуры **быстрсорт** с использованием разностного представления списков. Отношение **разбиение(X, Спис, Меньш, Больш)** определено, как на рис. 9.2.

быстрсорт2. Здесь, как и раньше, процедура **быстрсорт** использует обычное представление списков, но в действительности сортировку выполняет более эффективная процедура **быстрсорт2**, использующая разностное представление. Эти две процедуры связаны между собой соотношением

```
быстрсорт( L, S) :-
    быстрсорт2( L, S-[] ).
```

Упражнения

9.5. Напишите процедуру слияния двух упорядоченных списков в один третий список. Например:

```
?- слить( [2,5,6,6,8], [1,3,5,9], L).
L = [1,2,3,5,5,6,6,8,9]
```

9.6. Программы сортировки, показанные на рис. 9.2 и 9.3, отличаются друг от друга способом представления списков. Первая из них использует обычное представление, в то время как вторая – разностное представление. Преобразование из одного представления в другое очевидно и может быть автоматизировано. Введите в программу рис. 9.2 необходимые изменения, чтобы преобразовать ее в программу рис. 9.3.

9.7. Наша программа **быстрсорт** в случае, когда исходный список уже упорядочен или почти упорядочен, работает очень неэффективно. Проанализируйте причины этого явления.

9.8. Существует еще одна хорошая идея относительно механизма сортировки списков, позволяющая избавиться от недостатков программы **быстрсорт**, а именно: разбить список на два меньших списка, отсортировать их, а затем слить вместе. Итак, для того, чтобы отсортировать список L, необходимо

- разбить L на два списка L1 и L2 примерно одинаковой длины;
- произвести сортировку списков L1 и L2, получив списки S1 и S2;
- слить списки S1 и S2, завершив на этом сортировку списка L.

Реализуйте этот принцип сортировки и сравните его эффективность с эффективностью программы **быстрсорт**.

9.2. Представление множеств двоичными деревьями

Списки часто применяют для представления множеств. Такое использование списков имеет тот недостаток, что проверка принадлежности элемента множеству оказывается довольно неэффективной. Обычно предикат **принадлежит(X, L)** для проверки принадлежности X к L программируют так:

```
принадлежит( X, [X  | L] ).
принадлежит( X, [ Y  | L] ) :-
    принадлежит( X, L).
```

Для того, чтобы найти X в списке L, эта процедура последовательно просматривает список элемент за элементом, пока ей не встретится либо элемент X, либо конец списка. Для длинных списков такой спо-

соб крайне неэффективен.

Для облегчения более эффективной реализации отношения принадлежности применяют различные древовидные структуры. В настоящем разделе мы рассмотрим двоичные деревья.

Двончное дерево либо пусто, либо состоит из следующих трех частей:

- корень
- левое поддерево
- правое поддерево

Корень может быть чем угодно, а поддеревья должны сами быть двоичными деревьями. На рис. 9.4 показано представление миожества [a, b, c, d] двоичным деревом. Элементы множества хранятся в виде вершин дерева. Пустые поддеревья на рис. 9.4 не показаны. Например, вершина b имеет два поддерева, которые оба пусты.

Существует много способов представления двоичных деревьев на Прологе. Одна из простых возможностей – сделать корень главным функтором соответствующего терма, а поддеревья – его аргументами. Тогда дерево рис. 9.4 примет вид

```
a( b, c(d) )
```

Такое представление имеет среди прочих своих недостатков то слабое место, что для каждой вершины дерева нужен свой функтор. Это может привести к неприятностям, если вершины сами являются структурными объектами.

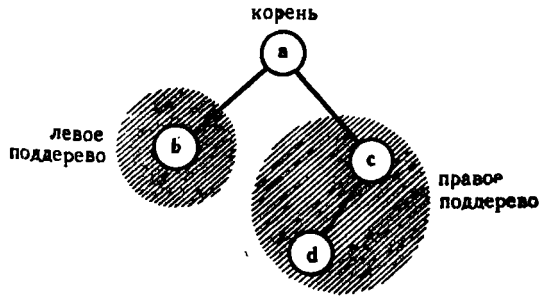


Рис. 9.4. Двоичное дерево.

Существует более эффективный и более привычный способ представления двоичных деревьев: нам нужен специальный символ для обозначения пустого дерева и функтор для построения непустого дерева из трех компонент (корня и двух поддеревьев). Относительно функтора и специального символа сделаем следующий выбор:

- Пусть атом **nil** представляет пустое дерево.
- В качестве функтора примем **дер**, так что дерево с корнем **X**, левым поддеревом **L** и правым поддеревом **R** будет иметь внд терм: **дер(L, X, R)** (см. рис. 9.5).

В этом представлении дерево рис. 9.4 выглядит как

```
дер( дер( nil, b, nil), a,
     дер( дер( nil, d, nil), c, nil) ).
```

Теперь рассмотрим отношение принадлежности, которое будем обозначать **внутри**. Цель

```
внутри( X, T)
```

истинна, если **X** есть вершина дерева **T**. Отношение **внутри** можно определить при помощи следующих правил:

X есть вершина дерева T, если

- корень дерева T совпадает с X, или
- X – это вершина из левого поддерева, или
- X – это вершина из правого поддерева.

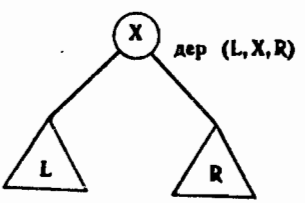


Рис. 9.5. Представление двоичных деревьев.

Эти правила непосредственно транслируются на Пролог следующим образом:

```
внутри( X, дер( –, X, –) ).
внутри( X, дер( L, –, –) ) :-
    внутри( X, L).
внутри( X, дер( –, –, R) ) :-
    внутри( X, R).
```

Очевидно, что цель

```
внутри( X, nil)
```

терпит неудачу при любом X.

Посмотрим, как ведет себя наша процедура. Рассмотрим рис. 9.4. Цель

```
внутри( X, T)
```

используя механизм возвратов, находит все элементы данных, содержащиеся в множестве, причем обнаруживает их в следующем порядке:

```
X = a; X = b; X = c; X = d
```

Теперь рассмотрим вопрос об эффективности. Цель

```
внутри( a, T)
```

достигается сразу же после применения первого предложения процедуры **внутри**. С другой стороны, цель

```
внутри( d, T)
```

будет успешно достигнута только после нескольких рекурсивных обращений. Аналогично цель

```
внутри( e, T)
```

потерпит неудачу только после того, как будет просмотрено все дерево в результате рекурсивного применения процедуры **внутри** ко *всем* поддеревьям дерева T.

В этом последнем случае мы видим такую же неэффективность, как если бы мы представили множество просто списком. Положение можно улучшить, если между элементами множества существует отношение порядка. Тогда можно упорядочить данные в дереве слева направо в соответствии с этим отношением.

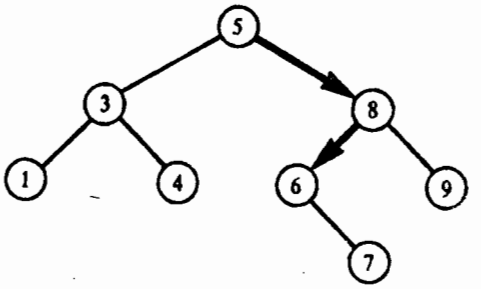


Рис. 9.6. Двоичный справочник. Элемент 6 найден после прохода по отмеченному пути 5—>8—>6.

Будем говорить, что непустое дерево **дер(Лев, X, Прав)** упорядочено слева направо, если

(1) все вершины левого поддерева **Лев** меньше X;

(2) все вершины правого поддерева **Прав** больше X;

(3) оба поддерева упорядочены.

Будем называть такое двоичное дерево *двоичным справочником*. Пример показан на рис.9.6.

Преимущество упорядочивания состоит в том, что для поиска некоторого объекта в двоичном справочнике всегда достаточно просмотреть не более одного поддерева. Экономия при поиске объекта X достигается за счет того, что, сравнив X с корнем, мы можем сразу же отбросить одно из поддеревьев. Например, пусть мы ищем элемент 6 в дереве, изображенном на рис. 9.6. Мы начинаем с корня 5, сравниваем 6 с 5, получаем 6 > 5. Поскольку все элементы данных в левом поддереве должны быть меньше, чем 5, единственная область, в которой еще осталась возможность найти элемент 6, – это правое поддерево. Продолжаем поиск в правом поддереве, переходя к вершине 8, и т.д.

Общий метод поиска в двоичном справочнике состоит в следующем:

Для того, чтобы найти элемент X в справочнике Д, необходимо:

- если X – это корень справочника Д, то считать, что X уже найден, иначе
- если X меньше, чем корень, то искать X в левом поддереве, иначе
- искать X в правом поддереве;
- если справочник Д пуст, - то поиск терпит неудачу.

Эти правила запрограммированы в виде процедуры, показанной на рис. 9.7. Отношение **больше(X, Y)**, означает, что X больше, чем Y. Если элемеиты, хранимые в дереве, – это числа, то под «больше, чем» имеется в виду просто X > Y.

Существует способ использовать процедуру **внутри** также и для *построения* двоичного справочника. Например, справочник Д, содержащий элементы 5, 3, 8, будет построен при помощи следующей последовательности целей:

```
?- внутри( 5, Д), внутри( 3, Д), внутри( 8, Д).
Д = дер( дер( Д1, 3, Д2), 5, дер( Д3, 8, Д4) ).
```

Переменные Д1, Д2, Д3 и Д4 соответствуют четырем неопределенным поддеревьям. Какими бы они ни были, все равно дерево Д будет содержать заданные элементы 3, 5 и 8. Структура построенного дерева зависит от того порядка, в котором указываются цели (рис. 9.8).

```
внутри( X, дер( _, X, _) ).

внутри( X, дер( Лев, Корень, Прав) ) :-
    больше( Корень, X),          % Корень больше, чем X
    внутри( X, Лев).             % Поиск в левом поддереве

внутри( X, дер( Лев, Корень, Прав) ) :-
    больше( X, Корень),          % X больше, чем корень
    внутри( X, Прав).            % Поиск в правом поддереве
```

Рис. 9.7. Поиск элемента X в двоичном справочнике.

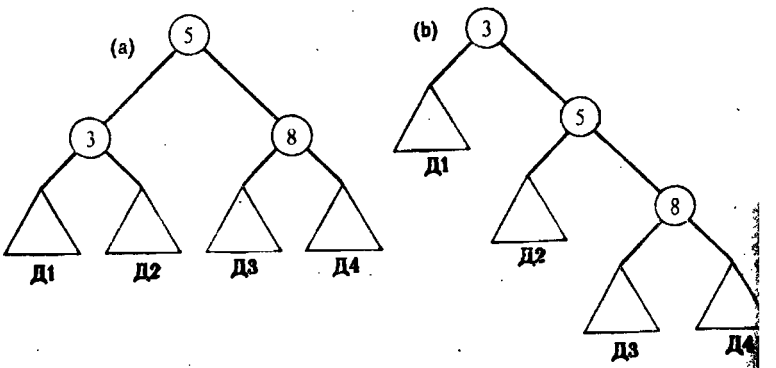


Рис. 9.8. (a) Дерево Д, построенное как результат достижения целей: **внутри(5,Д)**, **внутри(3,Д)**, **внутри(8,Д)**. (b) Дерево, полученное при другом порядке целей: **внутри(5,Д)**, **внутри(3,Д)**, **внутри(8, Д)**.

Здесь уместно сделать несколько замечаний относительно эффективности поиска в справочниках. Вообще говоря, поиск элемента в справочнике эффективнее, чем поиск в списке. Но насколько? Пусть *n* – число элементов множества. Если множество представлено списком, то ожидаемое время поиска будет пропорционально его длине *n*. В средием нам придется просмотреть примерио половину списка. Если множество представлено двоичиым деревом, то время поиска будет пропорционально глубиие дерева. Глубииа дерева – это длииа самого длинного пути между корнем и листом дерева. Однако следует помнить, что глубина дерева зависит от его формы.

Мы говорим, что дерево (приближенно) сбалансироваио, если для каждой вершины дерева соответствующие два поддерева содержат примерно равное число элементов. Если дерево хорошо сбалаисироваио, то его глубина пропорциоиальиа log *n*. В этом случае мы говорим, что дерево имеет логарифмическую сложность. Сбалансированный справочиик лучше списка иастолько же, насколько log *n* меньше *n*. К сожаленню, это верно только для приближенно сбалансированиого дерева. Если происходит разбалансировка дерева, то производительность падает. В случае полиостью разбалансированиых деревьев, дерево

фактически превращается в список. Глубина дерева в этом случае равна n, а производительность поиска оказывается столь же низкой, как и в случае списка. В связи с этим мы всегда заинтересованы в том, чтобы справочники были сбалансированы. Методы достижения этой цели мы обсудим в гл. 10.

Упражнения

9.9. Определите предикаты

двдерево(Объект)
справочник(Объект)

распознающие, является ли **Объект** двоичным деревом или двоичным справочником соответственно. Используйте обозначения, введенные в данном разделе.

9.10. Определите процедуру

глубина(ДвДерево, Глубина)

вычисляющую глубину двоичного дерева в предположении, что глубина пустого дерева равна **0**, а глубина одноэлементного дерева равна 1.

9.11. Определите отношение

линеаризация(Дерево, Список)

соответствующее «выстраиванию» всех вершин дерева в список.

9.12. Определите отношение

максэлемент(Д, Элемент)

таким образом, чтобы переменная **Элемент** приняла значение наибольшего из элементов, хранящихся в дереве **Д**.

9.13. Внесите изменения в процедуру

внутри(Элемент, ДвСправочник)

добавив в нее третий аргумент **Путь** таким образом, чтобы можно было бы получить путь между корнем справочника и указанным элементом.

9.3. Двоичные справочники: добавление и удаление элемента

Если мы имеем дело с динамически изменяемым множеством элементов данных, то нам может понадобиться внести в него новый элемент или удалить из него один из старых. В связи с этим набор основных операций, выполняемых над множеством S, таков:

```
внутри( X, S)          % X содержится в S
добавить( S, X, S1)    % Добавить X к S, результат - S1
удалить( S, X, S1)     % Удалить X из S, результат - S1
```

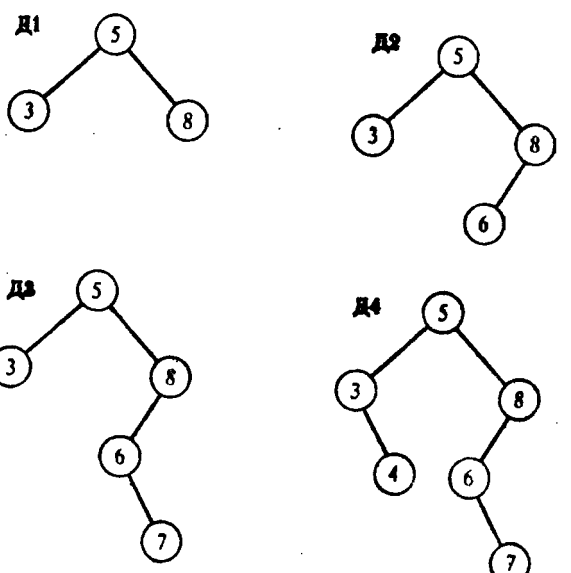


Рис. 9.9. Введение в двоичный справочник нового элемента на уровне листьев. Показанные деревья соответствуют следующей последовательности вставок:
добавить(Д1,6,Д2), добавить(Д2,6,Д3), добавить(Д3,6,Д4)

```
доблист( nil, X, дер( nil, X, nil) ).

доблист( дер(Лев,X,Прав),X,дер(Лев,X,Прав) ).

доблист(дер(Лев,Кор,Прав),X,дер(Лев1,Кор,Прав)) :-
    больше( Кор, X),
    доблист( Лев, X, Лев1).

доблист(дер(Лев,Кор,Прав),X,дер(Лев,Кор,Прав1)) :-
    больше( X, Кор),
    доблист( Прав, X, Прав1).
```

Рис. 9.10. Вставление в двоичный справочник нового элемента в качестве листа.

Определим отношение *добавить*. Простейший способ: ввести новый элемент на самый нижний уровень дерева, так что он станет его листом. Место, на которое помещается новый элемент, выбрать таким образом, чтобы не нарушить упорядоченность дерева. На рис. 9.9 показано, какие изменения претерпевает дерево в процессе введения в него новых элементов. Назовем такой метод вставления элемента в множество

доблист(Д, X, Д1)

Правила добавления элемента на уровне листьев таковы:

- Результат добавления элемента X к пустому дереву есть дерево **дер(nil, X, nil)**.
- Если X совпадает с корнем дерева Д, то Д1 = Д (в множестве не допускается дублирования элементов).
- Если корень дерева Д больше, чем X, то X вносится в левое поддерево дерева Д; если корень меньше, чем X, то X вносится в правое поддерево.

На рис. 9.10 показана соответствующая программа.

Теперь рассмотрим операцию *удалить*. Лист дерева удалить легко, однако удалить какую-либо внутреннюю вершину – дело не простое. Удаление листа можно на самом деле определить как операцию, обратную

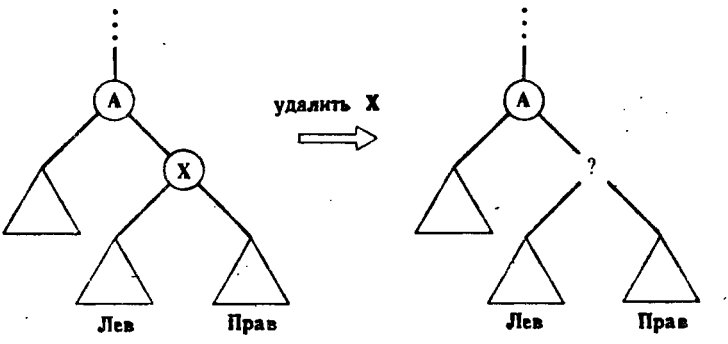


Рис. 9.11. Удаление X из двоичного справочника. Возникает проблема наложения "заплаты" на место удаленного элемента X.

операции добавления листа:

```
удлист( Д1, X, Д2) :-
    доблист( Д2, X, Д1).
```

К сожалению, если X – это внутренняя вершина, то такой способ не работает, поскольку возникает проблема, иллюстрацией к которой служит рис. 9.11. Вершина X имеет два поддерева **Лев** и **Прав**. После удаления вершины X в дереве образуется «дыра», и поддеревья **Лев** и **Прав** теряют свою связь с остальной частью дерева. К вершине A оба эти поддерева присоединить невозможно, так как вершина A способна принять только одно из них.

Если одно из поддеревьев **Лев** и **Прав** пусто, то существует простое решение: подсоединить к A непустое поддерево. Если же оба поддерева непусты,

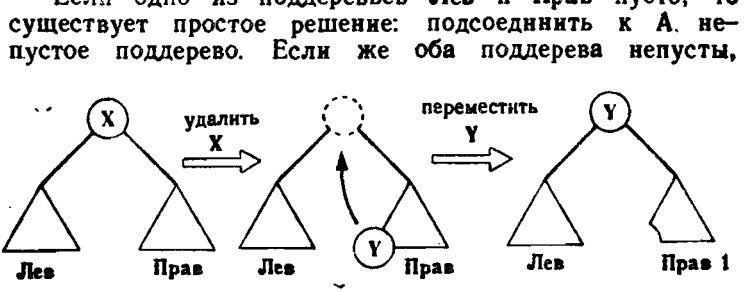


Рис. 9.12. Заполнение пустого места после удаления X.

то можно использовать следующую идею (рис. 9.12): если самую левую вершину Y поддерева Прав переместить из ее текущего положения вверх и заполнить ею пробел, оставшийся после X, то упорядоченность дерева не нарушится. Разумеется, та же идея сработает и в симметричном случае, когда перемещается самая правая вершина поддерева Лев.

На рис. 9.13 показана программа, реализующая операцию удаления элементов в соответствии с изложенными выше соображениями. Основную работу по перемещению самой левой вершины выполняет отношение

удмин(Дер, Y, Дер1)

Здесь Y – минимальная (т.е. самая левая) вершина дерева Дер, а Дер1 – то, во что превращается дерево Дер после удаления вершины Y.

Существует другой, элегантный способ реализации операции *добавить* и *удалить*. Отношение *добавить* можно сделать недетерминированным в том смысле, что новый элемент вводится на произвольный уровень дерева, а не только на уровень листьев. Правила таковы:

```
уд( дер( nil, X, Прав), X, Прав).

уд( дер( Лев, X, nil), X, Лев).

уд( дер(Лев,X,Прав), X, дер(Лев,Y,Прав1) ) :-
      удмин( Прав, Y, Прав1).

уд( дер(Лев,Кор,Прав), X, дер(Лев1,Кор,Прав) ) :-
      больше( Кор, X),
      уд( Лев, X, Лев1).

уд( дер(Лев,Кор,Прав), X, дер(Лев,Кор,Прав1) ) :-
      больше( X, Кор),
      уд( Прав, X, Прав1).

удмин( дер( nil, Y, Прав), Y, Прав).

удмин( дер(Лев,Кор,Прав),Y,дер(Лев1,Кор,Прав) ) :-
      удмин( Лев, Y, Лев1).
```

Рис. 9.13. Удаленне элемента из двоичного справочника.

Для того, чтобы добавить X в двоичный справочник Д, необходимо одно из двух:

- добавить X на место корня дерева (так, что X станет новым корнем) или
- если корень больше, чем X, то внести X в левое поддерево, иначе – в правое поддерево.

Трудным моментом здесь является введение X на место корня. Сформулируем эту операцию в виде отношения

добкор(Д, X, X1)

где X – новый элемент, вставляемый вместо корня в Д, а Д1 – новый справочник с корнем X. На рис. 9.14 показано, как соотносятся X, Д и Д1. Остается вопрос: что из себя представляют поддеревья L1 и L2 (или, соответственно, R1 и R2) на рис. 9.14?

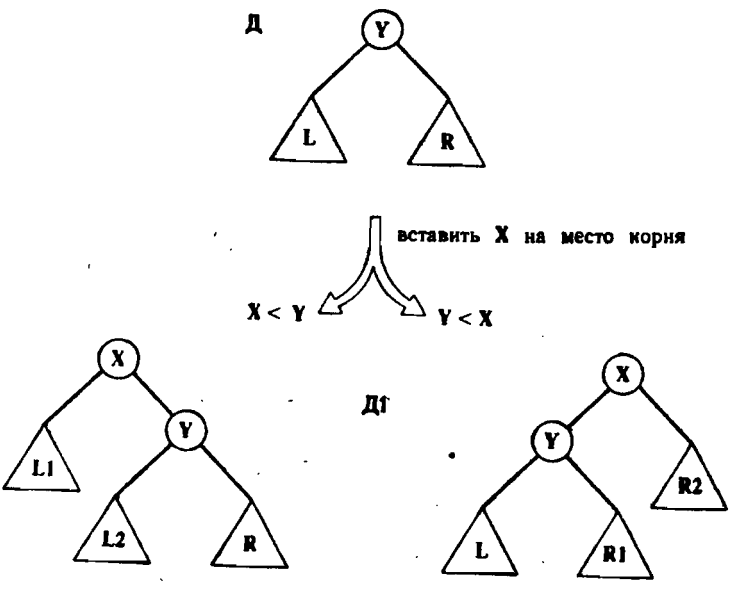


Рис. 9.14. Внесение X в двоичный справочник в качестве корня.

Ответ мы получим, если учтем следующие ограничения на L1, L2:

- L1 и L2 – двоичные справочники;
- множество всех вершин, содержащихся как в L1, так и в L2, совпадает с множеством вершин справочника L;
- все вершины из L1 меньше, чем X; все вершины из L2 больше, чем X.

Отношение, которое способно наложить все эти ограничения на L1, L2, – это как раз и есть наше отношение добкор. Действительно, если бы мы вводили X в L на место корня, то поддеревьями результирующего дерева как раз и оказались бы L1 и L2. В терминах Пролога L1 и L2 должны быть такими, чтобы достигалась цель

```
добкор( L, X, дер( L1, X, L2) ).
```

Те же самые ограничения применимы к R1, R2:

```
добкор( R, X, дер( R1, X, R2) ).
```

```
добавить( Д, X, Д1) :-                % Добавить X на место корня
    добкор( Д, X, Д1).

добавить( дер( L, Y, R), X, дер( L1, Y, R) ) :-
    больше( Y, X),                    % Ввести X в левое поддерево
    добавить( L, X, L1).

добавить( дер( L, Y, R), X, дер( L, Y, R1) ) :-
    больше( X, Y),                    % Ввести X в правое поддерево
    добавить( R, X, R1).

добкор( nil, X; дер( nil, X, nil) ).
                                      % Ввести X в пустое дерево

добкор( дер(L,Y,R),X,дер(L1,X,дер(L2,Y,R) )) :-
    больше( Y, X),
    добкор( L, X, дер( L1, X, L2) ).

добкор( дер(L,Y,R),X,дер(дер(L,Y,R1),X,R2) ) :-
    больше( X, Y),
    добкор( R, X, дер( R1, X, R2) ).
```

Рис. 9.15. Внесение элемента на произвольный уровень двоичного справочника.

На рис. 9.15 показана программа для «недетерминированного» добавления элемента в двоичный справочник.

Эта процедура обладает тем замечательным свойством, что в ее не заложено никаких ограничений на уровень дерева, в который вносится новый элемент. В связи с этим операцию *добавить* можно использовать «в обратном направлении» для удаления элемента из справочника. Например, приведенная ниже последовательность целей строит справочник Д, содержащий элементы 3, 5, 1, 6, а затем удаляет из него элемент 5, после чего получается справочник ДД:

```
добавить( nil, 3, Д1), добавить( Д1, 5, Д2),
добавить( Д2, 1, Д3), добавить( Д3, 6, Д),
добавить( ДД, 5, Д).
```

9.4. Отображение деревьев

Так же, как и любые объекты данных в Прологе, двоичное дерево Т может быть непосредственно выведено на печать при помощи встроенной процедуры write. Однако цель

```
write( Т)
```

хотя и отпечатает всю информацию, содержащуюся в дереве, но действительная структура дерева никак при этом не будет выражена графически. Довольно утомительная работа – пытаться представить себе структуру дерева, рассматривая прологовский терм, которым она представлена. Поэтому во многих случаях желательно иметь возможность отпечатать дерево в такой форме, которая графически соответствует его структуре.

Существует относительно простой способ это сделать. Уловка состоит в том, чтобы изображать дерево растущим слева направо, а не сверху вниз, как обычно. Дерево нужно повернуть влево таким образом, чтобы корень стал его крайним слева элементом, а листья сдвинулись вправо (рис. 9.16).

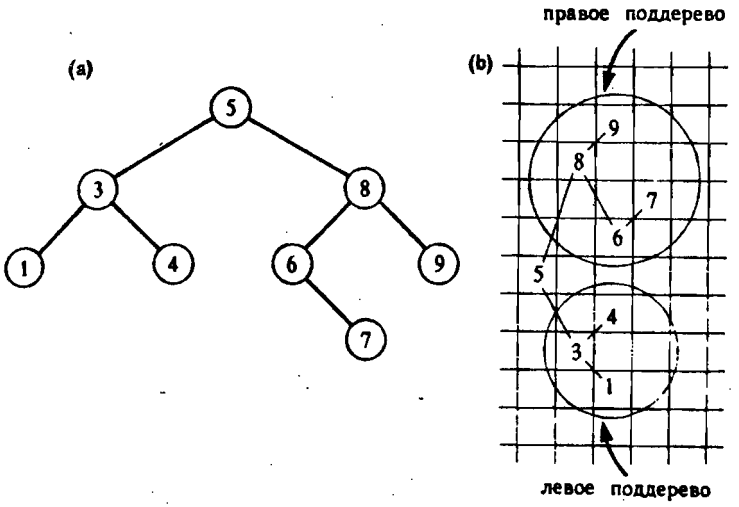


Рис. 9.16. (a) Обычное изображение дерева. (b) То же дерево, отпечатанное процедурой **отобр** (дуги добавлены для ясности).

Давайте определим процедуру

отобр(T)

так, чтобы она отображала дерево в форме, показанной на рис. 9.16. Принцип работы этой процедуры:

Для того, чтобы отобразить непустое дерево T, необходимо:

(1) отобразить правое поддерево дерева T с отступом вправо на расстояние H;

(2) отпечатать корень дерева T;

(3) отобразить левое поддерево дерева T с отступом вправо на расстояние H.

Величина отступа H, которую можно выбирать по желанию, – это дополнительный параметр при отображении деревьев. Введем процедуру

отобр2(Т, Н)

печатающую дерево Т с отступом на Н пробелов от левого края листа. Связь между процедурами **отобр** и **отобр2** такова:

отобр(Т) :- отобр2(Т, 0).

На рис. 9.17 показана программа целиком. В этой программе предусмотрен сдвиг на 2 позиции для каждого уровня дерева. Описанный принцип отображения можно легко приспособить для деревьев других типов.

```
отобр( Т) :-
    отобр2( Т, 0).

отобр2( nil, _ ).

отобр2( дер( L, X, R), Отступ) :-
    Отступ2 is Отступ + 2,
    отобр2( R, Отступ2),
    tab( Отступ), write( X), nl,
    отобр( L, Отступ2).
```

Рис. 9.17. Отображение двоичного дерева.

Упражнение

9.14. Наша процедура изображает дерево, ориентируя его необычным образом: корень находится слева, а листья – справа. Напишите (более сложную) процедуру для отображения дерева, ориентированного обычным образом, т.е. с корнем наверху и листьями внизу.

9.5. Графы

9.5.1. Представление графов

Графы используются во многих приложениях, например для представления отношений, ситуаций или структур задач. Граф определяется как множество *вершин* вместе с множеством *ребер*, причем каждое ребро задается парой вершин. Если ребра направлены, то их также называют *дугами*. Дуги задаются *упорядоченными* парами. Такие графы называются *направленными*. Ребрам можно приписывать стоимости, имена или метки произвольного вида, в зависимости от конкретного приложения. На рис. 9.18 показаны примеры графов.

В Прологе графы можно представлять различными способами. Один из них – каждое ребро записывать в виде отдельного предложения. Например, графы, показанные на рис. 9.18, можно представить в виде следующего множества предложений:

```
связь( a, b).
связь( b, c).
...

дуга( s, t, 3).
дуга( t, v, 1).
дуга( u, t, 2).
...
```

Другой способ – весь граф представлять как один объект. В этом случае графу соответствует пара множеств – множество вершин и множество ребер. Каждое множество можно задавать при помощи списка, каждое ребро – парой вершин. Для объединения двух множеств в пару будем применять функтор граф, а для записи ребра – функтор р. Тогда (ненаправленный) граф рис. 9.18 примет вид:

```
G1 = граф( [a,b,c,d],
           [p(a,b), p(b,d), p(b,c), p(c,d)] )
```

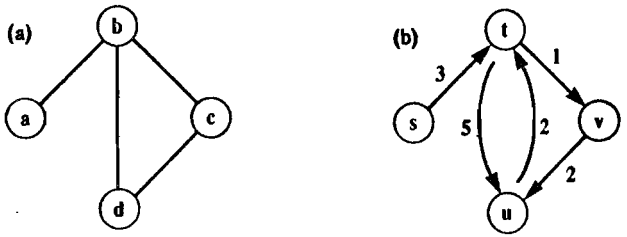


Рис. 9.18. (a) Граф. (b) Направленный граф. Каждой дуге приписана ее стоимость.

Для представления направленного графа (рис. 9.18), применив функторы **дграф** и **д** (для дуг), получим

```
G2 = дграф( [s,t,u,v],
            [д(s,t,3), д(t,v,1), д(t,u,5),
             д(u,t,2), д(v,u,2) ] )
```

Если каждая вершина графа соединена ребром еще по крайней мере с одной вершиной, то в представлении графа можно опустить множество вершин, поскольку оно неявным образом содержится в списке ребер.

Еще один способ представления графа – связать с каждой вершиной список смежных с ней вершин. В этом случае граф превращается в список пар, каждая из которых состоит из вершины плюс ее список смежности. Наши графы (рис. 9.18), например, можно представить как

```
G1 = [ a->[b],b->[a,c,d],c->[b,d],d->[b,c] ]
G2 = [ s->[t/3],t->[u/5,v/1],u->[t/2],v->[u/2]]
```

Здесь символы '->' и '/' – инфиксные операторы.

Какой из способов представления окажется более удобным, зависит от конкретного приложения, а также от того, какие операции имеется в виду выполнять над графами. Вот типичные операции:

- найти путь между двумя заданными вершинами;
- найти подграф, обладающий некоторыми заданными свойствами.

Примером последней операции может служить построение остовного дерева графа. В последующих разделах мы рассмотрим некоторые простые программы для поиска пути в графе и построения остовного дерева.

9.5.2. Поиск пути в графе

Пусть G – граф, а A и Z – две его вершины. Определим отношение

путь(A, Z, G, P)

где P – ациклический путь между A и Z в графе G. Если G – граф, показанный в левой части рис. 9.18, то верно:

путь(a, d, G, [a,b,d])
путь(a, d, G, [a,b,c,d])

Поскольку путь не должен содержать циклов, любая вершина может присутствовать в пути не более одного раза. Вот один из методов поиска пути:

Для того, чтобы найти ациклический путь P между A и Z в графе G, необходимо:

Если A = Z , то положить P = [A], иначе найти ациклический путь P1 из произвольной вершины Y в Z, а затем найти путь из A в Y, не содержащий вершин из P1.

В этой формулировке неявно предполагается , что существует еще одно отношение, соответствующее поиску пути со следующим ограничением: путь не должен проходить через вершины из некоторого подмножества (в данном случае P1) множества всех вершин графа. В связи с этим мы определим еще одну процедуру:

путь1(A, P1, G, P)

Аргументы в соответствии с рис. 9.19 имеют следующий смысл:

- A – некоторая вершина,

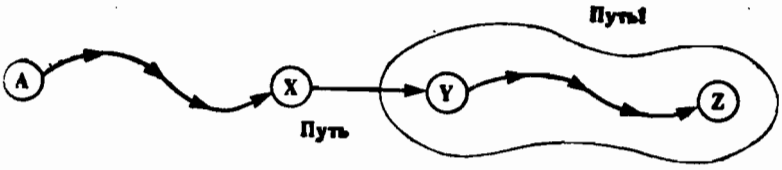


Рис. 9.19. Отношение путь1: Путь–это путь между A и Z, в своей заключительной части он перекрывается с Путь1.

- G – граф,
- P1 – путь в G,
- P – ациклический путь в G, идущий из A в начальную вершину пути P1, а затем – вдоль пути P1 вплоть до его конца.

Между путь и путь1 имеётся следующее соотношение:

путь(A, Z, G, P) :- путь1(A, [Z], G, P).

На рис. 9.19 показана идея рекурсивного определения отношения путь1. Существует «граничный» случай, когда начальная вершина пути P1 (Y на рис. 9.19) совпадает с начальной вершиной A пути P. Если же начальные вершины этих двух путей не совпадают, то должна существовать такая вершина X, что

(1) Y – вершина, смежная с X,
(2) X не содержится в P1 и
(3) для P выполняется отношение
путь1(A, [X|P1], G, P).

```
путь( A, Z, Граф, Путь) :-
    путь1( A, [Z], Граф, Путь).

путь1( A, [A | Путь1], _, [A |Путь1] ).

путь1( A, [Y | Путь1], Граф, Путь) :-
    смеж( X, Y, Граф),
    принадлежит( X, Путь1),  % Условие отсутствия цикла
    путь1( A, [ X, Y | Путь1], Граф, Путь).
```

Рис. 9.20. Поиск в графе Граф ациклического пути Путь из A в Z.

На рис. 9.20 программа показана полностью. Здесь **принадлежит** – отношение принадлежности элемента списку. Отношение

```
смеж( X, Y, G)
```

означает, что в графе G существует дуга, ведущая из X в Y. Определение этого отношения зависит от способа представления графа. Если G представлен как пара множеств (вершин и ребер)

```
G = граф( Верш, Реб)
```

то

```
смеж( X, Y, граф( Верш, Реб) ) :-
    принадлежит( р( X, Y), Реб);
    принадлежит( р( Y, X), Реб).
```

Классическая задача на графах – поиск Гамильтонова цикла, т.е. ациклического пути, проходящего через все вершины графа. Используя отношение **путь**, эту задачу можно решить так:

```
гамильтон( Граф, Путь) :-
    путь( _, _, Граф, Путь),
    всевершины( Путь, Граф).

всевершины( Путь, Граф) :-
    not (вершина( В, Граф),
         not принадлежит( В, Путь) ).
```

Здесь **вершина(В, Граф)** означает: **В** – вершина графа **Граф**.

Каждому пути можно приписать его стоимость. Стоимость пути равна сумме стоимостей входящих в него дуг. Если дугам не приписаны стоимости, то тогда, вместо стоимости, говорят о длине пути.

Для того, чтобы наши отношения **путь** и **путь1** могли работать со стоимостями, их нужно модифицировать, введя дополнительный аргумент для каждого пути:

```
путь( A, Z, G, P, C)
путь1( A, P1, C1, G, P, C)
```

Здесь C – стоимость пути P, а C1 – стоимость пути P1. В отношении **смеж** также появится дополнительный аргумент, стоимость дуги. На рис. 9.21 показана программа поиска пути, которая строит путь и вычисляет его стоимость.

```
путь( A, Z, Граф, Путь, Ст) :-
    путь1( A, [Z], 0, Граф, Путь, Ст).

путь1( A, [A | Путь1], Ст1, Граф, [A |Путь1], Ст).

путь1( A, [Y | Путь1], Ст1, Граф, Путь, Ст) :-
    смеж( X, Y, СтXY, Граф),
    not принадлежит( X, Путь1),
    Ст2 is Ст1 + СтXY,
    путь1( A, [ X, Y | Путь1], Ст2, Граф, Путь, Ст).
```

Рис. 9.21. Поиск пути в графе: **Путь** – путь между A и Z в графе **Граф** стоимостью Ст.

Эту процедуру можно использовать для нахождения пути минимальной стоимости. Мы можем построить путь минимальной стоимостн между вершинамн **Верш1**, **Верш2** графа **Граф**, задав цели

```
путь( Верш1, Верш2, Граф, МинПуть, МинСт),
not ( путь(Верш1,Верш2,Граф,_,Ст), Ст<МинСт )
```

Аналогично можно среди всех путей между вершинами графа найти путь максимальной стоимости, задав цели

```
путь( _, _, Граф, МаксПуть, МаксСт),
not ( путь( _, _, Граф, _, Ст), Ст > МаксСт)
```

Заметим, что приведенный способ поиска максимальных и минимальных путей крайне неэффективен, так как он предполагает просмотр всех возможных путей и потому не подходит для больших графов из-за своей высокой временной сложности. В искусственном интеллекте задача поиска пути возникает довольно часто. В главах 11 и 12 мы изучим более сложные методы нахождения оптимальных путей.

9.5.3. Построение остовного дерева

Граф называется *связным*, если между любыми двумя его вершинами существует путь. Пусть G = (V, E) – связный граф с множеством вершин V и множеством

ребер E. *Остовное дерево* графа G – это связный граф T = (V, E′), где E′ – подмножество E такое, что

(1) T – связный граф,
(2) в T нет циклов.

Выполнение этих двух условий гарантирует то, что T – дерево. Для графа, изображенного в левой части рис. 9.18, существует три остовных дерева, соответствующих следующим трем спискам ребер:

```
Дер1 = [a–b, b–c, c–d]
Дер2 = [a–b, b–d, d–c]
Дер3 = [a–b, b–d, b–c]
```

Здесь каждый терм вида X–Y обозначает ребро, соединяющее вершины X и Y. В качестве корня можно взять любую из вершин, указанных в списке. Остовные деревья представляют интерес, например в задачах проектирования сетей связи, поскольку они позволяют, имея минимальное число линий, установить связь между любыми двумя узлами, соответствующими вершинам графа.

Определим процедуру

остдерево(G, T)

где T – остовное дерево графа G. Будем предполагать, что G – связный граф. Можно представить себе алгоритмический процесс построения остовного дерева следующим образом. Начать с пустого множества ребер и постепенно добавлять новые ребра, постоянно следя за тем, чтобы не образовывались циклы. Продолжать этот процесс до тех пор, пока не обнаружится, что нельзя присоединить ни одного ребра, поскольку любое новое ребро порождает цикл. Полученное множество ребер будет остовным деревом. Отсутствие циклов можно обеспечить, если придерживаться следующего простого правила: ребро присоединяется к дереву только в том случае, когда одна из его вершин уже содержится в строящемся дереве, а другая пока еще не включена в него. Программа, реализующая эту идею, показана на рис. 9.22. Основное отношение, используемое в этой программе, – это

расшмрить(Дер1, Дер, G)

Здесь все три аргумента – множества ребер. G –

```
% Построение остовного дерева графа
%
% Деревья и графы представлены списками
% своих ребер, например:
%       Граф = [a-b, b-c, b-d, c-d]

остдерево( Граф, Дер) :-    % Дер - остовное дерево Граф'а
    принадлежит( Ребро, Граф),
    расширить( [Ребро], Дер, Граф).

расширить( Дер1, Дер, Граф) :-
    добребро( Дер1, Дер2, Граф),
    расширить( Дер2, Дер, Граф).

расширить( Дер, Дер, Граф) :-
    not добребро( Дер, _, Граф).
                % Добавление любого ребра приводит к циклу

добребро( Дер, [A-B | Дер], Граф) :-
    смеж( A, B, Граф),          % A и B - смежные вершины
    вершина( A, Дер),           % A содержится в Дер
    не вершина( B, Дер).        % A-B не порождает цикла

смеж( A, B, Граф) :-
    принадлежит( A-B, Граф);
    принадлежит( B-A, Граф).

вершина( A, Граф) :-            % A содержится в графе, если
    смеж( A, _, Граф).          % A смежна какой-нибудь вершине
```

Рис. 9.22. Построение остовного дерева: алгоритмический подход. Предполагается, что **Граф** – связный граф.

связный граф; **Дер1** и **Дер** – два подмножества G, являющиеся деревьями. **Дер** – остовное дерево графа G, полученное добавлением некоторого (может быть пустого) множества ребер из G к **Дер1**. Можно сказать, что «**Дер1** расширено до **Дер**».

Интересно, что можно написать программу построения остовного дерева совершенно другим, полностью декларативным способом, просто формулируя на Прологе некоторые математические определения. Допустим, что как графы, так и деревья задаются списками своих ребер, как в программе рис. 9.22. Нам понадобятся следующие определения:

(1) T является остовным деревом графа G, если
- T – это подмножество графа G и
- T – дерево и
- T «накрывает» G, т.е. каждая вершина из G содержится также в T.

(2) Множество ребер T есть дерево, если
- T – связный граф и
- T не содержит циклов.

Эти определения можно сформулировать на Прологе (с использованием нашей программы путь из предыдущего раздела) так, как показано на рис. 9.23. Следует, однако, заметить, что эта программа в таком ее виде не представляет практического интереса из-за своей неэффективности.

```
% Построение остовного дерева
% Графы и деревья представлены списками ребер.

остдерево( Граф, Дер) :-
    подмнож( Граф, Дер),
    дерево( Дер),
    накрывает( Дер, Граф).

дерево( Дер) :-
    связн( Дер),
    not имеетцикл( Дер).

связн( Дер) :-
    not (вершина( A, Дер), вершина( B, Дер),
         not путь( A, A, Дер, _) ).

имеетцикл( Дер) :-
    смеж( A, B, Дер),
    путь( A, B, Дер, [A, X, Y | _] ).   % Длина пути > 1

накрывает( Дер, Граф) :-
    not (вершина( A, Граф), not вершина( A, Дер) ).

подмнож( [], []).

подмнож( [ X | L], S) :-
    подмнож( L, L1),
    ( S = L1; S = [ X | L1] ).
```

Рис. 9.23. Построение остовного дерева: "декларативный подход". Отношения вершина и смеж см. на рис. 9.22.

Упражнение

9.15. Рассмотрите остовные деревья в случае, когда каждому ребру графа приписана его стоимость. Пусть *стоимость* остовного дерева определена как сумма стоимостей составляющих его ребер. Напишите программу построения для заданного графа его остовного дерева минимальной стоимости.

Резюме

В данной главе мы изучали реализацию на Прологе некоторых часто используемых структур данных и соответствующих операций над ними. В том числе

- Списки:
  - варианты представления списков
  - сортировка списков:
    - сортировка методом «пузырька»
    - сортировка со вставками
    - быстрая сортировка
    - эффективность этих процедур
- Представление множеств двоичными деревьями и двоичными справочниками:
  - поиск элемента в дереве
  - добавление элемента
  - удаление элемента
  - добавление в качестве листа или корня
  - сбалансированность деревьев и его связь с эффективностью этих операций
  - отображение деревьев
- Графы:
  - представление графов
  - поиск пути в графе
  - построение остовного дерева

Литература

В этой главе мы занимались такими важными темами, как сортировка и работа со структурами данных для представления множеств. Общее описание структур

данных, а также алгоритмов, запрограммированных в данной главе, можно найти, например, в Aho, Hopcroft and Ullman (1974, 1983) или Baase (1978). В литературе рассматривается также поведение этих алгоритмов, особенно их временная сложность. Хороший и краткий обзор соответствующих алгоритмов и результатов их математического анализа можно найти в Gonnet (1984).

Прологовская программа для внесения нового элемента на произвольный уровень дерева (раздел 9.3) была впервые показана автору М. Ван Эмденом (при личном общении).

Aho A. V., Hopcroft J. E. and Ullman J. D. (1974). *The Design and Analysis of Computer Algorithms.* Addison-Wesley. [Имеется перевод: Ахо А., Хопкрофт Дж. Построение и анализ вычислительных алгоритмов. Пер. с англ. – М.: Мир, 1979.]

Aho A. V., Hopcroft J. E. and Ullman J. D. (1983). *Data Structures and Algorithms.* Addison-Wesley.

Baase S. (1978). *Computer Algorithms.* Addison-Wesley.

Gonnet G. H. (1984). *Handbook of Algorithms and Data Structures.* Addison-Wesley.

10 УСОВЕРШЕНСТВОВАННЫЕ МЕТОДЫ ПРЕДСТАВЛЕНИЯ МНОЖЕСТВ ДЕРЕВЬЯМИ

В данной главе мы рассмотрим усовершенствованные методы представления множеств при помощи деревьев. Основная идея состоит в том, чтобы поддерживать сбалансированность или приближенную сбалансированность дерева, с тем чтобы избежать вырождения его в список. Механизмы балансировки деревьев гарантируют, даже в худшем случае, относительно быстрый доступ к элементам данных, хранящихся в дереве, при логарифмическом порядке времени доступа. В этой главе изложено два таких механизма: двоично-троичные (кратко, 2-3) деревья и AVL-деревья. (Для изучения остальных глав понимание данной главы не обязательно.)

10.1. Двоично-троичные справочники

Двоичное дерево называют хорошо сбалансированным, если оба его поддерева имеют примерно одинаковую глубину (или размер) и сами сбалансированы. Глубина сбалансированного дерева приближенно равна $\log n$, где n – число вершин дерева. Время, необходимое для вычислений, производимых отношениями внутри, добавить и удалить над двоичными справочниками, пропорционально глубине дерева. Таким образом, в случае двоичных справочников это время имеет порядок $\log n$. Логарифмический рост сложности алгоритма, проверяющего принадлежность элемента множеству, – это определенное достижение по сравнению со списковым представлением, поскольку в последнем случае мы имеем линейный рост сложности

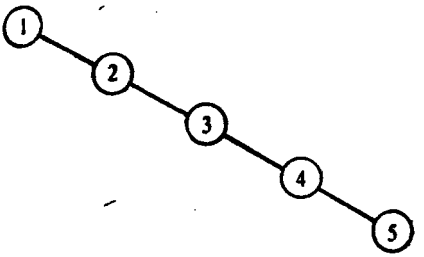


Рис.10.1. Полностью разбалансированный двоичный справочник. Производительность его та же, что и у списка.

с ростом размера множества. Однако плохая сбалансированность дерева ведет к деградации производительности алгоритмов, работающих со справочником. В крайнем случае, двоичный справочник вырождается в список, как показано на рис. 10.1. Форма справочника зависит от той последовательности, в которой в него записываются элементы данных. В лучшем случае мы получаем хорошую балансировку и производительность порядка $\log n$, а в худшем – производительность будет порядка n. Анализ показывает, что в среднем сложность алгоритмов **внутри**, **добавить** и **удалить** сохраняет порядок $\log n$ в допущении, что все возможные входные последовательности равновероятны. Таким образом, средняя производительность, к счастью, оказывается ближе к лучшему случаю, чем к худшему. Существует, однако, несколько довольно простых механизмов, которые поддерживают хорошую сбалансированность дерева, вне зависимости от входной последовательности, формирующей дерево. Эти механизмы гарантируют производительность алгоритмов **внутри**, **добавить** и **удалить** порядка $\log n$ даже *в худшем случае*. Один из этих механизмов – двоично-троичные деревья (кратко, 2-3 деревья), а другой – AVL-деревья.

2-3 дерево определяется следующим образом: оно или пусто, или состоит из единственной вершины, или удовлетворяет следующим условиям:

- каждая внутренняя вершина имеет две или три дочерних вершины, и

- все листья дерева находятся на одном и том же уровне.

Двончио-троичиым (2-3) справочником называется 2-3 дерево, все элементы данных которого храиятся в листьях и упорядочены слева направо. На рис. 10.2 показан пример. Виутреиние вершины содержат меткн, равные минимальным элементам тех или иных своих поддеревьев, в соответствии со следующими правилами:

- если виутреиняя вершиа имеет два поддерева, то она содержит миимальиый элемент второго из них;
- еслн внутреиняя вершиа имеет три поддерева, то она содержит минимальиые элементы второго н третьего поддеревьев.

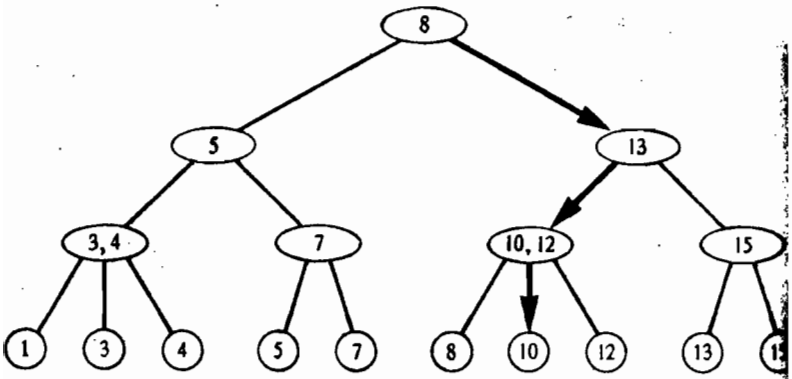


Рис. 10.2. 2-3 справочник. Отмеченый путь показывает процесс поиска элемента 10.

При поиске элемента X в 2-3 справочнике мы начинаем с корня и двигаемся в иаправлеиии самого нижнего уровня, руководствуясь при этом метками внутренних вершии дерева. Пусть корень содержит меткн M1 и M2, тогда

- если X < M1, то поиск продолжается в левом поддереве, иначе

- если X < M2, то поиск продолжается в среднем поддереве, иначе –
- в правом поддереве.

Если в корне находится только одна метка M, то переходим к левому поддереву при X < M и к правому поддереву – в противоположном случае. Продолжаем применять сформулированные выше правила, пока не окажемся на самом нижнем уровне дерева, где и выяснится найден ли элемент X, или же поиск потерпел неудачу.

Так как все листья 2–3 дерева находятся на одном и том же уровне, 2–3 дерево идеально сбалансировано с точки зрения глубины составляющих его поддеревьев. Все пути от корня до листа, которые мы проходим при поиске, имеют одну и ту же длину порядка $\log n$, где n – число элементов, хранящихся в дереве.

При добавлении нового элемента данных 2–3 дерево может расти не только в глубину, но и в ширину. Каждая внутренняя вершина, имеющая два поддерева, может приобрести новое поддерево, что приводит к росту вширь. Если же, с другой стороны, у вершины уже есть три поддерева, и она должна принять еще одно, то она расщепляется на две вершины, каждая из которых берет на себя по два из имеющихся четырех поддеревьев. Образовавшаяся при этом новая вершина передается вверх по дереву для присоединения к одной из выше расположенных вершин. Если же эта ситуация возникает на самом высоком уровне, то дерево вынуждено «вырасти» на один уровень вверх. Рис 10.3 иллюстрирует описанный принцип.

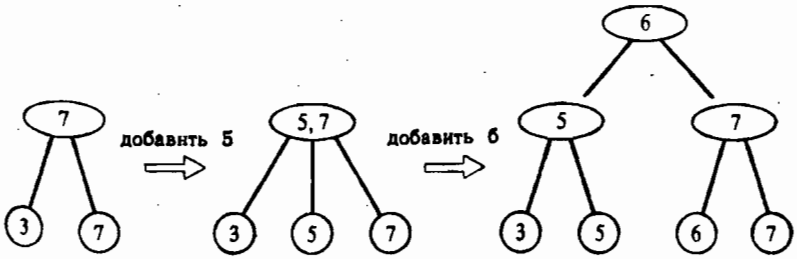


Рис. 10.3. Вставление нового элемента в 2–3 справочник. Дерево растет сначала вширь, а затем уже вглубь.

Включение нового элемента в 2-3 справочник мы запрограммируем как отношение

доб23(Дер, X, НовДер)

где дерево **НовДер** получено введением элемента **X** в дерево **Дер**. Основную работу мы поручим двум дополнительным отношениям, которые мы назовем **встав**. Первое из них имеет три аргумента:

встав(Дер, X, НовДер).

Здесь **НовДер** – результат вставления элемента **X** в **Дер**. Деревья **Дер** и **НовДер** имеют *одну и ту же глубину*. Разумеется, не всегда возможно сохранить ту же глубину дерева. Поэтому существует еще одно отношение с пятью аргументами специально для этого случая:

встав(Дер, X, НДа, Мб, НДб).

Имеется в виду, что при вставлении **X** в **Дер** дерево **Дер** разбивается на два дерева **НДа** и **НДб**, имеющих ту же глубину, что и **Дер**. **Мб** – это минимальный элемент из **НДб**. Пример показан на рис. 10.4.

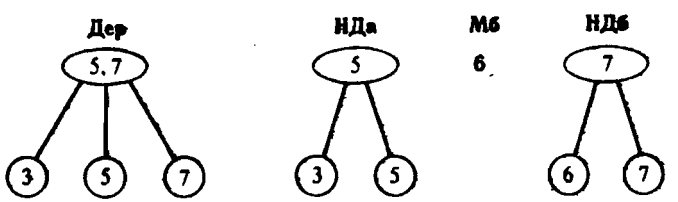


Рис. 10.4. Объекты, показанные на рисунке, удовлетворяют отношению **встав(Дер, 6, НДа, Мб, НДб)**.

2-3 деревья мы будем представлять в программе следующим образом:

- **nil** представлнет пустое дерево;
- **л(X)** представляет дерево, состоящее из одной вершины – листа с элементом X;
- **в2(Д1, М, Д2)** представляет дерево с двумя поддеревьями **Д1** и **Д2**; **М** – минимальный элемент из Д2;

- **в3(Д1, М2, Д2, М3, Д3)** представляет дерево с тремя поддеревьями Д1, Д2 и Д3; М2 – минимальный элемент из Д2; М3 – минимальный элемент из Д3; Д1, Д2 и Д3 – 2-3 деревья.

Между **доб23** и **встав** существует следующая связь: если после вставления нового элемента дерево не «вырастает», то

```
доб23( Дер, Х, НовДер) :-
    встав( Дер, Х, НовДер).
```

Однако если после вставления элемента глубина дерева увеличивается, то **встав** порождает два поддерева Д1 и Д2, а затем составляет из них дерево большей глубины:

```
доб23( Дер, Х, в2( Д1, М, Д2) ) :-
    встав( Дер, Х, Д1, М, Д2).
```

Отношение **встав** устроено более сложным образом, поскольку ему приходится иметь дело со многими случаями, а именно вставление в пустое дерево, в дерево, состоящее из одного листа, и в деревья типов в2 и в3. Возникают также дополнительные подслучаи, так как новый элемент можно вставить в первое, либо во второе, либо в третье поддерево. В связи с этим мы определим **встав** как набор правил таким образом, чтобы каждое предложение процедуры **встав** имело дело с одним из этих случаев. На рис. 10.5 показаны некоторые из возможных случаев. На Пролог они транслируются следующим образом:

Случай а

```
встав( в2( Д1, М, Д2), Х, в2( НД1, М, Д2) ) :-
    больше( М, Х),          % М больше, чем Х
    встав( Д1, Х, НД1).
```

Случай b

```
встав(в2(Д1,М,Д2),Х,в3(НД1а,Мб,НД1б,М,Д2) ) :-
    больше( М, Х),
    встав( Д1, Х, НД1а, Мб, НД1б).
```

Случай с

```
встав( в3( Д1, М2, Д2, М3, Д3), Х,
    в2( НД1а, Мб, НД1б), М2, в2(Д2, М3, Д3) ) :-
    больше( М2, Х),
    встав( Д1, Х, НД1а, Мб, НД1б).
```

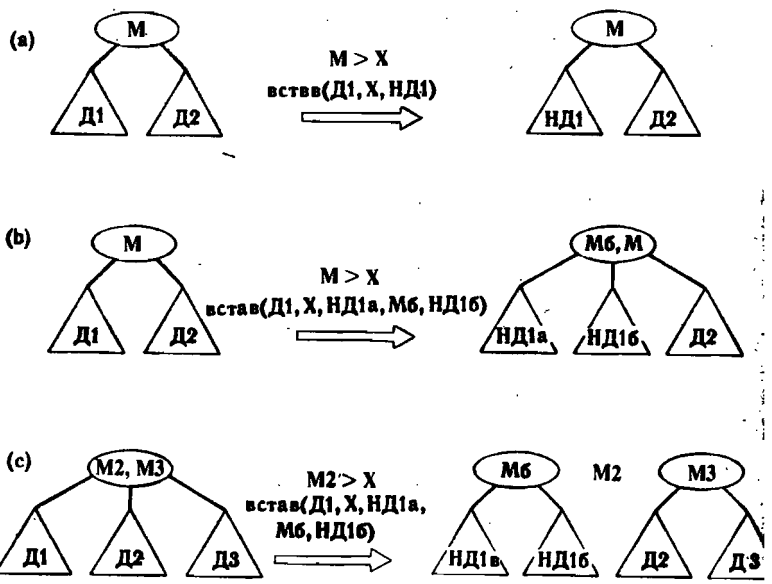


Рис. 10.5. Некоторые из случаев работы отношения **встав**.
(a) **встав(в2(Д1, М, Д2), Х, в2(НД1, М, Д2));**
(b) **встав(в2(Д1, М, Д2), Х,**
в3(НД1а, Мб, НД1б, М, Д2));
(c) **встав(в3(Д1, М2, Д2, М3, Д3), Х,**
в2(НД1а, Мб, НД1б), М2, в2(Д2, М3, Д3)).

```
% Вставление элемента в 2-3 справочник

доб23( Дер, Х, Дер1) :- % Вставить Х в Дер, получить Дер1
    встав( Дер, Х, Дер1).% Дерево растет вширь

доб23( Дер, Х, в2( Д1, М2, Д2) ) :-
    встав( Дер, Х, Д1, М2, Д2).    % Дерево растет вглубь

доб23( nil, Х, л(Х) ).

встав( л(А), Х, л(А), Х, л(Х) ) :-
    больше( Х, А).

встав( л(А), Х, л(Х), А, л(А) ) :-
    больше( А, Х).
```

```
встав( в2( Д1, М, Д2), Х, в2( НД1, М, Д2) ) :-
    больше( М, Х),
    встав( Д1, Х, НД1).

встав(в2(Д1,М,Д2),Х,в3(НД1а,Мб,НД1б,М,Д2) ) :-
    больше( М, Х),
    встав( Д1, Х, НД1а, Мб, НД1б).

встав( в2( Д1, М, Д2), Х, в2( Д1, М, НД2) ) :-
    больше( Х, М),
    встав( Д2, Х, НД2).

встав(в2(Д1,М,Д2),Х,в3(Д1,М,НД2а,Мб,НД2б) ) :-
    больше( Х, М),
    встав( Д2, Х, НД2а, Мб, НД2б).

встав(в3(Д1,М2,Д2,М3,Д3),Х,в3(НД1,М2,Д2,М3,Д3)) :-
    больше( М2, Х),
    встав( Д1, Х, НД1).

встав( в3( Д1, М2, Д2, М3, Д3), Х,
     в2( НД1а, Мб, НД1б), М2, в2( Д2, М3, Д3) ) :-
     больше( М2, Х),
     встав( Д1, Х, НД1а, Мб, НД1б).

встав( в3( Д1, М2, Д2, М3, Д3), Х,
    в3( Д1, М2, НД2, М3, Д3) ) :-
    больше( Х, М2), больше( М3, Х),
    встав( Д2, Х, НД2).

встав( в3( Д1, М2, Д2, М3, Д3), Х,
    в2( Д1, М2, НД2а), Мб, в2( НД2б, М3, Д3) ) :-
    больше( Х, М2), больше( М3, Х),
    встав( Д2, Х, НД2а, Мб, НД2б).

встав( в3( Д1, М2, Д2, М3, Д3), Х,
    в3( Д1, М2, Д2, М3, НД3) ) :-
    больше( Х, М3),
    встав( Д3, Х, НД3).

встав( в3( Д1, М2, Д2, М3, Д3), Х,
    в2( Д1, М2, Д2), М3, в2( НД3а, Мб, НД3б) ) :-
    больше( Х, М3),
    встав( Д3, Х, НД3а, Мб, НД3б).
```

Рис. 10.6. Вставление элемента в 2-3 справочник. В этой программе предусмотрено, что попытка повторного вставления элемента терпит неудачу.

Программа для вставления нового элемента в 2-3 справочник показана полностью на рис 10.6. На рис. 10.7 показана программа вывода на печать 2-3 деревьев.

Наша программа иногда выполняет лишние возвраты. Так, если встав с тремя аргументами терпит неудачу, то вызывается процедура встав с пятью аргументами, которая часть работы делает повторно. Можно устранить источник неэффективности, если, например, переопределить встав как

встав2(Дер, X, Деревья)

где Деревья - список, состоящий либо из одного, либо из трех аргументов:

Деревья= [НовДер], если встав(Дер, X, НовДер)
Деревья= [НДа, Мб, НДб],
если встав(Дер, X, НДа, Мб, НДб)

Теперь отношение **доб23** можно переопределить так:

```
доб23( Д, X, Д1) :-
    встав( Д, X, Деревья),
    соединить( Деревья, Д1).
```

Отношение соединить формирует одно дерево Д1 из деревьев, находящихся в списке Деревья.

Упражнения

10.1. Определите отношение

внутри(Элем, Дер)

для поиска элемента Элем в 2-3 справочнике Дер.

10.2. Введите в программу рис. 10.6 изменения для устранения лишних возвратов (определите отношения встав2 и соединить).

```
% Отображение 2-3 справочников
отобр(Д) :-
    отобр( Д, 0).

отобр( nil, _).

отобр( л(А), Н) :-
    tab( Н), write( А), nl.

отобр( в2( Д1, М, Д2), Н) :-
    Н1 is Н + 5
    отобр( Д2, Н1),
    tab( Н), write( --), nl,
    tab( Н), write( М), nl,
    tab( Н), write( --), nl,
    отобр( Д1, Н1).

отобр( в3( Д1, М2, Д2, М3, Д3), Н) :-
    Н1 is Н + 5
    отобр( Д3, Н1),
    tab( Н), write( --), nl,
    tab( Н), write( М3), nl,
    отобр( Д2, Н1),
    tab( Н), write( М2), nl,
    tab( Н), write( --), nl,
    отобр( Д1, Н1).
```

(a)

```
15
--
   15
   --
      13
   13
   --
      12
   --
   12
      10
   10
   --
      8
--
   8
--
      7
   --
   7
   --
      5
--
5
--
      4
   --
   4
      3
   3
   --
      1
```

(b)

Рис.10.7. (a) Программа для отображения 2-3 справочника. (b) Справочник (см. рис. 10.2), отпечатанный этой программой.

10.2. AVL–дерево: приближенно сбалансированное дерево

AVL–дерево – это дерево, обладающее следующими свойствами:

(1) Левое и правое поддеревья отличаются по глубине не более чем на 1.

(2) Оба поддерева являются AVL–деревьями.

Деревья, удовлетворяющие этому определению, могут быть слегка разбалансированными. Однако можно показать, что даже в худшем случае глубина AVL–дерева примерно пропорциональна log *n*, где *n* – число вершин дерева. Таким образом гарантируется логарифмический порядок производительности операций **внутри**, **добавить** и **удалить**.

Операции над AVL–деревом работают по существу так же, как и над двоичным справочником. В них только сделаны добавления, связанные с поддержанием приближенной сбалансированности дерева. Если после вставления или удаления дерево перестает быть приближенно сбалансированным, то специальные механизмы возвращают ему требуемую степень сбалансированности. Для того, чтобы эффективно реализовать этот механизм, нам придется сохранять некоторую дополнительную информацию относительно степени сбалансированности дерева. На самом деле, нам нужно знать только разность между глубинами поддеревьев, которая может принимать значения –1, 0 или +1. Тем не менее для простоты мы предпочтем сохранять сами величины глубин поддеревьев, а не разности между ними.

Мы определим отношение вставления элемента как

доб_avl(Дер, X, НовДер)

где оба дерева **Дер** и **НовДер** – это AVL–деревья, причем **НовДер** получено из **Дер** вставлением элемента **X**. AVL–деревья будем представлять как термы вида

д(Лев, A, Прав)/Глуб

где **A** – корень, **Лев** и **Прав** – поддеревья, а **Глуб** –

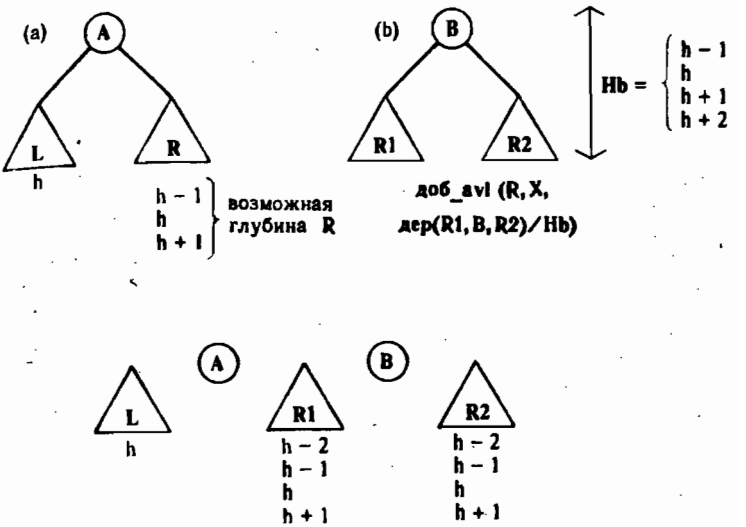


Рис.10.8. Задача вставления элемента в AVL–справочник
(a) AVL–дерево перед вставлением X, X > A;
(b) AVL–дерево после вставления X в R;
(c) составные части, из которых следует построить новое дерево.

глубина дерева. Пустое дерево изображается как **nil/0**. Теперь рассмотрим вставление элемента X в непустой AVL–справочник

Дер = д(L, A, R)/H

Начнем со случая, когда X больше A. X необходимо вставить в R, поэтому имеем следующее отношение:

доб_avl(R, X, д(R1, B, R2)/Hb)

На рис. 10.8 показаны составные части, из которых строится дерево **НовДер:**

L, A, R1, B, R2

Какова глубина деревьев L, R, R1 и R2? L и R могут отличаться по глубине не более, чем на 1. На рис. 10.8 видно, какую глубину могут иметь R1 и R2. Поскольку в R был добавлен только один элемент X, только одно из поддеревьев R1, R2 может иметь

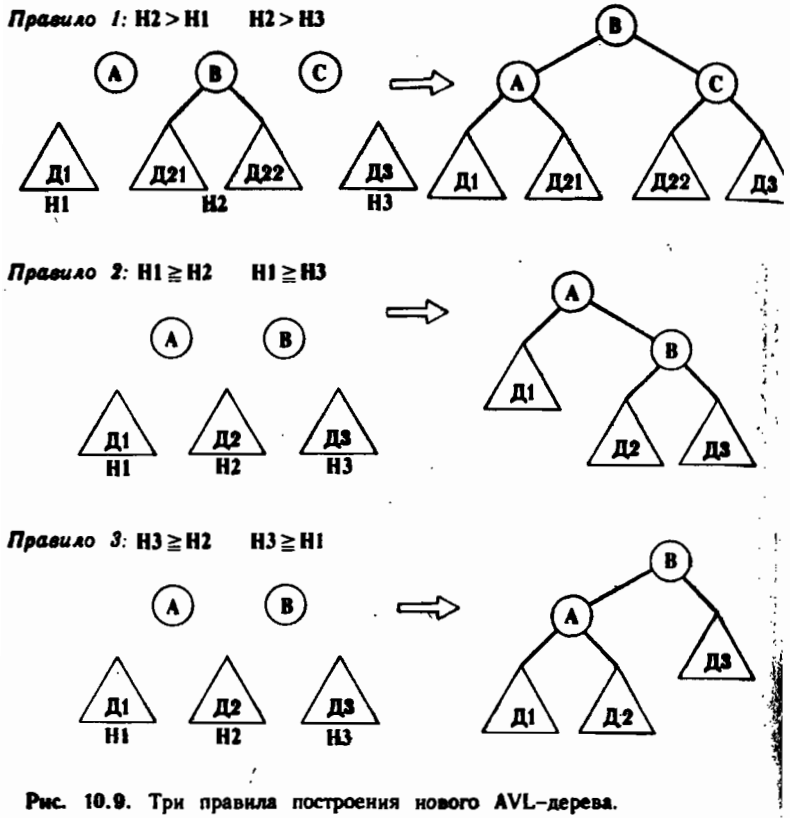


Рис. 10.9. Три правила построения нового AVL-дерева.

глубину h+1.

В случае, когда X меньше, чем A, имеем аналогичную ситуацию, причем левое и правое поддеревья меняются местами. Таким образом, в любом случае мы должны построить дерево **НовДер**, используя три дерева (назовем их **Дер1**, **Дер2** и **Дер3**) и два отдельных элемента A и B. Теперь рассмотрим вопрос: как соединить между собой эти пять составных частей, чтобы дерево **НовДер** было AVL-справочником? Ясно, что они должны располагаться внутри **НовДер** в следующем порядке (слева направо):

Дер1, A, Дер2, B, Дер3

Рассмотрим три случая:

(1) Среднее дерево **Дер2** глубже остальных двух деревьев.

(2) **Дер1** имеет глубину не меньше, чем **Дер2** и **Дер3**.

(3) **Дер3** имеет глубину не меньше, чем **Дер2** и **Дер1**.

На рис. 10.9 видно, как можно построить дерево **НовДер** в каждом из этих трех случаев. Например, в случае 1 среднее дерево **Дер2** следует разбить на две части, а затем включить их в состав **НовДер**. Три правила, показанные на рис.10.9, нетрудно записать на Прологе в виде отношения

соединить(Дер, А, Дер2, В, Дер3, НовДер)

Последний аргумент **НовДер** – это AVL-дерево, построенное из пяти составных частей, пяти первых аргументов. Правило 1, например, принимает вид:

```
соединить(
    Д1/Н1, А, д(Д21,В,Д22)/Н2, С, Д3/Н3,
                                  %Пять частей
  д(д(Д1/Н1,А,Д21)/Ha,B,д(Д22,C,Д3/Н3)/Hc)/Hb) :-
                                  % Результат
      Н2 > Н1, Н2 > Н3,  % Среднее дерево глубже остальных
      Ha is H1 + 1,      % Глубина левого поддерева
      Hc is H3 + 1,      % Глубина правого поддерева
      Hb is Ha + 1,      % Глубина всего дерева
```

Программа **доб_avl**, вычисляющая также глубину дерева и его поддеревьев, показана полностью на рис. 10.10.

Упражнение

10.3. Определите отношение

avl(Дер)

для проверки того, является ли Дер AVL-деревом, т.е. верно ли, что любые два его поддерева, подсоединенные к одной и той же вершине, отличаются по глубине не более чем на 1. Двоичные

```
% Вставление элемента в AVL-справочник

доб_avl( nil/0, X, д(nil/0, X, nil/0)/1).
                        % Добавить X к пустому дереву
доб_avl( д(L,Y,R)/Ну, X, НовДер) :-
                        % Добавить X к непустому дереву
    больше( Y, X),
    доб_avl( L, X, д(L1, Z, L2)/_),
                        % Добавить к левому поддереву
    соединить( L1, Z, L2, Y, R, НовДер).
                        % Сформировать новое дерево
доб_avl( д(L,Y,R)/Ну, X, НовДер) :-
    больше( X, Y),
    доб_avl( R, X, д(R1, Z, R2)/_),
                        % Добавить к правому поддереву
    соединить( L1, Y, R1, Z, R2, НовДер).

соединить( Д1/Н1, A, д(Д21, B, Д22)/Н2, C, Д3/Н3,
    д( д(Д1/Н1,A,Д21)/На,B,д(Д22,C,Д3/Н3)/Нс)/Нb) :
    Н2 > Н1, Н2 > Н3,    % Среднее дерево глубже остальных
    На is Н1 + 1,
    Нс is Н3 + 1,
    Нb is На + 1.

соединить( Д1/Н1, A, Д2/Н2, C, Д3/Н3,
    д( Д1/Н1, A, д(Д2/Н2,C,Д3/Н3)/Нс)/На) :-
    Н1 >= Н2, Н1 >= Н3,  % "Глубокое" левое дерево
    max1( Н2, Н3, Нс),
    max1( Н1, Нс, На).

соединить( Д1/Н1, A, Д2/Н2, C, Д3/Н3,
    д( д(Д1/Н1,A,Д2/Н2)/На, C, Д3/Н3)/Нс) :-
    Н3 >= Н2, Н3 >= Н1,  % "Глубокое" правое дерево
    max1( Н1, Н2, На),
    max1( На, Н3, Нс).

max1( U, V, M) :-
    U > V, !, M is U + 1;    % M равно 1 плюс max(U,V)
    M is V + 1.
```

Рис. 10.10. Вставление элемента в AVL-справочник. В этой программе предусмотрено, что попытка повторного вставления элемента терпит неудачу. По поводу процедуры **соединить** см. рис. 10.9.

деревья представляйте в виде термов
д(Лев, Кор, Прав) или **nil**.

Резюме

- 2–3 деревья и AVL–деревья, представленные в настоящей главе, – это примеры *сбалансированных* деревьев.
- Сбалансированные или приближенно сбалансированные деревья гарантируют эффективное выполнение трех основных операций над деревьями: поиск, добавление и удаление элемента. Время выполнения этих операций пропорционально $\log n$, где n – число вершин дерева.

Литература

2–3 деревья детально описаны, например, в Aho, Hopcroft and Ullman (1974, 1983). В книге этих авторов, вышедшей в 1983 г., дается также реализация соответствующих алгоритмов на языке Паскаль. Н.Вирт (см. Wirth (1976)) приводит программу на Паскале для работы с AVL–деревьями. 2–3 деревья являются частным случаем более общего поиятия B–деревьев. B–деревья, а также несколько других вариантов структур данных, имеющих отношение к 2–3 деревьям и AVL–деревьям, рассматриваются в книге Gonnet (1984). В этой книге, кроме того, даны результаты анализа поведения этих структур.

Программа вставления элемента в AVL–дерево, использующая только величиину «перекоса» дерева (т.е. значение разности глубин поддеревьев, равной -1, 0 или 1, вместо самой глубины) опубликована ван Эмденом (1981).

Aho A. V., Hopcroft J. E. and Ullman J. D. (1974). *The Design and Analysis of Computer Algorithms.* Addison-Wesley. [Имеется перевод: Ахо А., Хопкрофт Дж. Построение и анализ вычислительных алгоритмов. Пер. с англ. – М.: Мир, 1979.]

Aho A. V., Hopcroft J. E. and Ullman J. D. (1983). *Data Structures and Algorithms.* Addison-Wesley.

Gonnet G. H. (1984). *Handbook of Algorithms + Data Structures*. Addison-Wesley.

van Emden M. (1981). *Logic Programming Newsletter* 2.

Wirth N. (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall. [Имеется перевод: Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир, 1985.]

11. ОСНОВНЫЕ СТРАТЕГИИ РЕШЕНИЯ ЗАДАЧ

В данной главе мы сосредоточим свое внимание на одной общей схеме для представления задач, называемой *пространством состояний*. Пространство состояний – это граф, вершины которого соответствуют ситуациям, встречающимся в задаче («проблемные ситуации»), а решение задачи сводится к поиску пути в этом графе. Мы изучим на примерах, как формулируются задачи в терминах пространства состояний, а также обсудим общие методы решения задач, представленных в рамках этого формализма. Процесс решения задачи включает в себя поиск в графе, при этом, как правило, возникает проблема, как обрабатывать альтернативные пути поиска. В этой главе будут представлены две основные стратегии перебора альтернатив, а именно поиск в глубину и поиск в ширину.

11.1. Предварительные понятия и примеры

Рассмотрим пример, представленный на рис. 11.1. Задача состоит в выработке плана переупорядочивания кубиков, поставленных друг на друга, как показано на рисунке. На каждом шагу разрешается переставлять только один кубик. Кубик можно взять только тогда, когда его верхняя поверхность свободна. Кубик можно поставить либо на стол, либо на другой кубик. Для того, чтобы построить требуемый план, мы должны отыскать последовательность ходов, реализующую заданную трансформацию.

Эту задачу можно представлять себе как задачу

выбора среди множества возможных альтернатив. В исходной ситуации альтернатива всего одна: поставить кубик С на стол. После того как кубик С поставлен на стол, мы имеем три альтернативы:

- поставить А на стол или
- поставить А на С, или
- поставить С на А.

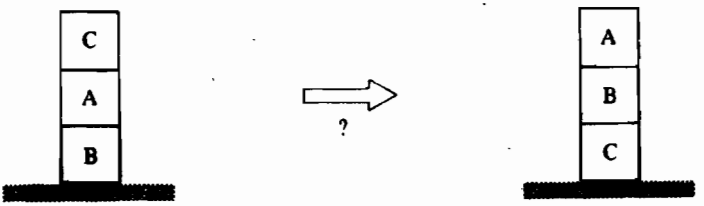


Рис. 11.1. Задача перестановки кубиков.

Ясно, что альтернативу «поставить С на стол» не имело смысла рассматривать всерьез, так как этот ход никак не влияет на ситуацию.

Как показывает рассмотренный пример, с задачами такого рода связано два типа понятий:

(1) Проблемные ситуации.
(2) Разрешенные ходы или действия, преобразующие одни проблемные ситуации в другие.

Проблемные ситуации вместе с возможными ходами образуют направленный граф, называемый *пространством состояний*. Пространство состояний для только что рассмотренного примера дано на рис. 11.2. Вершины графа соответствуют проблемным ситуациям, дуги – разрешенным переходам из одних состояний в другие. Задача отыскания плана решения задачи эквивалентна задаче построения пути между заданной начальной ситуацией («стартовой» вершиной) и некоторой указанной заранее конечной ситуацией, называемой также *целевой вершиной*.

На рис. 11.3 показан еще один пример задачи: головоломка «игра в восемь» и ее представление в виде задачи поиска пути. В головоломке используется восемь перемещаемых фишек, пронумерованных цифрами от 1 до 8. Фишки располагаются в девяти ячейках, образующих матрицу 3 на 3. Одна из ячеек

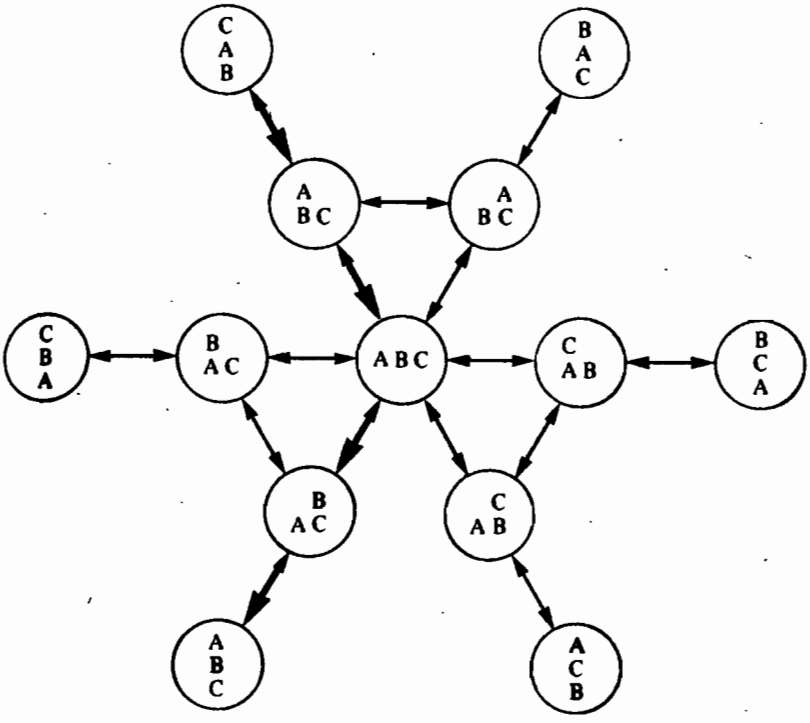


Рис. 11.2. Графическое представление задачи манипулирования кубиками. Выделенный путь является решением задачи рис. 11.1.

всегда пуста, и любая смежная с ней фишка может быть передвинута в эту пустую ячейку. Можно сказать и по-другому, что пустой ячейке разрешается перемещаться, меняясь местами с любой из смежных с ней фишек. Конечная ситуация – это некоторая заранее заданная конфигурация фишек, как показано на рис. 11.3.

Нетрудно построить аналогичное представление в виде графа и для других популярных головоломок. Наиболее очевидные примеры – это задача о «ханойской башне» и задача о перевозке через реку волка, козы и капусты. Во второй из этих задач предполагается, что вместе с человеком в лодке помещается только один объект и что человеку приходится охра-

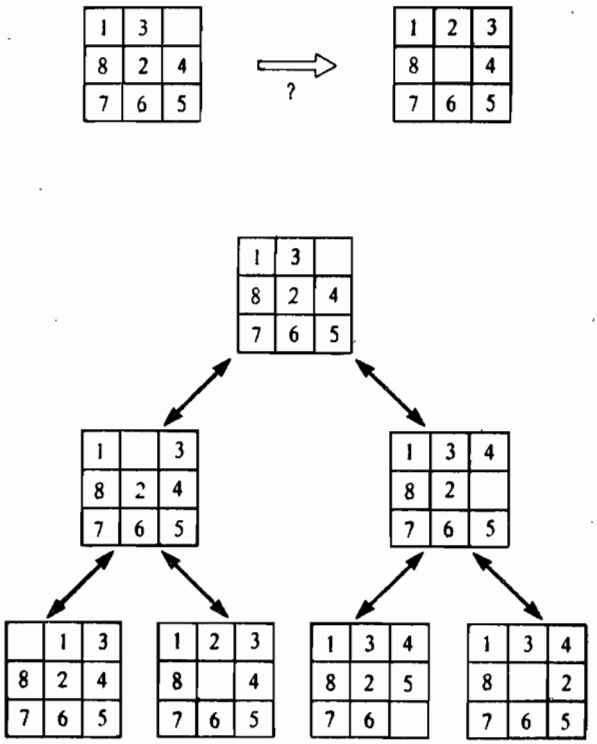


Рис. 11.3. "Игра в восемь" и ее представление в форме графа.

ИЯТЬ козу от волка и капусту от козы. С описанной парадигмой согласуются также многие задачи, имеющие практическое значение. Среди них – задача о коммивояжере, которая может служить моделью для многих практических оптимизационных задач. В задаче дается карта с n городами и указываются расстояния, которые надо преодолеть по дорогам при переезде из города в город. Необходимо найти маршрут, начинающийся в некотором городе, проходящий через все города и заканчивающийся в том же городе. Ни один город, за исключением начального, не разрешается посещать дважды.

Давайте подытожим те понятия, которые мы ввели, рассматривая примеры. Пространство состояний некоторой задачи определяет «правила игры»: вершины пространства состояний соответствуют ситуациям, а дуги – разрешениым ходам или действиям, или шагам решения задачи. Конкретная задача определяется

- пространством состояний
- стартовой вершиной
- целевым условием (т.е. условием, к достижению которого следует стремиться); «целевые вершины» – это вершины, удовлетворяющие этим условиям.

Каждому разрешенному ходу или действию можно приписать его стоимость. Например, в задаче манипуляции кубиками стоимости, приписаниые тем или иным перемещениям кубиков, будут указывать нам на то, что некоторые кубики перемещать труднее, чем другие. В задаче о коммивояжере ходы соответствуют переездам из города в город. Ясно, что в данном случае стоимость хода – это расстояние между соответствующими городами.

В тех случаях, когда каждый ход имеет стоимость, мы заинтересованы в отыскании решения минимальной стоимости. Стоимость решения – это сумма стоимостей дуг, из которых состоит «решающий путь» – путь из стартовой вершины в целевую. Даже если стоимости не заданы, все равно может возникнуть оптимизационная задача: нас может интересовать кратчайшее решение.

Прежде чем будут рассмотрены некоторые программы, реализующие классический алгоритм поиска в пространстве состояний, давайте сначала обсудим, как пространство состояний может быть представлено в прологовской программе.

Мы будем представлять пространство состояний при помощи отношения

после(X, Y)

которое истинно тогда, когда в пространстве состояний существует разрешенный ход из вершины X в вершину Y. Будем говорить, что Y – это *преемник* вершины X. Если с ходами связаны их стоимости, мы добавим третий аргумент, стоимость хода:

после(X, Y, Ст)

Эти отношения можно задавать в программе явным образом при помощи набора соответствующих фактов. Однако такой принцип оказывается непрактичным и нереальным для тех типичных случаев, когда пространство состояний устроено достаточно сложно. Поэтому отношение следования **после** обычно определяется неявно, при помощи правил вычисления вершин-преемников некоторой заданной вершины. Другим вопросом, представляющим интерес с самой общей точки зрения, является вопрос о способе представления состояний, т.е. самих вершин. Это представление должно быть компактным, но в то же время оно должно обеспечивать эффективное выполнение необходимых операций, в частности операции вычисления вершин-преемников, а возможно и стоимостей соответствующих ходов.

Рассмотрим в качестве примера задачу манипулирования кубиками, проиллюстрированную на рис. 11.1. Мы будем рассматривать более общий случай, когда имеется произвольное число кубиков, из которых составлены столбики, – один или несколько. Число столбиков мы ограничим некоторым максимальным числом, чтобы задача была интереснее. Такое ограничение, кроме того, является вполне реальным, поскольку рабочее пространство, которым располагает робот, манипулирующий кубиками, ограничено.

Проблемную ситуацию можно представить как список столбиков. Каждый столбик в свою очередь представляется списком кубиков, из которых он составлен. Кубики упорядочены в списке таким образом, что самый верхний кубик находится в голове списка. «Пустые» столбики изображаются как пустые списки. Таким образом, исходную ситуацию рис. 11.1 можно записать как терм

```
[ [c,a,b], [], [] ]
```

Целевая ситуация – это любая конфигурация кубиков, содержащая столбик, составленный из всех имеющихся кубиков в указанном порядке. Таких ситуаций три:

```
[ [a,b,c], [], [] ]
[ [], [a,b,c], [] ]
```

[[], [], [a,b,c]]

Отношение следования можно запрограммировать, исходя из следующего правила: ситуация **Сит2** есть преемник ситуации **Сит1**, если в **Сит1** имеется два столбика **Столб1** и **Столб2**, такие, что верхний кубик из **Столб1** можно поставить сверху на **Столб2** и получить тем самым **Сит2**. Поскольку все ситуации – это списки столбиков, правило транслируется на Пролог так:

```
после(Столбы,[Столб1,[Верх1|Столб2],Остальные]) :-
                        % Переставить Верх1 на Столб2
    удалить( [Верх1|Столб1], Столбы, Столбы1),
                        % Найти первый столбик
    удалить( Столб2, Столбы1, Остальные).
                        % Найти второй столбик

удалить( X, [X | L], L).

удалить( X, [Y | L], [Y | L1] ) :-
    удалить( L, X, L1).
```

В нашем примере целевое условие имеет вид:

```
цель( Ситуация) :-
    принадлежит( [a,b,c], Ситуация)
```

Алгоритм поиска мы запрограммируем как отношение

```
решить( Старт, Решение)
```

где **Старт** – стартовая вершина пространства состояний, а **Решение** – путь, ведущий из вершины **Старт** в любую целевую вершину. Для нашего конкретного примера обращение к пролог-системе имеет вид:

```
?- решить( [ [c,a,b], [], [] ], Решение).
```

В результате успешного поиска переменная **Решение** конкретизируется и превращается в список конфигураций кубиков. Этот список представляет собой план преобразования исходного состояния в состояние, в котором все три кубика поставлены друг на друга в указанном порядке [a,b,c].

11.2. Стратегия поиска в глубину

Существует много различных подходов к проблеме поиска решающего пути для задач, сформулированных в терминах пространства состояний. Основные две стратегии поиска – это поиск *в глубину* и поиск *в ширину*. В настоящем разделе мы реализуем первую из них.

Мы начнем разработку алгоритма и его вариантов со следующей простой идеи:

Для того, чтобы найти решающий путь **Реш** из заданной вершины **В** в некоторую целевую вершину, необходимо:

- если **В** – это целевая вершина, то положить **Реш = [В]**, или
- если для исходной вершины **В** существует вершина-преемник **В1**, такая, что можно провести путь **Реш1** из **В1** в целевую вершину, то положить **Реш = [В | Реш1]**.

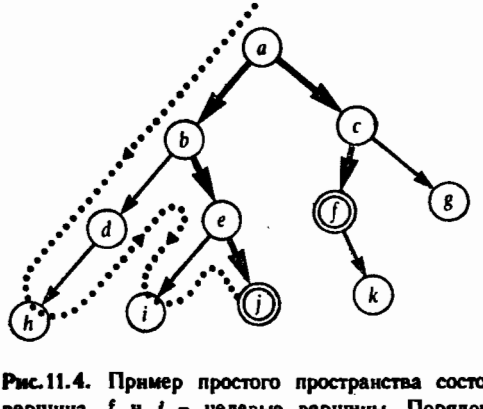


Рис. 11.4. Пример простого пространства состояний: *a* – стартовая вершина, *f* и *j* – целевые вершины. Порядок, в котором происходит проход по вершинам пространства состояний при поиске в глубину. *a, b, d, h, e, i, j*. Найдено решение **[a, b, e, j]**. После возврата обнаружено другое решение: **[a, c, f]**.

На Пролог это правило транслируется так:

```
решить( В, [В] ) :-
    цель( В).
решить( В, [В | Реш1] ) :-
    после( В, В1 ),
    решить( В1, Реш1).
```

Эта программа и есть реализация поиска в глубину. Мы говорим «в глубину», имея в виду тот порядок, в котором рассматриваются альтернативы в пространстве состояний. Всегда, когда алгоритму поиска в глубину надлежит выбрать из нескольких вершин ту, в которую следует перейти для продолжения поиска, он предпочитает самую «глубокую» из них. Самая глубокая вершина – это вершина, расположенная дальше других от стартовой вершины. На рис. 11.4 мы видим на примере, в каком порядке алгоритм проходит по вершинам. Этот порядок в точности соответствует результату трассировки процесса вычислений в пролог-системе при ответе на вопрос

```
?- решить( а, Реш).
```

Поиск в глубину наиболее адекватен рекурсивному стилю программирования, принятому в Прологе. Причина этого состоит в том, что, обрабатывая цели, пролог-система сама просматривает альтернативы именно в глубину.

Поиск в глубину прост, его легко программировать и он в некоторых случаях хорошо работает. Программа для решения задачи о восьми ферзях (см. гл. 4) фактически была примером поиска в глубину. Для того, чтобы можно было применить к этой задаче описанную выше процедуру **решить**, необходимо сформулировать задачу в терминах пространства состояний. Это можно сделать так:

- вершины пространства состояний – позиции, в которых поставлено 0 или более ферзей на нескольких последовательно расположенных горизонтальных линиях доски;
- вершина-преемник данной вершины может быть получена из нее после того, как в соответствующей позиции на следующую горизонтальную линию доски будет поставлен еще один ферзь, причем таким образом, чтобы ни один из уже поставленных ферзей не оказался под боем;

- стартовая вершина – пустая доска (представляется пустым списком);
- целевая вершина – любая позицня с восемью ферзями (правило получения вершины-преемника гарантнрует, что ферзи не бьют друг друга).

Позицию на доске будем представлять как спнсок Y-координат поставленных ферзей. Получаем программу:

```
после( Ферзи, [Ферзь | Ферзи] ) :-
    принадлежит( Ферзь, [1,2,3,4,5,6,7,8] ),
                % Поместить ферзя на любую вертикальную лннню
    небьет( Ферзь, Ферзи).

цель( [_, _, _, _, _, _, _, _] )
                % Позицня с восемью ферзямн
```

Отношение небьет означает, что **Ферзь** не может поразнть ни одного ферзя из спнска **Ферзи**. Эту процедуру можно легко запрограммировать так же, как это сделано в гл. 4. Ответ на вопрос

```
?- решить( [], Решение)
```

будет выглядеть как список позиций с постепенно увеличивающимся количеством поставленных ферзей. Список завершается «безопасной» конфигурацией из восьми ферзей. Механизм возвратов позволит получить и другие решения задачи.

Поиск в глубину часто работает хорошо, как в рассмотренном примере, однако наша простая процедура **решить** может попасть в затруднительное положение, причем многими способами. Случится ли это или нет – зависит от структуры пространства состояний. Для того, чтобы затруднить работу процедуры **решить** в примере рис. 11.4, достаточно внести в задачу совсем небольшое изменение: добавить дугу, ведущую из h в d, чтобы получился цикл (рис. 11.5). В этом случае поиск будет выглядеть так: начиная с вершины a, спускаемся вплоть до h, придерживаясь самой левой ветви графа. На этот раз, в отличие от рис. 11.4, у вершины h будет преемник d. Поэтому произойдет *не возврат* из h, а *переход* к d. Затем мы найдем преемника вершины d, т.е. вершину h, и т.д., в результате программа зациклится между h и d.

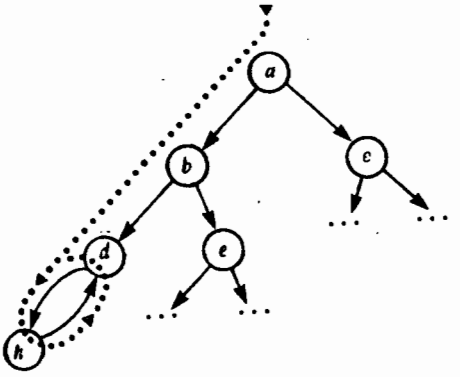


Рис. 11.5. Начинаясь в *a*, поиск вглубину заканчивается бесконечным циклом между *d* и *h*: *a*, *b*, *d*, *h*, *d*, *h*, *d* ....

Очевидное усовершенствование нашей программы поиска в глубину – добавление к ней механизма обнаружения циклов. Ни одну из вершин, уже содержащихся в пути, построенном из стартовой вершины в текущую вершину, не следует вторично рассматривать в качестве возможной альтернативы продолжения поиска. Это правило можно сформулировать в виде отношения

вглубину(Путь, Верш, Решение)

Как видно из рис. 11.6, **Верш** – это состояние, из которого необходимо найти путь до цели; **Путь** – путь (список вершин) между стартовой вершиной и **Верш**; **Решение** – **Путь**, продолженный до целевой вершины.

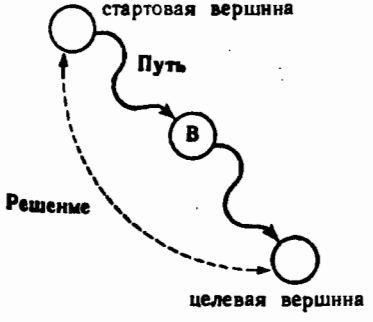


Рис. 11.6. Отношение **вглубину(Путь, В, Решение)**.

Для облегчения программирования вершины в списках, представляющих пути, будут расставляться в обратном порядке. Аргумент **Путь** нужен для того,

(1) чтобы не рассматривать тех преемников вершины **Верш**, которые уже встречались раньше (обнаружение циклов);

(2) чтобы облегчить построение решающего пути **Решение**.

Соответствующая программа поиска в глубину показана на рис. 11.7.

```
решить( Верш, Решение) :-
   вглубину( [], Верш, Решение).

вглубину( Путь, Верш, [Верш | Путь] ) :-
   цель( Верш).

вглубину( Путь, Верш, Реш) :-
   после( Верш, Верш1),
   not принадлежит( Верш1, Путь),          % Цикл?
   вглубину( [Верш | Путь], Верш1, Реш).
```

Рис. 11.7. Программа поиска в глубину без зацикливания.

Теперь наметим один вариант этой программы. Аргументы **Путь** и **Верш** процедуры **вглубину** можно объединить в один список **[Верш | Путь]**. Тогда, вместо вершины-кандидата **Верш**, претендующей на то, что она находится на пути, ведущем к цели, мы будем иметь *путь*-кандидат **П** = **[Верш | Путь]**, который претендует на то, что его можно продолжить вплоть до целевой вершины. Программирование соответствующего предиката

вглубину(П, Решение)

оставим читателю в качестве упражнения.

Наша процедура поиска в глубину, снабженная механизмом обнаружения циклов, будет успешно находить решающие пути в пространствах состояний, подобных показанному на рис. 11.5. Существуют, однако, такие пространства состояний, в которых наша процедура не дойдет до цели. Дело в том, что многие пространства состояний бесконечны. В таком

пространстве алгоритм поиска в глубину может «потерять» цель, двигаясь вдоль бесконечной ветви графа. Программа будет бесконечно долго обследовать эту бесконечную область пространства, так и не приблизившись к цели. Пространство состояний задачи о восьми ферзях, определенное так, как это сделано в настоящем разделе, на первый взгляд содержит ловушку именно такого рода. Но оказывается, что оно все-таки конечно, поскольку Y-координаты выбираются из ограниченного множества, и поэтому на доску можно поставить «безопасным образом» не более восьми ферзей.

```
вглубину2( Верш, [Верш], _) :-
    цель( Верш).

вглубину2( Верш, [Верш | Реш], МаксГлуб) :-
    МаксГлуб > 0,
    после( Верш, Верш1),
    Макс1 is МаксГлуб - 1,
    вглубину2( Верш1, Реш, Макс1).
```

Рис. 11.8. Программа поиска в глубину с ограничением по глубине.

Для того, чтобы предотвратить бесцельное блуждание по бесконечным ветвям, мы можем добавить в базовую процедуру поиска в глубину еще одно усовершенствование, а именно, ввести ограничение на глубину поиска. Процедура поиска в глубину будет тогда иметь следующие аргументы:

вглубину2(Верш, Решение, МаксГлуб)

Не разрешается вести поиск на глубине большей, чем **МаксГлуб**. Программная реализация этого ограничения сводится к уменьшению на единицу величины предела глубины при каждом рекурсивном обращении к **вглубину2** и к проверке, что этот предел не стал отрицательным. В результате получаем программу, показанную на рис. 11.8.

Упражнения

11.1. Напишите процедуру поиска в глубину (с обнаружением циклов)

вглубину1(ПутьКандидат, Решение)

отыскивающую решающий путь **Решение** как продолжение пути **ПутьКандидат**. Оба пути представляйте списками вершин, расположенных в обратном порядке так, что целевая вершина окажется в голове списка **Решение**.

11.2. Напишите процедуру поиска в глубину, сочетающую в себе обнаружение циклов с ограничением глубины, используя рис. 11.7 и 11.8.

11.3. Проведите эксперимент по применению программы поиска в глубину к задаче планирования в «мире кубиков» (рис. 11.1).

11.4. Напишите процедуру

отобр(Ситуация)

для отображения состояния задачи «перестановки кубиков». Пусть **Ситуация** – это список столбиков, а столбик, в свою очередь, – список кубиков. Цель

отобр([[a], [e,d], [c,b]])

должна отпечатать соответствующую ситуацию, например так:

```
      e     c
a     d     b
===============
```

11.3. Поиск в ширину

В противоположность поиску в глубину стратегия поиска в ширину предусматривает переход в первую очередь к вершинам, ближайшим к стартовой вершине. В результате процесс поиска имеет тенденцию развиваться более в ширину, чем в глубину, что иллюстрирует рис. 11.9.

Поиск в ширину программируется не так легко, как поиск в глубину. Причина состоит в том, что

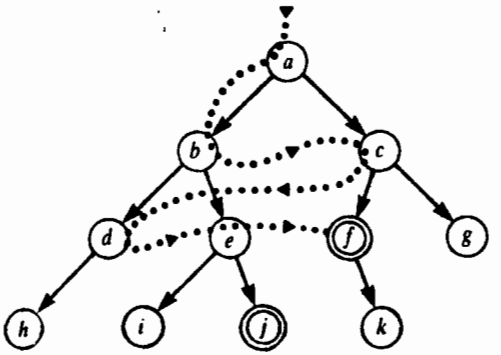


Рис. 11.9. Простое пространство состояний: a – стартовая вершина, f и j – целевые вершины. Применение стратегии стратегии поиска в ширину дает следующий порядок прохода по вершинам: a, b, c, d, e, f. Более короткое решение [a, c, f] найдено раньше, чем более длинное [a, b, e, j].

нам приходится сохранять все множество альтернативных вершин-кандидатов, а не только одиу вершину, как при поиске в глубину. Более того, если мы желаем получить при помощи процесса поиска решающий путь, то одиого множества вершин не достаточно. Поэтому мы будем хранить не множество вершин-кандидатов, а множество *путей*-кандидатов. Таким образом, цель

вширину(Пути, Решения)

истинна только тогда, когда существует путь из множества кандидатов **Пути**, который может быть продолжен вплоть до целевой вершины. Этот продолженный путь и есть **Решение**.

11.3.1. Списковое представление множества кандидатов

В нашей первой реализации этой идеи мы будем использовать следующее представление для миожества

```
решить( Старт, Решение) :-
    вширину( [ [Старт] ], Решение).

вширину( [ [Верш | Путь] | _], [Верш | Путь] ) :-
    цель( Верш).

вширину( [ [В | Путь] | Пути], Решение ) :-
    bagof( [В1, В | Путь ],
        (после(В,В1), not принадлежит(В1,[В|Путь])),
        НовПути),
            %НовПути - ациклические продолжения пути [В|Путь]
    конк( Пути, НовПути, Пути1), !,
    вширину( Пути1, Решение);
    вширину( Пути, Решение).
            %Случай, когда у В нет преемнпка
```

Рис. 11.10. Реализация поиска в ширину.

путей-кандидатов. Само множество будет списком путей, а каждый путь – списком вершин, перечисленных в обратном порядке, т. е. головой списка будет самая последняя из порожденных вершин, а последним элементом списка будет стартовая вершина. Поиск начинается с одноэлементного множества кандидатов

```
[ [СтартВерш] ]
```

Общие принципы поиска в ширину таковы:

Для того, чтобы выполнить поиск в ширину при заданном множестве путей-кандидатов, нужно:

- если голова первого пути – это целевая вершина, то взять этот путь в качестве решения, иначе
- удалить первый путь из множества кандидатов и породить множество всех возможных продолжений этого пути на один шаг; множество продолжений добавить в конец множества кандидатов, а затем выполнить поиск в ширину с полученным новым множеством.

В случае примера рис. 11.9 этот процесс будет развиваться следующим образом:

```
решить( Старт, Решение) :-
    вширь( [ [Старт] | Z ]-Z, Решение).

вширь( [ [Верш | Путь] | _]-_, [Верш | Путь] ) :-
    цель( Верш).

вширь( [ [B | Путь] | Пути]-Z, Решение ) :-
    bagof( [B1, B | Путь ],
           ( после(B,B1),
             not принадлежит(B1,[B|Путь]) ),
           Нов ),
    конк( Нов, ZZ, Z), !,
    вширь( Пути-ZZ, Решение);
    Пути \== Z,    % Множество кандидатов не пусто
    вширь( Пути-Z, Решение).
```

Рис. 11.11. Программа поиска в ширину более эффективная, чем программа рис. 11.10. Усовершенствование основано на разностном представлении списка путей-кандидатов.

(1) Начинаем с начального множества кандадатов:

```
[ [a] ]
```

(2) Порождаем продолжения пути [a]:

```
[ [b,a], [c,a] ]
```

(Обратите внимание, что пути записаны в обратном порядке.)

(3) Удаляем первый путь из множества кандидатов и порождаем его продолжения:

```
[ [d,b,a], [e,b,a] ]
```

Добавляем список продолжений в конец списка кандидатов:

```
[ [c,a], [d,b,a], [e,b,a] ]
```

(4) Удаляем [c,a], а затем добавляем все его продолжения в конец множества кандидатов. Получаем:

```
[ [d,b,a], [e,b,a], [f,c,a], [g,c,a] ]
```

Далее, после того, как пути [d,b,a] и [e,b,a] будут продолжены, измененный список кандидатов примет вид

```
[[f,c,a],[g,c,a],[h,d,b,a],[i,e,b,a],[j,e,b,a]]
```

В этот момент обнаруживается путь [f,c,a], содержащий целевую вершину f. Этот путь выдается в качестве решения.

Программа, порождающая этот процесс, показана на рис. 11.10. В этой программе все продолжения пути на один шаг генерируются встроенной процедурой bagof. Кроме того, делается проверка, предотвращающая порождение циклических путей. Обратите внимание на то, что в случае, когда путь продолжить невозможно, и цель bagof терпит неудачу, обеспечивается альтернативный запуск процедуры **вширину**. Процедуры **принадлежит** и **конк** реализуют отношения принадлежности списку и конкатенации списков соответственно.

Недостатком этой программы является неэффективность операции **конк**. Положение можно исправить, применив разностное представление списков (см. гл. 8). Тогда множество путей-кандидатов будет представлено парой списков **Пути** и **Z**, записанной в виде

Пути–Z

При введении этого представления в программу рис. 11.10 ее можно постепенно преобразовать в программу, показанную на рис. 11.11. Оставим это преобразование читателю в качестве упражнения.

11.3.2. Древовидное представление множества кандидатов

Рассмотрим теперь еще одно изменение нашей программы поиска в ширину. До сих пор мы представляли множества путей-кандидатов как списки путей. Это расточительный способ, поскольку начальные участки путей являются общими для нескольких из них. Таким образом, эти общие части путей приходится хранить во многих экземплярах. Избежать избыточности помогло бы более компактное представление множества кандидатов. Таким более компактным представлением является дерево, в котором общие участки путей хранятся в его верхней части без дублирования. Будем использовать в программе следующее представле-

ние дерева. Имеется два случая:

Случай 1: Дерево состоит только из одной вершины В; В этом случае оно имеет вид терма **л(В)**; Функтор л указывает на то, что В – это лист дерева.

Случай 2: Дерево состоит из корневой вершины В и множества поддеревьев Д1, Д2, ... Такое дерево представляется термом

д(В, Пд)

где **Пд** – список поддеревьев:

Пд = [Д1, Д2, ...]

В качестве примера рассмотрим ситуацию, которая возникает после того, как порождены три уровня дерева рис. 11.9. Множество путей-кандидатов в случае спискового представления имеет вид:

[[d,b,a], [e,b,a], [f,c,a], [g,c,a]]

В виде дерева это множество выглядит так:

д(a,[д(b,[л(d),л(e)]),д(c,[л(f),л(g)])])

На первый взгляд древовидное представление кажется еще более расточительным, чем списковое, однако это всего лишь поверхностное впечатление, связанное с компактностью прологовской нотации для списков.

В случае спискового представления множества кандидатов эффект распространения процесса в ширину достигался за счет перемещения продолженных путей в конец списка. В нашем случае мы уже не можем использовать этот прием, поэтому программа несколько усложняется. Ключевую роль в нашей программе будет играть отношение

расширить(Путь, Дер, Дер1, ЕстьРеш, Решение)

На рис. 11.12 показано, как связаны между собой аргументы отношения **расширить**. При каждом обращении к **расширить** переменные **Путь** и **Дер** будут уже конкретизированы. **Дер** – поддерево всего дерева поиска, одновременно оно служит для представления множества путей-кандидатов внутри этого поддерева. **Путь** – это путь, ведущий из стартовой вершины в корень поддерева **Дер**. Самая общая идея алгоритма –

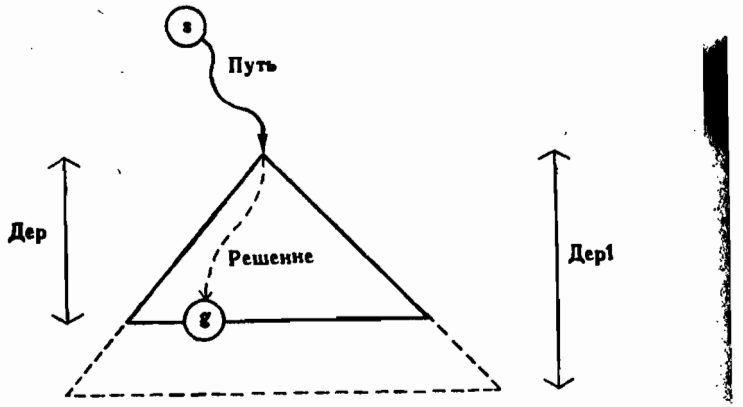


Рис. 11.12. Отношение **расширить(Путь, Дер, Дер1, ЕстьРеш, Решение)**: **s** – стартовая вершина, **g** – целевая вершина. **Решение** – это **Путь**, продолженный вплоть до **g**. **Дер1** – результат расширения дерева **Дер** на один уровень вниз.

получить поддерево Дер1 как результат расширения Дер на один уровень. Но в случае, когда в процессе расширения поддерева Дер встретится целевая вершина, процедура **расширить** должна сформировать соответствующий решающий путь.

Итак, процедура **расширить** будет порождать два типа результатов. На конкретный вид результата будет указывать значение переменной **ЕстьРеш**:

(1) **ЕстьРеш** = да
Решение = решающий путь, т. е. **Путь**, продолженный до целевой вершины.
Дер1 = неконкретизировано.

Разумеется, такой тип результата получится только в том случае, когда Дер будет содержать целевую вершину. Добавим также, что эта целевая вершина обязана быть листом поддерева Дер.

(2) **ЕстьРеш** = нет
Дер1 = результат расширения поддерева Дер на один уровень вниз от своего «подножья». Дер1 не содержит ни одной «тупиковой» ветви из Дер, т. е. такой

ветви, что она либо не может быть продолжена из-за отсутствия преемников, либо любое ее продолжение приводит к циклу.

Решение = неконкретизировано.

Если в дереве **Дер** нет ни одной целевой вершины и, кроме того, оно не может быть расширено, то процедура **расширить** терпит неудачу.

Процедура верхнего уровня для поиска в ширину

вширину(Дер, Решение)

отыскивает **Решение** либо среди множества кандидатов **Дер**, либо в его расширении. На рис. 11.3 показано, как выглядит программа целиком. В этой программе имеется вспомогательная процедура **расширитьвсе**. Она расширяет все деревья из некоторого *списка*, а затем, выбросив все «тупиковые» деревья», собирает все полученные расширенные деревья в один новый список. Используя механизм возвратов, она также порождает все решения, обнаруженные в деревьях из списка. Имеется одна дополнительная деталь: по крайней мере одно из деревьев должно «вырасти». Если это не так, то процедуре **расширитьвсе** не удается получить ни одного расширенного дерева – все деревья из списка оказываются «тупиковыми».

```
% ПОИСК В ШИРИНУ
% Множество кандидатов представлено деревом

решить( Старт, Решение) :-
    вширину( л(Старт), Решение).

вширину( Дер, Решение) :-
    расширить( [], Дер, Дер1, ЕстьРеш, Решение),
    (ЕстьРеш = да;
    ЕстьРеш = нет, вширину( Дер1, Решение) ).

расширить( П, Л(В), _, да, [В | П] ) :-
    цель( В).

расширить( П, Л( В), д( В, Пд), нет, _) :-
    bagof(л(В1),
          (после(В,В1),not принадлежит(В1,П)),Пд).

расширить( П, д(В,Пд), д(В,Пд1), ЕстьРеш, Реш) :-
    расширитьвсе([В | П],Пд,[],Пд1,ЕстьРеш,Реш).
```

```
расширитьвсе( _, [], [Д | ДД], [Д | ДД], нет, _).
                % По крайней мере одно дерево должно вырасти
расширитьвсе(П,[Д | ДД],ДД1,Пд1,ЕстьРеш,Реш) :-
    расширить( П, Д, Д1, ЕстьРеш1, Реш),
    ( ЕстьРеш1 = да, ЕстьРеш = да;
      ЕстьРеш1 = нет, !,
      расширитьвсе(П,ДД,[Д1|ДД1],Пд1,ЕстьРеш,Реш));
    расширитьвсе(П,ДД,ДД1,Пд1,ЕстьРеш,Реш ).
```

Рис. 11.13. Реализация поиска в ширину с использованием древовидного представления множества путей-кандидатов.

Мы разработали эту более сложную реализацию поиска в ширину не только для того, чтобы получить программу более экономичную по сравнению с предыдущей версией, но также и потому, что такое решение задачи может послужить хорошим стартом для перехода к усложненным программам поиска, управляемым эвристиками, таким как программа поиска с предпочтением из гл. 12.

Упражнения

11.5. Перепишите программу поиска в ширину рис. 11.10, используя разностное представление для списка путей-кандидатов и покажите, что в результате получится программа, приведенная на рис. 11.11. Зачем в программу рис. 11.11 включена цель

```
Пути \== Z
```

Проверьте, что случится при поиске в пространстве состояний рис. 11.9, если эту цель опустить. Различие в выполнении программы возникнет только при попытке найти новые решения в ситуации, когда не осталось больше ни одного решения.

11.6. Как программы настоящего раздела можно использовать для поиска, начинающегося от *стартового множества* вершин, вместо одной стартовой вершины?

11.7. Как программы этой главы можно использовать для поиска в обратном направленнн, т.е. от целевой вершины к стартовой вершине (или к одной из стартовых вершин, если их несколько). Указание: переопределите отношение после. В каких ситуациях обратный поиск будет иметь преимущества перед прямым поиском?

11.8. Иногда выгодно сделать поиск *двунаправленным*, т. е. продвигаться одновременно с двух сторон от стартовой и целевой вершин. Поиск заканчивается, когда оба пути «встречаются». Определите пространство поиска (отношение **после**) и целевое отношение для заданного графа таким образом, чтобы наши процедуры поиска в действительности выполняли двунаправленный поиск.

11.9. Проведите эксперименты с различными методами поиска применительно к задаче планирования в «мире кубиков».

11.4. Замечания относительно поиска в графах, оптимальности и сложности

Сейчас уместно сделать ряд замечаний относительно программ поиска, разработанных к настоящему моменту: во-первых, о поиске в графах, во-вторых, об оптимальности полученных решений и, в-третьих, о сложности поиска.

Приведенные примеры могли создать ложное впечатление, что наши программы поиска в ширину способны работать только в пространствах состояний, являющихся деревьями, а не графами общего вида. На самом деле, тот факт, что в одной из версий множество путей-кандидатов представлялось деревом, совсем не означает, что и само пространство состояний должно было быть деревом. Когда поиск проводится в графе, граф фактически разворачивается в дерево, причем некоторые пути, возможно, дублируются в разных частях этого дерева (см. рис. 11.14).

Наши программы поиска в ширину порождают решающие пути один за другим в порядке увеличения их

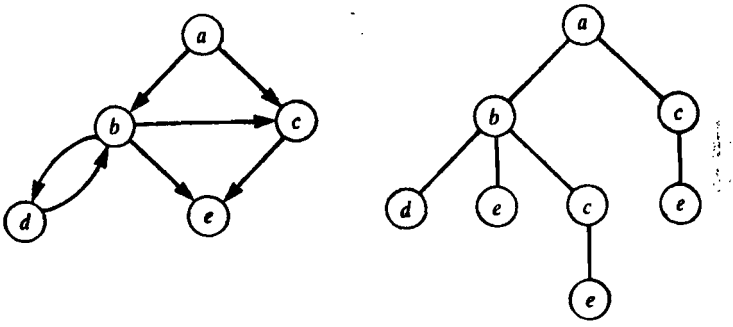


Рис. 11.14. (a) Пространство состояний; a – стартовая вершина. (b) Дерево всех возможных ациклических путей, ведущих из a, порожденное программой поиска в ширину.

длин – самые короткие решения идут первыми. Это является важным обстоятельством, если нам необходима оптимальность (в отношении длины решения). Стратегия поиска в ширину гарантирует получение кратчайшего решения первым. Разумеется, это неверно для поиска в глубину.

Наши программы, однако, не учитывают стоимости, приписанные дугам в пространстве состояний. Если критерием оптимальности является минимум стоимости решающего пути (а не его длина), то в этом случае поиска в ширину недостаточно. Поиск с предпочтением из гл. 12 будет направлен на оптимизацию стоимости.

Еще одна типичная проблема, связанная с задачей поиска, – это проблема *комбинаторной сложности*. Для нетривиальных предметных областей число альтернатив столь велико, что проблема сложности часто принимает критический характер. Легко понять, почему это происходит: если каждая вершина имеет b преемников, то число путей длины l, ведущих из стартовой вершины, равно b^l (в предположении, что циклов нет). Таким образом, вместе с увеличением длин путей наблюдается *экспоненциальный* рост объема множества путей-кандидатов, что приводит к ситуации, называемой *комбинаторным взрывом*. Стратегии поиска в глубину и ширину недостаточно «умны» для борьбы с такой степенью комбинаторной сложнос-

ти: отсутствие селективности приводит к тому, что все пути рассматриваются как одинаково перспективные.

По-видимому, более изощренные процедуры поиска должны использовать какую-либо информацию, отражающую специфику данной задачи, с тем чтобы на каждой стадии поиска принимать решения о наиболее перспективных путях поиска. В результате процесс будет продвигаться к целевой вершине, обходя бесполезные пути. Информация, относящаяся к конкретной решаемой задаче и используемая для управления поиском, называется *эвристикой*. Про алгоритмы, использующие эвристики, говорят, что они *руководствуются эвристиками*: они выполняют *эвристический поиск*. Один из таких методов изложен в следующей главе.

Резюме

- *Пространство состояний* есть формализм для представления задач.
- Пространство состояний – это направленный граф, вершины которого соответствуют проблемным ситуациям, а дуги – возможным ходам. Конкретная задача определяется *стартовой вершиной* и *целевым условием*. Решению задачи соответствует путь в графе. Таким образом, решение задачи сводится к поиску пути в графе.
- Оптимизационные задачи моделируются приписыванием каждой дуге пространства состояний некоторой стоимости.
- Имеются две основных стратегии поиска в пространстве состояний – *поиск в глубину* и *поиск в ширину*.
- Поиск в глубину программируется наиболее легко, однако подвержен зацикливаниям. Существуют два простых метода предотвращения зацикливания: ограничить глубину поиска и не допускать дублирования вершин.
- Реализация поиска в ширину более сложна, поскольку требуется сохранять множество кан-

дидатов. Это множество может быть с легкостью представлено списком списков, но более экономное представление – в виде дерева.

- Поиск в ширину всегда первым обнаруживает самое короткое решение, что не верно в отношении стратегии поиска в глубину.
- В случае обширных пространств состояний существует опасность *комбинаторного взрыва*. Обе стратегии плохо приспособлены для борьбы с этой трудностью. В таких случаях необходимо руководствоваться эвристиками.
- В этой главе были введены следующие понятия:

  пространство состояний
  стартовая вершина, целевое условие,
  решающий путь
  стратегия поиска
  поиск в глубину, поиск в ширину
  эвристический поиск.

Литература

Поиск в глубину и поиск в ширину – базовые стратегии поиска, они описаны в любом учебнике по искусственному интеллекту, см., например, Nilsson (1971, 1980) или Winston (1984). Р.Ковальский в своей книге Kowalski (1980) показывает, как можно использовать аппарат математической логики для реализации этих принципов.

Kowalski R. (1980). *Logic for Problem Solving*. North-Holland.

Nilsson N. J. (1971). *Problem Solving Methods in Artificial Intelligence*. McGraw-Hill.

Nilsson N. J. (1980). *Principles of Artificial Intelligence*. Tioga; also Springer-Verlag, 1981.

Winston P. H. (1984). *Artificial Intelligence* (second edition). Addison-Wesley. [Имеется перевод первого издания: Уинстон П. Искусственный интеллект. – М.: Мир, 1980.]

12 ПОИСК С ПРЕДПОЧТЕНИЕМ: ЭВРИСТИЧЕСКИЙ ПОИСК

Поиск в графах при решении задач, как правило, невозможен без решения проблемы комбинаторной сложности, возникающей из-за быстрого роста числа альтернатив. Эффективным средством борьбы с этим служит эвристический поиск.

Один из путей использования эвристической информации о задаче – это получение численных *эвристических оценок* для вершин пространства состояний. Оценка вершины указывает нам, насколько данная вершина перспективна с точки зрения достижения цели. Идея состоит в том, чтобы всегда продолжать поиск, начиная с наиболее перспективной вершины, выбранной из всего множества кандидатов. Именно на этом принципе основана программа поиска с предпочтением, описанная в данной главе.

12.1. Поиск с предпочтением

Программу поиска с предпочтением можно получить как результат усовершенствования программы поиска в ширину (рис. 11.13). Подобно поиску в ширину, поиск с предпочтением начинается со стартовой вершины и использует множество путей-кандидатов. В то время, как поиск в ширину всегда выбирает для продолжения самый короткий путь (т.е. переходит в вершины наименьшей глубины), поиск с предпочтением вносит в этот принцип следующее усовершенствование: для каждого кандидата вычисляется оценка и для продолжения выбирается кандидат с наилучшей оценкой.

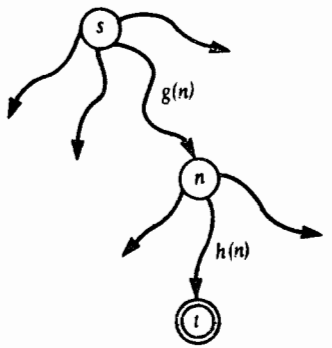


Рис. 12.1. Построение эвристической оценки $f(n)$ стоимости самого дешевого пути из s в t, проходящего через n: $f(n) = g(n) + h(n)$.

Мы будем в дальнейшем предполагать, что для дуг пространства состояний определена функция стоимости $c(n,n')$ – стоимость перехода из вершины n к вершине-преемнику n'.

Пусть f – это эвристическая оценочная функция, при помощи которой мы получаем для каждой вершины n оценку $f(n)$ «трудности» этой вершины. Тогда наиболее перспективной вершиной-кандидатом следует считать вершину, для которой f принимает минимальное значение. Мы будем использовать здесь функцию f специального вида, приводящую к хорошо известному A^*-алгоритму. Функция $f(n)$ будет построена таким образом, чтобы давать оценку стоимости оптимального решающего пути из стартовой вершины s к одной из целевых вершин при условии, что этот путь проходит через вершину n. Давайте предположим, что такой путь существует и что t – это целевая вершина, для которой этот путь минимален. Тогда оценку $f(n)$ можно представить в виде суммы из двух слагаемых (рис. 12.1):

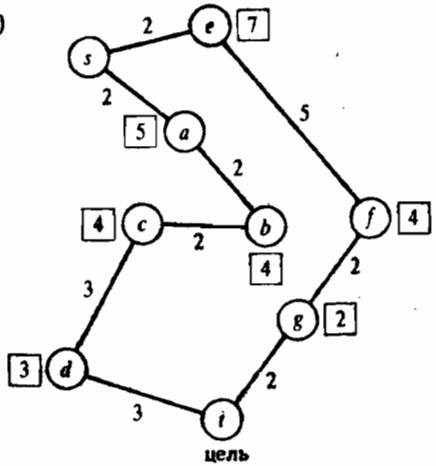
$$f(n) = g(n) + h(n)$$

Здесь $g(n)$ – оценка оптимального пути из s в n, $h(n)$ – оценка оптимального пути из n в t.

Когда в процессе поиска мы попадаем в вершину n, мы оказываемся в следующей ситуации: путь из s

в n уже найден, и его стоимость может быть вычислена как сумма стоимостей составляющих его дуг. Этот путь не обязательно оптимален (возможно, существует более дешевый, еще не найденный путь из s в n), однако стоимость этого пути можно использовать в качестве оценки $g(n)$ минимальной стоимости пути из s в n. Что же касается второго слагаемого $h(n)$, то о нем трудно что-либо сказать, поскольку к этому моменту область пространства состояний, лежащая между n и t, еще не «изучена» программой поиска. Поэтому, как правило, о значении $h(n)$ можно только строить догадки на основании эвристических соображений, т.е. на основании тех знаний о конкретной задаче, которыми обладает алгоритм. Поскольку значение h зависит от предметной области, универсального метода для его вычисления не существует. Конкретные примеры того, как строят эти «эвристические догадки», мы приведем позже. Сейчас же будем считать, что тем или иным способом функция h задана, и сосредоточим свое внимание на деталях нашей программы поиска с предпочтением.

Можно представлять себе поиск с предпочтением следующим образом. Процесс поиска состоит из некоторого числа конкурирующих между собой подпроцессов, каждый из которых занимается своей альтерна-



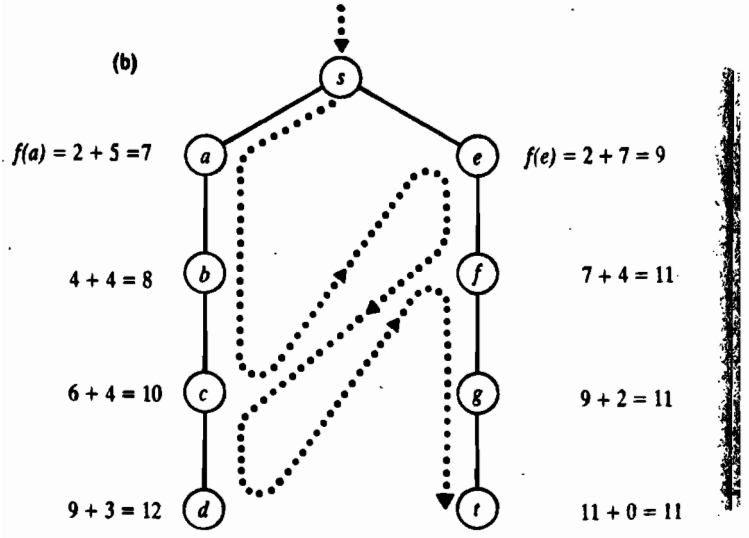


Рис. 12.2. Поиск кратчайшего маршрута из *s* в *t*. (a) Карта со связями между городами; связи помечены своими длинами; в квадратиках указаны расстояния по прямой до цели *t*. (b) Порядок, в котором при поиске с предпочтением происходит обход городов. Эвристические оценки основаны на расстояниях по прямой. Пунктирной линией показано переключение активности между альтернативными путями. Эта линия задает тот порядок, в котором вершины принимаются для *продолжения* пути, а не тот порядок, в котором они порождаются.

тивой, т.е. просматривает свое поддерево. У поддеревьев есть свои поддеревья, их просматривают подпроцессы подпроцессов и т.д. В каждый данный момент среди всех конкурирующих процессов активен только один – тот, который занимается наиболее перспективной к настоящему моменту альтернативой, т.е. альтернативой с наименьшим значением *f*. Остальные процессы спокойно ждут того момента, когда *f*-оценки изменятся и в результате какая-нибудь другая альтернатива станет наиболее перспективной. Тогда производится переключение активности на эту альтернативу. Механизм активации-дезактивации процессов функционирует следующим образом: процесс,

работающий над текущей альтернативой высшего приоритета, получает некоторый «бюджет» и остается активным до тех пор, пока его бюджет не исчерпается. Находясь в активном состоянии, процесс продолжает углублять свое поддерево. Встретив целевую вершину, он выдает соответствующее решение. Величина бюджета, предоставляемого процессу на данный конкретный запуск, определяется эвристической оценкой конкурирующей альтернативы, ближайшей к данной.

На рис. 12.2 показан пример поведения конкурирующих процессов. Дана карта, задача состоит в том, чтобы найти кратчайший маршрут из стартового города s в целевой город t. В качестве оценки стоимости остатка маршрута из города X до цели мы будем использовать расстояние по прямой *расст*(X, t) от X до t. Таким образом,

$$f(X) = g(X) + h(X) = g(X) + \text{расст}(X, t)$$

Мы можем считать, что в данном примере процесс поиска с предпочтением состоит из двух процессов. Каждый процесс прокладывает свой путь – один из двух альтернативных путей: Процесс 1 проходит через a, Процесс 2 – через e. Вначале Процесс 1 более активен, поскольку значения f вдоль выбранного им пути меньше, чем вдоль второго пути. Когда Процесс 1 достигает города c, а Процесс 2 все еще находится в e, ситуация меняется:

$$f(c) = g(c) + h(c) = 6 + 4 = 10$$
$$f(e) = g(e) + h(e) = 2 + 7 = 9$$

Поскольку $f(e) < f(c)$, Процесс 2 переходит к f, а Процесс 1 ждет. Однако

$$f(f) = 7 + 4 = 11$$
$$f(c) = 10$$
$$f(c) < f(f)$$

Поэтому Процесс 2 останавливается, а Процессу 1 дается разрешение продолжать движение, но только до d, так как $f(d) = 12 > 11$. Происходит активация Процесса 2, после чего он, уже не прерываясь, доходит до цели t.

Мы реализуем этот механизм программно при помо-

щи усовершеиствования программы поиска в ширииу (рис. 11.13). Миожество путей-кандидатов представим деревом. Дерево будет изображаться в программе в виде терма, имеющего одиу из двух форм:

(1) **л(В, F/G)** – дерево, состоящее из одной вершииы (листа); **В** – вершина пространства состояний, **G** – g(**В**) (стоимость уже найденного пути из стартовой вершииы в **В**); **F** – f(**В**)= **G** + h(**В**).

(2) **д(В, F/G, Пд)** – дерево с иепустыми поддеревьями; **В** – корень дерева, **Пд** – список поддеревьев; **G** – g(**В**); **F** – *уточненное* значеиие f(**В**), т.е. значеиие f для наиболее перспективиого преемиика вершииы **В**; список **Пд** упорядочен в порядке возрастаиия f-оцеиок поддеревьев.

Уточнение значения f иеобходимо для того, чтобы дать программе возможность распозиавать иаиболее перспективиое поддерево (т.е. поддерево, содержащее иаиболее перспективную концевую вершину) иа любом уровне дерева поиска. Эта модификация f-оценок иа самом деле приводит к обобщению, расширяющему область определеиия функции f. Теперь функция f определена ие только иа вершинах, но и на деревьях. Для одиовершиниых деревьев (листов) n остается первоначальиое определеиие

$$f(n) = g(n) + h(n)$$

Для дерева T с корием n, имеющем преемников m_1, m_2, ..., получаем

$$f(T) = \min_t f(m_t)$$

Программа поиска с предпочтением, составлеииая в соответствии с приведенными выше общими соображеииями, показана на рис 12.3. Ниже даются иекоторые дополнительные пояснения.

Так же, как и в случае поиска в ширину (рис. 11.13), ключевую роль играет процедура **расширить**, имеющая иа этот раз шесть аргументов:

расширить(Путь, Дер, Предел, Дер1, ЕстьРеш, Решение)

Эта процедура расширяет текущее (под)дерево, пока f-оценка остается равной либо меньшей, чем **Предел**.

```
% Поиск с предпочтением
эврпоиск( Старт, Решение):-
    макс_f( Fмакс).              % Fмакс > любой f-оценки
    расширить([],л(Старт,0/0),Fмакс,_,да,Решение).

расширить( П, л( В,_), _, _, да, [В | П] ) :-
    цель( В).

расширить(П,л(В,F/G),Предел,Дер1,ЕстьРеш,Реш) :-
   F <= Предел,
   (bagof(B1/C,(после(В,В1,С),not принадлежит(В1,П)),
          Преемники), !,
     преемспис( G, Преемники, ДД),
     опт_f(ДД, F1),
     расширить(П,д(В,F1/G,ДД),Предел,Дер1,
                                 ЕстьРеш,Реш));
   ЕстьРеш = никогда).          % Нет преемников - тупик

расширить(П,д(В,F/G,[Д|ДД]),Предел,Дер1,
                                 ЕстьРеш,Реш):-
    F <= Предел,
    опт_f(ДД, OF), мин( Предел, OF, Предел1),
    расширить([В|П],Д,Предел1,Д1,ЕстьРеш1,Реш),
    продолжить(П,д(В,F/G,[Д1,ДД]),Предел,Дер1,
                          ЕстьРеш1,ЕстьРеш,Реш).

расширить( _, д( _,_,[]), _, _, никогда, _) :- !.
                % Тупиковое дерево - нет решений

расширить( _, Дер, Предел, Дер, нет, _) :-
    f( Дер, F),   F > Предел.    %Рост остановлен
продолжить( _, _, _, _, да, да, Реш).

продолжить(П,д(В,F/G,[Д1,ДД]),Предел,Дер1,
                          ЕстьРеш1,ЕстьРеш,Реш):-
    (ЕстьРеш1 = нет, встав( Д1, ДД, НДД);
     ЕстьРеш1 = никогда, НДД = ДД),
    опт_f( НДД, F1),
    расширить(П,д(В,F1/G,НДД),Предел,Дер1,
                                 ЕстьРеш, Реш).

преемспис( _, [], []).

преемспис( G0, [B/C | BB], ДД) :-
    G is G0 + C,
    h( B, H),                    % Эвристика h(B)
    F is G + H,
    преемспис( G0, BB, ДД1),
    встав( л(B, F/G), ДД1, ДД).
```

```
% Вставление дерева Д в список деревьев ДД с сохранением
% упорядоченности по f-оценкам
встав( Д, ДД, [Д | ДД] ) :-
    f( Д, F), опт_f( ДД, F1),
    F =< F1, !.

встав( Д, [Д1 | ДД], [Д1 | ДД1] ) ) :-
  встав( Д, ДД, ДД1).

% Получение f-оценки
f( л( _, F/_), F).                  % f-оценка листа

f( д( _, F/_, _), F).               % f-оценка дерева

опт_f( [Д|_], F) :-                 % Наилучшая f-оценка для
  f( Д, F).                         % списка деревьев

опт_f( [], Fмакс) :-                % Нет деревьев:
    макс_f( Fмакс).                 % плохая f-оценка

мин( X, Y, X) :-
  X =< Y, !.

мин( X, Y, Y).
```

Рис. 12.3. Программа поиска с предпочтением.

Аргументы процедуры **расширить** имеют следующий смысл:

Путь Путь между стартовой вершиной и корнем дерева **Дер**.

Дер Текущее (под)дерево поиска.

Предел Предельное значение *f*-оценки, при котором допускается расширение.

Дер1 Дерево **Дер**, расширенное в пределах ограничения **Предел**; *f*-оценка дерева **Дер1** больше, чем **Предел** (если только при расширении не была обнаружена целевая вершина).

ЕстьРеш Индикатор, принимающий значения «да», «нет» и «никогда».

Решение Решающий путь, ведущий из стартовой вершины через дерево **Дер1** к целевой вершине

и имеющий стоимость, не превосходящую ограниченне **Предел** (если такая целевая вершина была обнаружена).

Переменные **Путь**, **Дер**, и **Предел** – это «входные» параметры процедуры **расширить** в том смысле, что при каждом обращении к **расширить** они всегда конкретизированы. Процедура **расширить** порождает результаты трех видов. Какой вид результата получен, можно определить по значению индикатора **ЕстьРеш** следующим образом:

(1) **ЕстьРеш = да.**

Решение = решающий путь, найденный при расширении дерева **Дер** с учетом ограничения **Предел**.

Дер1 = неконкретизировано.

(2) **ЕстьРеш = нет**

Дер1 = дерево **Дер**, расширенное до тех пор, пока его f-оценка не превзойдет **Предел** (см. рис. 12.4).

Решение = неконкретизировано.

(3) **ЕстьРеш = никогда.**

Дер1 и Решение = неконкретизированы.

В последнем случае **Дер** является «тупиковой» альтернативой, и соответствующий процесс никогда не будет реактивирован для продолжения просмотра этого дерева. Случай этот возникает тогда, когда f-оценка дерева **Дер** не превосходит ограничения **Предел**, однако дерево не может «расти» потому, что ни один его лист не имеет преемников, или же любой преемник порождает цикл.

Некоторые предложения процедуры **расширить** требуют пояснений. Предложение, относящееся к наиболее сложному случаю, когда **Дер** имеет поддеревья, т.е.

Дер = д(В, F/G, [Д | ДД])

означает следующее. Во-первых, расширению подвергается наиболее перспективное дерево **Д**. В качестве ограничения этому дереву выдается не **Предел**, а не-

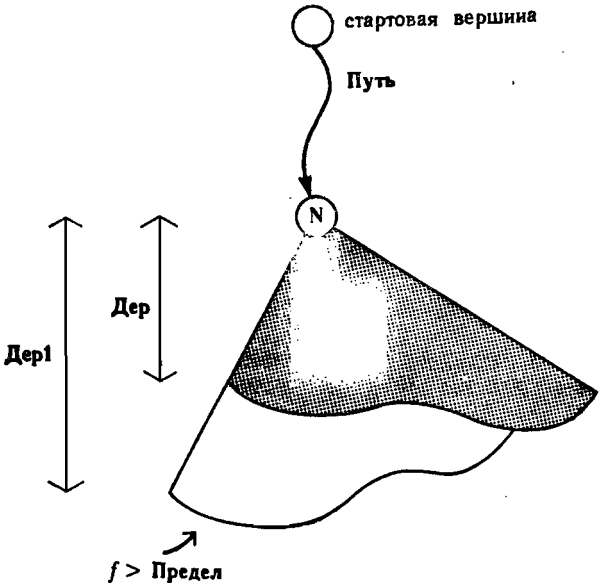


Рис. 12.4. Отношение **расширить**: расширение дерева **Дер** до тех пор, пока f-оценка не превзойдет **Предел**, приводит к дереву **Дер1**.

которое, возможно, меньшее значение **Предел1**, зависящее от f-оценок других конкурирующих поддеревьев **ДД**. Тем самым гарантируется, что «растущее» дерево – это всегда наиболее перспективное дерево, а переключение активности между поддеревьями происходит в соответствии с их f-оценками. После того, как самый перспективный кандидат расширен, вспомогательная процедура **продолжить** решает, что делать дальше, а это зависит от типа результата, полученного после расширения. Если найдено решение, то оно и выдается, в противном случае процесс расширения деревьев продолжается.

Предложение, относящееся к случаю

Дер = л(B, F/G)

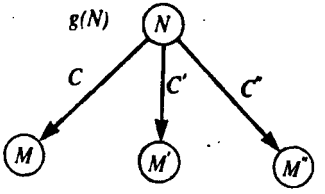
порождает всех преемников вершины **B** вместе со стоимостями дуг, ведущих в них из **B**. Процедура **преемспис** формирует список поддеревьев, соответ-

ствующих вершинам-преемникам, а также вычисляет их g- и f-оценки, как показано на рис. 12.5. Затем полученное таким образом дерево подвергается расширению с учетом ограничения **Предел**. Если преемников нет, то переменной **ЕстьРеш** придается значение «никогда» и в результате лист **В** покидается навсегда.

Другие отношения:

после(В, В1, С)	**В1** – преемник вершины **В**; **С** – стоимость дуги, ведущей из **В** в **В1**.
h(В, Н)	**Н** – эвристическая оценка стоимости оптимального пути из вершины **В** в целевую вершину.
макс_f(Fмакс)	**Fмакс** – некоторое значение, задаваемое пользователем, про которое известно, что оно больше любой возможной f-оценки.

В следующих разделах мы покажем на примерах, как можно применить нашу программу поиска с предпочтением к конкретным задачам. А сейчас сделаем несколько заключительных замечаний общего характера относительно этой программы. Мы реализовали один из вариантов эвристического алгоритма, известного в литературе как A^*-алгоритм (ссылки на литературу см. в конце главы). A^*-алгоритм привлек внимание многих исследователей. Здесь мы приведем один важный результат, полученный в результате математического анализа A^*-алгоритма:



$g(M) = g(N) + C$

$f(M) = g(M) + h(M)$

Рис. 12.5. Связь между g-оценкой вершины B и f- и g-оценками ее "детей" в пространстве состояний.

Алгоритм поиска пути называют *допустимым*, если он всегда отыскивает оптимальное решение (т.е. путь минимальной стоимости) при условии, что такой путь существует. Наша реализация алгоритма поиска, пользуясь механизмом возвратов, выдает все существующие решения, поэтому, в нашем случае, условием допустимости следует считать оптимальность *первого* из найденных решений. Обозначим через $h^*(n)$ стоимость оптимального пути из произвольной вершины n в целевую вершину. Верна следующая теорема о допустимости A^*-алгоритма: A^*-алгоритм, использующий эвристическую функцию h, является допустимым, если

$$h(n) \leq h^*(n)$$

для всех вершин n пространства состояний.

Этот результат имеет огромное практическое значение. Даже если нам не известно точное значение h^*, нам достаточно найти какую-либо нижнюю грань h^* и использовать ее в качестве h в A^*-алгоритме – оптимальность решения будет гарантирована.

Существует тривиальная нижняя грань, а именно:

$h(n) = 0$, для всех вершин n пространства состояний.

И при таком значении h допустимость гарантирована. Однако такая оценка не имеет никакой эвристической силы и ничем не помогает поиску. A^*-алгоритм при $h=0$ ведет себя аналогично поиску в ширину. Он, действительно, превращается в поиск в ширину, если, кроме того, положить $c(n,n')=1$ для всех дуг (n,n') пространства состояний. Отсутствие эвристической силы оценки приводит к большой комбинаторной сложности алгоритма. Поэтому хотелось бы иметь такую оценку h, которая была бы нижней гранью h^* (чтобы обеспечить допустимость) и, кроме того, была бы как можно ближе к h^* (чтобы обеспечить эф-

фективность). В идеальном случае, если бы нам была известна сама точная оценка h^*, мы бы ее и использовали: A^*-алгоритм, пользующийся h^*, находит оптимальное решение сразу, без единого возврата.

Упражнение

12.1. Определите отношения **после**, **цель** и **h** для задачи поиска маршрута рис. 12.2. Посмотрите, как наш алгоритм поиска с предпочтением будет вести себя при решении этой задачи.

12.2. Поиск с предпочтением применительно к головоломке «игра в восемь»

Если мы хотим применить программу поиска с предпочтением, показанную на рис. 12.3, к какой-нибудь задаче, мы должны добавить к нашей программе отношения, отражающие специфику этой конкретной задачи. Эти отношения определяют саму задачу («правила игры»), а также вносят в алгоритм эвристическую информацию о методе ее решения. Эвристическая информация задается в форме эвристической функции.

```
/* Процедуры, отражающие специфику головоломки
"игра в восемь".
Текущая ситуация представлена списком положений фишек;
первый элемент списка соответствует пустой клетке.
Пример:
        3 | 1 2 3 |
        2 | 8   4 |        Эта позиция представляется так:
        1 | 7 6 5 |
          -------
            1 2 3        [2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 1/2]
"Пусто" можно перемещать в любую соседнюю клетку,
т.е. "Пусто" меняется местами со своим соседом.
*/

после( [Пусто | Спис], [Фшк | Спис1], 1) :-
                    % Стоимости всех дуг равны 1
```

```
    перест( Пусто, Фшк, Спис, Спис1).
                    % Переставив Пусто и Фшк, получаем СПИС1
перест( П, Ф, [Ф | С], [П | С] ) :-
    расст( П, Ф, 1).
перест( П, Ф, [Ф1 | С], [Ф1 | С1] ) :-
    перест( П, Ф, С, С1).

расст( X/Y, X1/Y1, Р) :-
            % Манхеттеновское расстояние между клетками
    расст1( X, X1, Рх),
    расст1( Y, Y1, Ру),
    Р is Рх + Ру.

расст1( А, В, Р) :-
    Р is А-В,   Р >= 0, ! ;
    Р is В-А.

% Эвристическая оценка h равна сумме расстояний фишек
% от их "целевых" клеток плюс "степень упорядоченности",
% умноженная на 3

h( [ Пусто | Спис], Н) :-
    цель( [Пусто1 | Цспис] ),
    сумрасст( Спис, ЦСпис, Р),
    упоряд( Спис, Уп),
    Н is Р + 3*Уп.

сумрасст( [], [], 0).

сумрасст( [Ф | С], [Ф1 | С1], Р) :-

    расст( Ф, Ф1, Р1),
    сумрасст( С, С1, Р2),
    Р is Р1 + Р2.

упорид( [Первый | С], Уп) :-
    упорид( [Первый | С], Первый, Уп).

упорид( [Ф1, Ф2 | С], Первый, Уп) :-
    очки( Ф1, Ф2, Уп1),
    упорид( [Ф2 | С], Первый, Уп2),
    Уп is Уп1 + Уп2.

упорид( [Последний], Первый, Уп) :-
    очки( Последний, Первый, Уп).

очки( 2/2, _, 1) :- !.          % Фишка в центре - 1 очко
очки( 1/3, 2/3, 0) :- !.
                % Правильная последовательность - 0 очков
```

```
очки( 2/3, 3/3, 0) :- !.
очки( 3/3, 3/2, 0) :- !.
очки( 3/2, 3/1, 0) :- !.
очки( 3/1, 2/1, 0) :- !.
очки( 2/1, 1/1, 0) :- !.
очки( 1/1, 1/2, 0) :- !.
очки( 1/2, 1/3, 0) :- !.
очки( _, _, 2).   % Неправильная последовательность
цель( [2/2,1/3,2/3,3/3,3/2,3/1,2/1,1/1,1/2] ).
% Стартовые позиции для трех головоломок
старт1( [2/2,1/3,3/2,2/3,3/3,3/1,2/1,1/1,1/2] ).
                % Требуется для решения 4 шага
старт2( [2/1,1/2,1/3,3/3,3/2,3/1,2/2,1/1,2/3] ).
                % 5 шагов
старт3( [2/2,2/3,1/3,3/1,1/2,2/1,3/3,1/1,3/2] ).
                % 18 шагов

% Отображение решающего пути в виде списка позиций на доске
показреш( []).
показреш( [ Поз | Спис] :-
    показреш( Спис),
    nl, write( '---'),
    показпоз( Поз).

% Отображение позиции на доске
показпоз( [S0,S1,S2,S3,S4,S5,S6,S7,S8] ) :-
    принадлежит( Y, [3,2,1] ),      % Порядок Y-координат
    nl, принадлежит( X, [1,2,3] ), % Порядок X-координат
    принадлежит(Фшк-X/Y,
    [' '-S0,1-S1,2-S2,3-S3,4-S4,5-S5,6-S6,7-S7,8-S8]),
    write( Фшк),
    fail.          %Возврат с переходом к следующей клетке
    показпоз(_).
```

Рис. 12.6. Процедуры для головоломки "игра в восемь", предназначенные для использования программой поиска с предпочтением рис. 12.3.

Существуют три отношения, отражающих специфику конкретной задачи:

после(Верш, Верш1, Ст)

Это отношение истинно, когда в пространстве состояний существует дуга стоимостью **Ст** между вершинами **Верш** и **Верш1**.

цель(Верш)

Это отношение истинно, если **Верш** – целевая вершина.

h(Верш, Н)

Здесь **Н** – эвристическая оценка стоимости самого дешевого пути из вершины **Верш** в целевую вершину.

В данном и следующих разделах мы определим эти отношения для двух примеров предметных областей: для головоломки «игра в восемь» (описанной в разделе 11.1) и планирования прохождения задач в многопроцессорной системе.

Отношения для «игры в восемь» показаны на рис. 12.6. Вершина пространства состояний – это некоторая конфигурация из фишек на игровой доске. В программе она задается списком текущих положений фишек. Каждое положение определяется парой координат X/Y. Элементы списка располагаются в следующем порядке:

(1) текущее положение пустой клетки,

(2) текущее положение фишки 1,

(3) текущее положение фишки 2,

...

Целевая ситуация (см. рис. 11.3) определяется при помощи предложения

цель([2/2,1/3,2/3,3/3,3/2,3/1,2/1,1/1,1/2]).

Имеется вспомогательное отношение

расст(К1, К2, Р)

Р – это «манхеттеновское расстояние» между клетками К1 и К2, равное сумме двух расстояний между К1 и К2: расстояния по горизонтали и расстояния по вертикали.

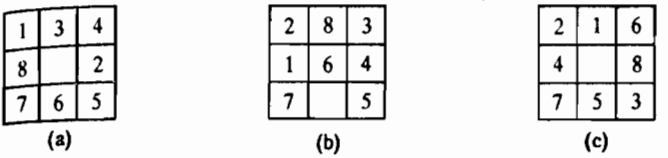


Рис.12.7. Три стартовых позиции для "игры в восемь": (a) решение требует 4 шага; (b) решение требует 5 шагов; (c) решение требует 18 шагов.

Наша задача – минимизировать *длину* решения, поэтому мы положим стоимости всех дуг пространства состояний равными 1. В программе рис. 12.6 даны также определения трех начальных позиций (см. рис. 12.7).

Эвристическая функция *h*, запрограммирована как отношение

h(Поз, Н)

Поз – позиция на доске; **Н** вычисляется как комбинация из двух оценок:

(1) **сумрасст** – «суммарное расстояние» восьми фишек, находящихся в позиции **Поз**, от их положений в целевой позиции. Например, для начальной позиции, показанной на рис. 12.7(a), **сумрасст** = 4.

(2) **упоряд** – степень упорядоченности фишек в текущей позиции по отношению к тому порядку, в котором они должны находиться в целевой позиции. Величина **упоряд** вычисляется как сумма очков, приписываемых фишкам, согласно следующим правилам:

- фишка в центральной позиции – 1 очко;
- фишка не в центральной позиции, и непосредственно за ней следует (по часовой стрелке) та фишка, какая и должна за ней следовать в целевой позиции – 0 очков.
- то же самое, но за фишкой следует «не та» фишка – 2 очка.

Например, для начальной позиции рис.12.7(a), **упоряд** = 6.

Эвристическая оценка **Н** вычисляется как сумма

Н = сумрасст + 3 * упоряд

Эта эвристическая функция хорошо работает в том смысле, что она весьма эффективно направляет поиск к цели. Например, при решении головоломок рис. 12.7(a) и (b) первое решение обнаруживается без единого отклонения от кратчайшего решающего пути. Другими словами, кратчайшие решения обнаруживаются сразу, без возвратов. Даже трудная головоломка рис. 12.7 (c) решается почти без возвратов. Но данная эвристическая функция страдает тем недостатком, что она не является допустимой: нет гарантии, что более короткие пути обнаруживаются раньше более длинных. Дело в том, что для функции h условие $h \leq h^*$ выполнено не для всех вершин пространства состояний. Например, для начальной позиции рис. 12.7 (a)

$$h = 4 + 3 * 6 = 22, \quad h^* = 4$$

С другой стороны, оценка «суммарное расстояние» допустима: для всех позиций

$$\text{сумрасст} \leq h^*$$

Доказать это неравенство можно при помощи следующего рассуждения: если мы ослабим условия задачи и разрешим фишкам взбираться друг на друга, то каждая фишка сможет добраться до своего целевого положения по траектории, длина которой в точности равна манхеттеновскому расстоянию между ее начальным и целевым положениями. Таким образом, длина оптимального решения упрощенной задачи будет в точности равна сумрасст. Однако в исходном варианте задачи фишки взаимодействуют друг с другом и мешают друг другу, так что им уже трудно идти по своим кратчайшим траекториям. В результате длина оптимального решения окажется больше либо равной сумрасст.

Упражнение

12.2. Введите в программу поиска с предпочтением, приведенную на рис. 12.3, подсчет числа вершин, порожденных в процессе поиска. Один из простых способов это сделать – хранить текущее число вершин в виде факта, устанавливаемого при помощи assert. Всегда, когда порождаются новые вершины, уточнять это значение при помощи retract

и assert. Проведите эксперименты с различными эвристическими функциями задачи «игра в восемь» с целью оценить их эвристическую силу. Используйте для этого вычисленное количество порожденных вершин.

12.3. Применение поиска с предпочтением к планированию выполнения задач

Рассмотрим следующую задачу планирования. Дана совокупность *задач* t_1, t_2, ..., имеющих времена выполнения соответственно *T1*, *T2*, ... Все эти задачи нужно решить на m идентичных процессорах. Каждая задача может быть решена на любом процессоре, но в каждый данный момент каждый процессор решает только одну из задач. Между задачами существует отношение предшествования, определяющее, какие задачи (если таковые есть) должны быть завершены, прежде чем данная задача может быть запущена. Необходимо распределить задачи между процессорами без нарушения отношения предшествования, причем таким образом, чтобы вся совокупность задач была решена за минимальное время. Время, когда последняя задача в соответствии с выработанным планом завершает свое решение, называется *временем окончания* плана. Мы хотим минимизировать время окончания по всем возможным планам.

На рис. 12.8 показан пример задачи планирования, а также приведено два корректных плана, один из которых оптимален. Из примера видно, что оптимальный план обладает одним интересным свойством, а именно в нем может предусматриваться «время простоя» процессоров. В оптимальном плане рис. 12.8 процессор 1, выполнив задачу t_1, ждет в течение двух квантов времени, несмотря на то, что он мог бы начать выполнение задачи t_3.

Один из способов построить план можно грубо сформулировать так. Начинаем с пустого плана (с незаполненными временными промежутками для каждого процессора) и постепенно включаем в него задачи

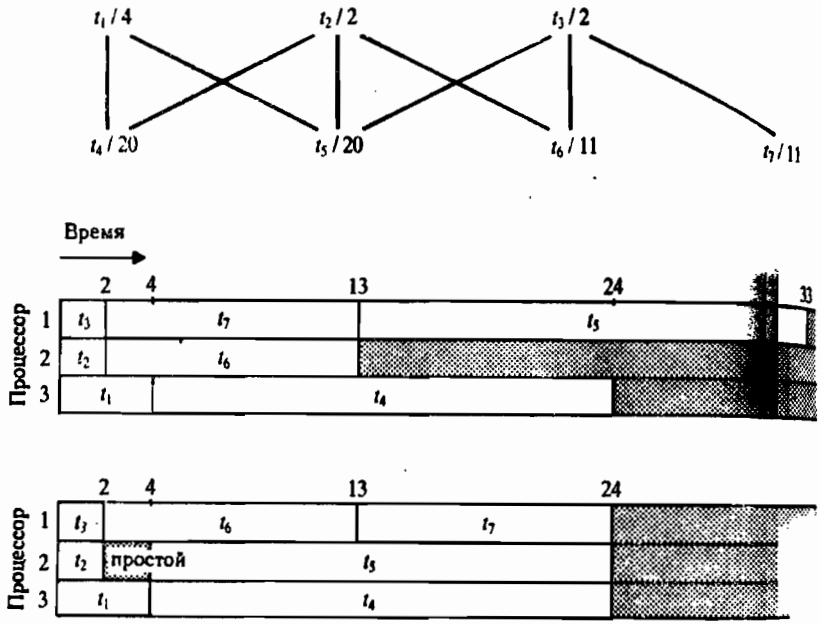


Рис. 12.8. Планирование прохождения задач в многопроцессорной системе для 7 задач и 3 процессоров. Вверху показано предшествование задач и величины продолжительности их решения. Например, задача t_5 требует 20 квантов времени, причем ее выполнение может начаться только после того, как будет завершено решение трех других задач t_1, t_2 и t_3. Показано два корректных плана: оптимальный план с временем окончания 24 и субоптимальный – с временем окончания 33. В данной задаче любой оптимальный план должен содержать время простоя.
Coffman/Denning, *Operating Systems Theory.* © 1973, p.86.. Приведено с разрешения Prentice-Hall, Englewood Cliffs, New Jersey.

одну за другой, пока все задачи не будут исчерпаны. Как правило, на каждом шагу мы будем иметь несколько различных возможностей, поскольку окажется, что одновременно несколько задач-кандидатов ждут своего выполнения. Таким образом, для составления плана потребуется перебор. Мы можем сформу-

лировать задачу планирования в терминах пространства состояний следующим образом:

- состояния – это частично составленные планы;
- преемник частичного плана получается включением в план еще одной задачи; другая возможность – оставить процессор, только что закончивший свою задачу, в состоянии простоя;
- стартовая вершина – пустой план;
- любой план, содержащий все задачи, – целевое состояние;
- стоимость решения (подлежащая минимизации) – время окончания целевого плана;
- стоимость перехода от одного частичного плана к другому равна K_2–K_1, где K_1, K_2 – времена окончания этих планов.

Этот грубый сценарий требует некоторых уточнений. Во-первых, мы решим заполнять план в порядке возрастания времен, так что задачи будут включаться в него слева направо. Кроме того, при добавлении каждой задачи следует проверять, выполнены ли ограничения, связанные с отношениями предшествования. Далее, не имеет смысла оставлять процессор бездействующим на неопределенное время, если имеются задачи, ждущие своего запуска. Поэтому мы разрешим процессору простаивать только до того момента, когда какой-нибудь другой процессор завершит выполнение своей задачи. В этот момент мы еще раз вернемся к свободному процессору с тем, чтобы рассмотреть возможность приписывания ему какой-нибудь задачи.

Теперь нам необходимо принять решение относительно представления проблемных ситуаций, т.е. частичных планов. Нам понадобится следующая информация:

(1) список ждущих задач вместе с их временами выполнения;
(2) текущая загрузка процессоров задачами.

Добавим также для удобства программирования

(3) время окончания (частичного) плана, т.е.

самое последнее время окончания задачи среди всех задач, приписанных процессорам.

Список ждущих задач вместе с временами их выполнения будем представлять в программе при помощи списка вида

[Задача1/T1, Задача2/T2, ...]

Текущую загрузку процессоров будем представлять как список решаемых задач, т. е. список пар вида

Задача/ВремяОкончания

В списке *m* таких пар, по одной на каждый процессор. Новая задача будет добавляться к плану в момент, когда закончится первая задача из этого списка. В связи с этим мы должны постоянно поддерживать упорядоченность списка загрузки по возрастанию времени окончания. Эти три компоненты частичного плана (ждущие задачи, текущая загрузка и время окончания плана) будут объединены в одно выражение вида

Ждущие * Активные * ВремяОкончания

Кроме этой информации у нас есть ограничения, налагаемые отношениями предшествования, которые в программе будут выражены в форме отношения

предш(ЗадачаX, ЗадачаY)

Рассмотрим теперь эвристическую оценку. Мы будем использовать довольно примитивную эвристическую функцию, которая не сможет обеспечить высокую эффективность управления алгоритмом поиска. Эта функция допустима, так что получение оптимального плана будет гарантировано. Однако следует заметить, что для решения более серьезных задач планирования потребуется более мощная эвристика.

Нашей эвристической функцией будет оптимистическая оценка времени окончания частичного плана с учетом всех ждущих задач. Оптимистическая оценка будет вычисляться в предположении, что два из ограничений, налагаемых на действительно корректный план, ослаблены:

(1) не учитываются отношения предшествования;

(2) делается (не реальное) допущение, что воз-

можно распределенное выполнение задачи одновременно на нескольких процессорах, причем сумма времен выполнения задачи на процессорах равна исходному времени выполнения этой задачи на одном процессоре.

Пусть времена выполнения ждущих задач равны T_1, T_2, ..., а времена окончания задач, выполняемых на процессорах – K_1, K_2, ... Тогда оптимистическая оценка времени *ОбщКон* окончания всех активных к настоящему моменту, а также всех ждущих задач имеет вид:

$$ОбщКон = (\sum_i T_i + \sum_j K_j)/m$$

где m – число процессоров. Пусть **время окончания** текущего частичного плана равно

$$Кон = \max_j(K_j)$$

Тогда эвристическая оценка H (дополнительное время для включения в частичный план ждущих задач) определяется следующим выражением:

if *ОбщКон*>*Кон* then H=*ОбщКон*–*Кон* else H=0

Программа, содержащая определения отношений, связанных с пространством состояний нашей задачи планирования, приведена полностью на рис. 12.9. Эта программа включает в себя также спецификацию конкретной задачи планирования, показанной на рис. 12.3. Одно из оптимальных решений, полученных в процессе поиска с предпочтением в определенном таким образом пространстве состояний, показано на рис. 12.8.

Проект

Вообще говоря, задачи планирования характеризуются значительной комбинаторной сложностью. Наша простая эвристическая функция не обеспечивает высокой эффективности управления поиском. Предложите другие эвристические функции и проведите с ними эксперименты.

```
/* Отношения для задачи планирования.
Вершины пространства состояний - частичные планы,
записываемые как

    [ Задача1/Т1, Задача2/Т2, ...]*
    [ Задача1/К1, Задача2/К2, ...]* ВремяОкончания

В первом списке указываются ждущие задачи и  продолжи-
тельности их выполнения; во втором - текущие решаемые
задачи и их времена окончания, упорядоченные так, что-
бы выполнялись неравенства К1≤К2, К2≤К3, ... .
Время  окончания  плана - самое  последнее по времени
время окончания задачи.
*/

после(Задачи1*[_/К|Акт1]*Кон1,Задачи2*Акт2*Кон2,Ст):-
    удалить( Задача/Т, Задачи1, Задачи2),
                                % Взять ждущую задачу
    not( принадлежит(Здч1/_,Задачи2),
         раньше(ЗДЧ,Задача) ),
                                % Проверить предшествовани
    not( принадлежит(Здч1/К1,Акт1),К1<К2,
         раньше(К1,Задача) ),  % Активные задачи
    Время is К + Т,
                    % Время окончания работающей задачи
    встав( ЗадачаВремя, Акт1, Акт2, Кон1, Кон2),
    Ст is Кон2 - Кон1.

после(Задачи*[_/К|Акт1]*Кон,Задачи2*Акт2*Кон,0):-
    вставпростой( К, Акт1, Акт2).
                    % Оставить процессор бездействующим

раньше( Задача1, Задача2) :-
                    % В соответствии с предшествованием
    предш( Задача1, Задача2).
                    % Задача1 раньше, чем Задача2

раньше( Здч1, Здч2) :-
    предш( Здч, Здч2),
    раньше( Здч1, Здч).

встав(Здч/А,[Здч1/В|Спис],[Здч/А,Здч1/В|Спис],К,К):-
                    % Список задач упорядочен
    А =< В, !.

встав(Здч/А,[Здч1/В|Спис],[Здч1/В|Спис1],К1,К2) :-
```

```
   встав( Здч/А, Спис, Спис1, К1, К2).
встав( Здч/А, [], [Здч/А], _, А).
вставпростой(А,[Здч/В|Спис],[простой/В,Здч/В|Спис]):-
                        % Оставить процессор бездействующим
   А < В, !             % до ближайшего времени окончания

вставпростой(А,[Здч/В|Спис],[Здч/В|Спис1]) :-
   вставпростой( А, Спис, Спис1 ).
удалить( А, [А | Спис], Спис ).
                        % Удалить элемент из списка
удалить( А, [В | Спис], [В | Спис1] ) :-
   удалить( А, Спис, Спис1 ).
цель( [] *_*_).         % Целевое состояние: нет ждущих задач

% Эвристическая оценка частичного плана основана на
% оптимистической оценке последнего времени окончания
% этого частичного плана,
% дополненного всеми остальными ждущими задачами.

h( Задачи * Процессоры * Кон, Н) :-
   сумвремя( Задачи, СумВремя),
                        % Суммарная продолжительность
                        % ждущих задач
   всепроц( Процессоры, КонВремя, N),
                        % КонВремя - сумма времен окончания
                        % для процессоров, N - их количество
   ОбщКон is ( СумВремя + КонВремя)/N,
   ( ОбщКон > Кон, !, Н is ОбщКон - Кон; Н = 0).
умвремя( [], 0).
сумвремя( [_/Т | Задачи], Вр) :-
   сумвремя( Задачи, Вр1),
   Вр is Вр1 + Т.
всепроц( [], 0, 0).
всепроц( [_/Т | СписПроц], КонВр, N) :-
   всепроц( СписПроц, КонВр1, N1),
   N is N1 + 1,
   КонВр is КонВр1 + Т.

% Граф предшествования задач
```

```
предш(t1,t4). предш(t1,t5). предш(t2,t4).
предш(t2,t5). предш(t3,t5). предш(t3,t6).
предш(t3,t7).

% Стартовая вершина
старт( [t1/4,t2/2,t3/2,t4/20,t5/20,t6/11,t7/11] *
       [простой/0,простой/0,простой/0] * 0 ).
```

Рис. 12.9. Отношения для задачи планирования. Даны также определения отношений для конкретной задачи планирования с рис. 12.8: граф предшествования и исходный (пустой) план в качестве стартовой вершины.

Резюме

- Для оценки степени удаленности некоторой вершины пространства состояний от ближайшей целевой вершины можно использовать эвристическую информацию. В этой главе были рассмотрены численные эвристические оценки.
- Эвристический принцип поиска с предпочтением направляет процесс поиска таким образом, что для продолжения поиска всегда выбирается вершина, наиболее перспективная с точки зрения эвристической оценки.
- В этой главе был запрограммирован алгоритм поиска, основанный на указанном принципе и известный в литературе как A^*-алгоритм.
- Для того, чтобы решить конкретную задачу при помощи A^*-алгоритма, необходимо определить пространство состояний и эвристическую функцию. Для сложных задач наиболее трудным моментом является подбор хорошей эвристической функции.
- *Теорема о допустимости* помогает установить, всегда ли A^*-алгоритм, использующий некоторую конкретную эвристическую функцию, находит оптимальное решение.

Литература

Программа поиска с предпочтением, представленная в настоящей главе, – это один из многих вариантов похожих друг на друга программ, из которых A*-алгоритм наиболее популярен. Общее описание A*-алгоритма можно найти в книгах Nilsson (1971, 1980) или Winston (1984). Теорема о допустимости впервые доказана авторами статьи Hart, Nilsson, and Raphael (1968). Превосходное и строгое изложение многих разновидностей алгоритмов поиска с предпочтением и связанных с ними математических результатов дано в книге Pearl (1984). В статье Doran and Michie (1966) впервые изложен поиск с предпочтением, управляемый оценкой расстояния до цели.

Головоломка «игра в восемь» использовалась многими исследователями в области искусственного интеллекта в качестве тестовой задачи при изучении эвристических принципов (см., например, Doran and Michie (1966), Michie and Ross (1970) и Gaschnig (1979)).

Задача планирования, рассмотренная в настоящей главе, также как и многие ее разновидности, возникает во многих прикладных областях в ситуации, когда необходимо спланировать обслуживание запросов на ресурсы. Один из примеров – операционные системы вычислительных машин. Задача планирования со ссылкой на это конкретное приложение изложена в книге Coffman and Denning (1973).

Найти хорошую эвристику – дело важное и трудное, поэтому изучение эвристик – одна из центральных тем в искусственном интеллекте. Существуют, однако, некоторые границы, за которые невозможно выйти, двигаясь в направлении улучшения качества эвристик. Казалось бы, все, что необходимо для эффективного решения комбинаторной задачи – это найти мощную эвристику. Однако есть задачи (в том числе многие задачи планирования), для которых не существует универсальной эвристики, обеспечивающей во всех случаях как эффективность, так и допустимость. Многие теоретические результаты, имеющие отношение к этому ограничению, собраны в работе Garey and Johnson (1979).

Coffman E.G. and Denning P.J. (1973). *Operating Systems Theory*. Prentice-Hall.

Doran J. and Michie D. (1966). Experiments with the graph traverser program. *Proc. Royal Society of London* **294**(A): 235–259.

Garey M.R. and Johnson D.S. (1979). *Computers and Intractability*. W.H.Freeman. [Имеется перевод: Гэри М., Джонсон Д.С. Вычислительные машины и труднорешаемые задачи.– М.: Мир, 1982.]

Gaschnig J. (1979). Performance measurement and analysis of certain search algorithms. Carnegie-Mellon University: Computer Science Department. Technical Report CMU-CS-79-124 (Ph.D.Thesis).

Hart P.E., Nilsson N.J. and Raphael B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Sciences and Cybernetics* **SSC-4(2)**:100-107

Michie D. and Ross R. (1970). Experiments with the adaptive graph traverser. *Machine Intelligence* 5: 301–308.

Nilsson N.J. (1971). *Problem – Solving Methods in Artificial Intelligence*. McGraw-Hill. [Имеется перевод: Нильсон Н. Искусственный интеллект. Методы поиска решений. – М.: Мир, 1973.]

Nilsson N.J. (1980). *Principles of Artificial Intelligence*. Tioga; also Springer-Verlag.

Pearl J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Winston P. H. (1984). *Artificial Intelligence* (second edition). Addison-Wesley. [Имеется перевод первого издания: Уинстон П. Искусственный интеллект. – М.: Мир, 1980.]

13 СВЕДЕНИЕ ЗАДАЧ К ПОДЗАДАЧАМ. И/ИЛИ–ГРАФЫ

Представление в виде И/ИЛИ-графов наиболее хорошо приспособлено для задач, которые естественным образом разбиваются на взаимно независимые подзадачи. Примерами таких задач могут служить поиск маршрута, символическое интегрирование, а также игровые задачи, доказательство теорем и т.п. В этой главе мы разработаем программы для поиска в И/ИЛИ-графах, в том числе программу поиска с предпочтением, управляемого эвристиками.

13.1. Представление задач в виде И/ИЛИ–графов

В главах 11 и 12, говоря о решении задач, мы сконцентрировали свое внимание на пространстве состояний как средстве представления этих задач. В соответствии с таким подходом решение задач сводилось к поиску пути в графе пространства состояний. Однако для некоторых категорий задач представление в форме И/ИЛИ-графа является более естественным. Такое представление основано на разбиении задач на подзадачи. Разбиение на подзадачи дает преимущества в том случае, когда подзадачи взаимно независимы, а, следовательно, и решать их можно независимо друг от друга.

Проиллюстрируем это на примере. Рассмотрим задачу отыскания на карте дорог маршрута между двумя заданными городами, как показано на рис. 13.1. Не будем пока учитывать длину путей. Разумеется, эту задачу можно сформулировать как поиск пути в про-

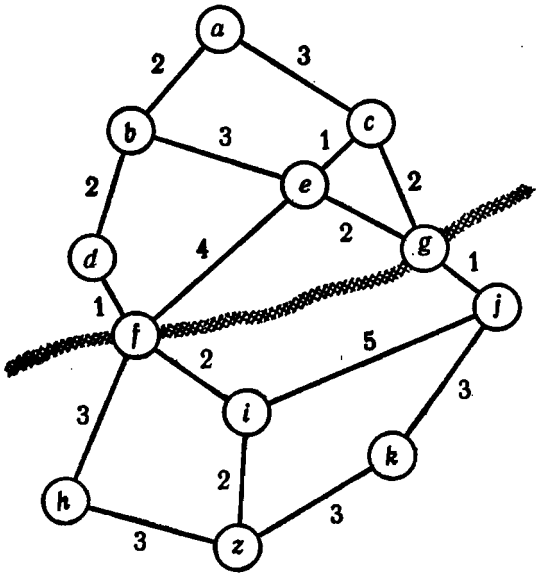


Рис. 13.1. Поиск маршрута из *a* в *z* на карте дорог. Через реку можно переправиться в городах *f* и *g*. И/ИЛИ-представление этой задачи показано на рис. 13.2.

странстве состояний. Соответствующее пространство состояний выглядело бы в точности, как карта рис. 13.1: вершины соответствуют городам, дуги – непосредственным связям между городами. Тем не менее давайте построим другое представление, основанное на естественном разбиении этой задачи на подзадачи.

На карте рис. 13.1 мы видим также реку. Допустим, что переправиться через нее можно только по двум мостам: один расположен в городе *f*, другой – в городе *g*. Очевидно, что искомый маршрут обязательно должен проходить через один из мостов, а значит, он должен пройти либо через *f*, либо через *g*. Таким образом, мы имеем две главных альтернативы:

Для того, чтобы найти путь из *a* в *z*,
необходимо найти *одно из двух*:

(1) путь из *a* в *z*, проходящий через *f*, **или**
(2) путь из *a* в *z*, проходящий через *g*.

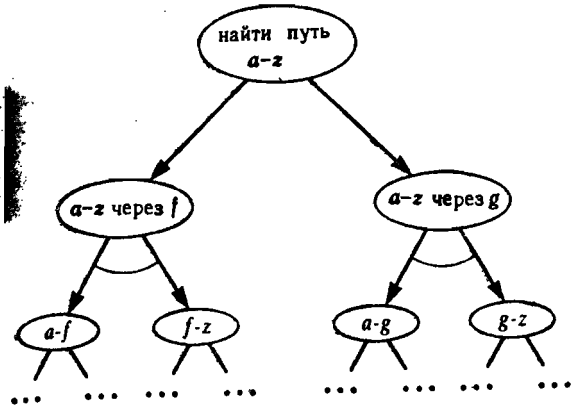


Рис. 13.2. И/ИЛИ-представление задачи поиска маршрута рис. 13.1. Вершины соответствуют задачам или подзадачам, полукруглые дуги означают, что все (точнее, обе) подзадачи должны быть решены.

Теперь каждую из этих двух альтернативных задач можно, в свою очередь, разбить следующим образом:

(1) Для того, чтобы найти путь из a в z через f, необходимо:

1.1 найти путь из a в f и
1.2 найти путь из f в z.

(2) Для того, чтобы найти путь из a в z через g, необходимо:

2.1 найти путь из a в g и
2.2 найти путь из g в z.

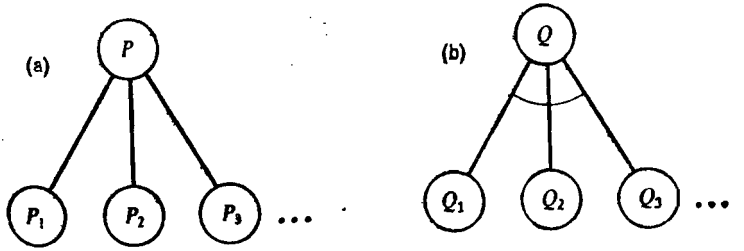


Рис. 13.3. (а) Решить P – это значит решить P_1 или P_2 или ... (б) Решить Q – это значит решить все: Q_1 и Q_2 и ... .

Итак, мы имеем две главных альтернативы для решения исходной задачи: (1) путь через *f* или (2) путь через *g*. Далее, каждую из этих альтернатив можно *разбить на подзадачи* (1.1 и 1.2 или 2.1 и 2.2 соответственно). Здесь важно то обстоятельство, что каждую из подзадач в обоих альтернативах можно решать независимо от другой. Полученное разбиение исходной задачи можно изобразить в форме *И/ИЛИ-графа* (рис.13.2). Обратите внимание на полукруглые дуги, которые указывают на отношение И между соответствующими подзадачами. Граф, показанный на рис. 13.2 – это всего лишь верхняя часть всего И/ИЛИ-дерева. Дальнейшее разбиение подзадач можно было бы строить на основе введения дополнительных промежуточных городов.

Какие вершины И/ИЛИ-графа являются целевыми? Целевые вершины – это тривиальные, или «примитивные» задачи. В нашем примере такой подзадачей можно было бы считать подзадачу «найти путь из *a* в *c*», поскольку между городами *a* и *c* на карте имеется непосредственная связь.

Рассматривая наш пример, мы ввели ряд важных понятий. И/ИЛИ-граф – это направленный граф, вершины которого соответствуют задачам, а дуги – отношениям между задачами. Между дугами также существуют свои отношения. Это отношения И и ИЛИ, в зависимости от того, должны ли мы решить только одну из задач-преемников или же несколько из них (см. рис. 13.3). В принципе из вершины могут выходить дуги, находящиеся в отношении И вместе с дугами, находящимися в отношении ИЛИ. Тем не менее, мы будем предполагать, что каждая вершина имеет либо только И-преемников, либо только ИЛИ-преемников; дело в том, что в такую форму можно преобразовать любой И/ИЛИ-граф, вводя в него при необходимости вспомогательные ИЛИ-вершины. Вершину, из которой выходят только И-дуги, называют И-вершиной; вершину, из которой выходят только ИЛИ-дуги, – ИЛИ-вершиной.

Когда задача представлялась в форме пространства состояний, ее решением был путь в этом пространстве. Что является решением в случае И/ИЛИ-представления? Решение должно, конечно, включать в себя все подзадачи И-вершины. Следовательно, это уже не путь, а дерево. Такое решающее

дерево T определяется следующим образом:

- исходная задача P – это корень дерева T;
- если P является ИЛИ-вершиной, то в T содержится только один из ее преемников (из И/ИЛИ-графа) вместе со своим собственным решающим деревом;
- если P – это И-вершина, то все ее преемники (из И/ИЛИ-графа) вместе со своими решающими деревьями содержатся в T.

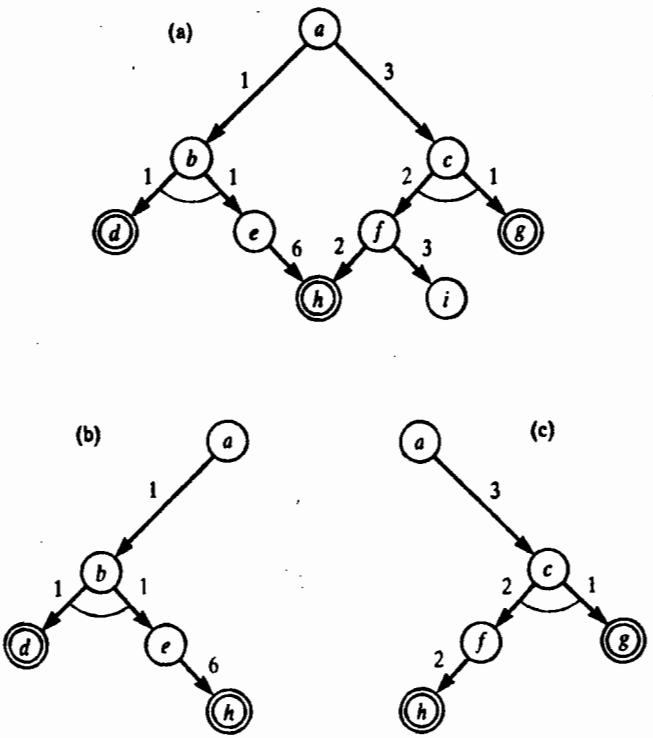


Рис. 13.4. (a) Пример И/ИЛИ-графа: *d*, *g* и *h* – целевые вершины; *a* – исходная задача. (b) и (c) Два решающих дерева, стоимости которых равны 9 и 8 соответственно. Здесь стоимость решающего дерева определена как сумма стоимостей всех входящих в него дуг.

Иллюстрацией к этому определению может служить рис. 13.4. Используя стоимости, мы можем формулировать критерии оптимальности решения. Например, можно определить стоимость решающего графа как сумму стоимостей всех входящих в него дуг. Тогда, поскольку обычно мы заинтересованы в минимизации стоимости, мы отдадим предпочтение решающему графу, изображенному на рис. 13.4(с).

Однако мы не обязательно должны измерять степень оптимальности решения, базируясь на стоимостях дуг. Иногда более естественным окажется приписывать стоимость не дугам, а вершинам, или же и тем, и другим одновременно.

Подведем итоги:

- И/ИЛИ-представление основано на философии сведения задач к подзадачам.
- Вершины И/ИЛИ-графа соответствуют задачам; связи между вершинами – отношениям между задачами.
- Вершина, из которой выходят ИЛИ-связи, называется ИЛИ-вершиной. Для того, чтобы решить соответствующую задачу, нужно решить одну из ее задач-преемников.
- Вершина, из которой выходят И-связи, называется И-вершиной. Для того, чтобы решить соответствующую задачу, нужно решить все ее задачи-преемники.
- При заданном И/ИЛИ-графе конкретная задача специфицируется заданием
  стартовой вершины и
  целевого условия для распознавания
  целевых вершин.
- *Целевые вершины* (или «терминальные вершины») соответствуют тривиальным (или «примитивным») задачам.
- Решение представляется в виде *решающего графа* – подграфа всего И/ИЛИ-графа.
- Представление задач в форме пространства состояний можно рассматривать как специальный частный случай И/ИЛИ-представления, когда все вершины И/ИЛИ-графа являются ИЛИ-вершинами.

- И/ИЛИ-представление имеет преимущество в том случае, когда вершинами, находящимися в отношении И, представлены подзадачи, которые можно решать независимо друг от друга. Критерий независимости можно несколько ослабить, а именно потребовать, чтобы существовал такой порядок решения И-задач, при котором решение более «ранних» подзадач не разрушалось бы при решении более «поздних» подзадач.
- Дугам или вершинам, или и тем, и другим можно приписать стоимости с целью получить возможность сформулировать критерий оптимальности решения.

13.2. Примеры И/ИЛИ—представления задач

13.2.1. И/ИЛИ—представление задачи поиска маршрута

Для задачи отыскания кратчайшего маршрута (рис. 13.1) И/ИЛИ-граф вместе с функцией стоимости можно определить следующим образом:

- ИЛИ-вершины представляются в форме X–Z, что означает: найти кратчайший путь из X в Z.
- И-вершины имеют вид

  X–Z через Y

  что означает: найти кратчайший путь из X в Z, проходящий через Y.
- Вершина X–Z является целевой вершиной (примитивной задачей), если на карте существует непосредственная связь между X и Z.

- Стоимость каждой целевой вершины X–Z равна расстоянию, которое необходимо преодолеть по дороге, соединяющей X с Z.
- Стоимость всех остальных (нетерминальных) вершин равна 0.

Стоимость решающего графа равна сумме стоимостей всех его вершин (в нашем случае это просто сумма стоимостей всех терминальных вершин). В задаче рис. 13.1 стартовая вершина – это *a–z*. На рис.

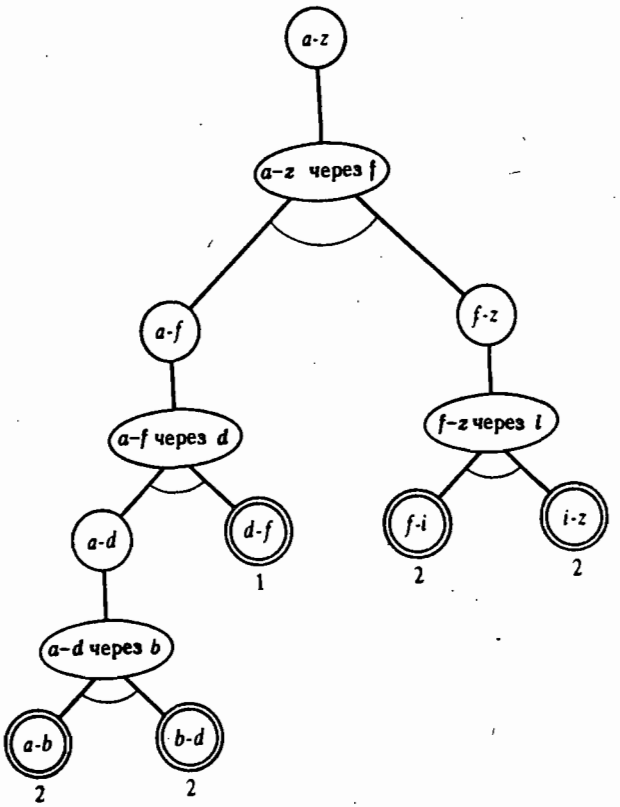


Рис. 13.5. Решающее дерево минимальной стоимости для задачи поиска маршрута рис. 13.1, сформулированной в терминах И/ИЛИ-графа.

13.5 показан решающий граф, имеющий стоимость 9. Это дерево соответствует пути [a,b,d,f,i,z], который можно построить, если пройти по всем листьям решающего дерева слева направо.

13.2.2. Задача о ханойской башне

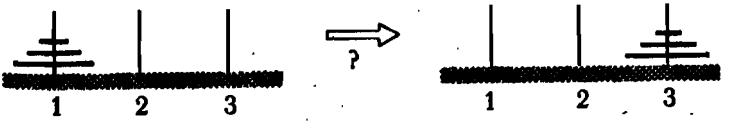
Задача о ханойской башне (рис. 13.6) – это еще один классический пример эффективного применения метода разбиения задачи на подзадачи и построения И/ИЛИ-графа. Для простоты мы рассмотрим упрощенную версию этой задачи, когда в ней участвует только три диска:

> Имеется три колышка 1, 2 и 3 и три диска *a*, *b* и *c* (*a* – наименьший из них, а *c* – наибольший). Первоначально все диски находятся на колышке 1. Задача состоит в том, чтобы переложить все диски на колышек 3. На каждом шагу можно перекладывать только один диск, причем никогда нельзя помещать больший диск на меньший.

Эту задачу можно рассматривать как задачу достижения следующих трех целей:

(1) Диск *a* – на колышек 3.
(2) Диск *b* – на колышек 3.
(3) Диск *c* – на колышек 3.

Беда в том, что эти цели не независимы. Например, можно сразу переложить диск *a* на колышек 3, и первая цель будет достигнута. Но тогда две другие цели станут недостижимыми (если только мы не отменим первое наше действие). К счастью, существует такой удобный порядок достижения этих целей, из которого можно легко вывести искомое решение.



Порядок этот можно установить при помощи следующего рассуждения: самая трудная цель – это цель 3 (диск *c* – на колышек 3), потому что на диск *c* наложено больше всего ограничений. В подобных ситуациях часто срабатывает хорошая идея: пытаться достичь первой самую трудную цель. Этот принцип основан на следующей логике: поскольку другие цели достигнуть легче (на них меньше ограничений), можно надеяться на то, что их достижение возможно без отмены действий на достижение самой трудной цели.

Применительно к нашей задаче это означает, что необходимо придерживаться следующей стратегии:

Первой достигнуть цель «диск *c* – на колышек 3», а затем – все остальные.

Но первая цель не может быть достигнута сразу, так как в начальной ситуации диск *c* двигать нельзя. Следовательно, сначала мы должны подготовить этот ход, и наша стратегия принимает такой вид

(1) Обеспечить возможность перемещения диска *c* с 1 на 3.
(2) Переложить *c* с 1 на 3.
(3) Достигнуть остальные цели (*a* на 3 и *b* на 3).

Переложить *c* с 1 на 3 возможно только в том случае, если диск *a* и *b* оба надеты на колышек 2. Таким образом наша исходная задача перемещения *a*, *b* и *c* с 1 на 3 сводится к следующим трем подзадачам:

Для того, чтобы переложить *a*, *b* и *c* с 1 на 3, необходимо

(1) переложить *a* и *b* с 1 на 2, *и*
(2) переложить *c* с 1 на 3, *и*
(1) переложить *a* и *b* с 2 на 3.

Задача 2 тривиальна (она решается за один шаг). Остальные две подзадачи можно решать независимо от задачи 2, так как диски *a* и *b* можно двигать, не обращая внимание на положение диска *c*. Для решения задач 1 и 3 можно применить тот же самый принцип разбиения (на этот раз диск *b* будет самым «трудным»). В соответствии с этим принципом задача 1 сводится к трем тривиальным подзадачам:

Для того, чтобы переложить *a* и *b* с 1 на 2, необходимо:

(1) переложить a с 1 на 3, $и$
(2) переложить b с 1 на 2, $и$
(1) переложить a с 3 на 2.

13.2.3. Формулировка игровых задач в терминах И ИЛИ—графов

Такие игры, как шахматы или шашки, естественно рассматривать как задачи, представленные И/ИЛИ-графами. Игры такого рода называются играми двух лиц с полной информацией. Будем считать, что существует только два возможных исхода игры: ВЫИГРЫШ и ПРОИГРЫШ. (Об играх с тремя возможными исходами - ВЫИГРЫШ, ПРОИГРЫШ и НИЧЬЯ, можно также говорить, что они имеют только два исхода: ВЫИГРЫШ и НЕВЫИГРЫШ). Так как участники игры ходят по очереди, мы имеем два вида позиций, в зависимости от того, чей ход. Давайте условимся называть участников игры «игрок» и «противник», тогда мы будем иметь следующие два вида позиций: позиция с ходом игрока («позиция игрока») и позиция с ходом противника («позиция противника»). Допустим также, что начальная позиция P – это позиция игрока. Каждый вариант хода игрока в этой позиции приводит к одной из позиций противника Q_1, Q_2, Q_3, ... (см. рис. 13.7). Далее каждый вариант хода противника в позиции Q_1 приводит к одной из позиций игрока R_{11}, R_{12}, ... В И/ИЛИ-дереве, показанном на рис. 13.7, вершины соответствуют позициям, а дуги – возможным ходам. Уровни позиций игрока чередуются в дереве с уровнями позиций противника. Для того, чтобы выиграть в позиции P, нужно найти ход, переводящий P в выигранную позицию Q_i (при некотором i). Таким образом, игрок выигрывает в позиции P, если он выигрывает в Q_1, *или* Q_2, *или* Q_3, *или* ... . Следовательно, P – это ИЛИ-вершина. Для любого i позиция Q_i – это позиция противника, поэтому если в этой позиции выигрывает игрок, то он выигрывает и после каждого вариан-

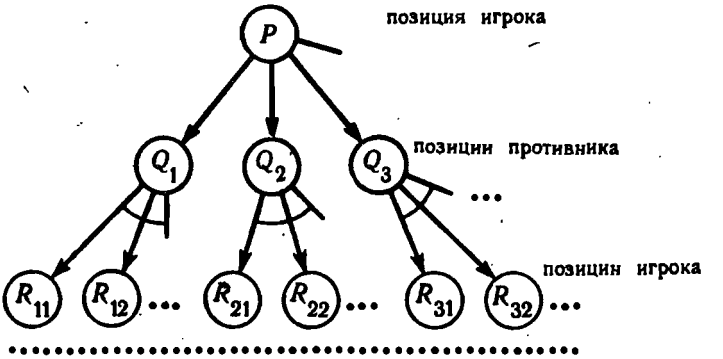


Рис. 13.7. Формулировка игровой задачи для игры двух лиц в форме И/ИЛИ-дерева; участники игры: "игрок" и "противник".

та хода противника. Другими словами, игрок выигрывает в Q_i, если он выигрывает во всех позициях R_{i1} *и* R_{i2} *и* ... . Таким образом, все позиции противника – это И-вершины. Целевые вершины – это позиции, выигранные согласно правилам игры, например позиции, в которых король противника получает мат. Позициям проигранным соответствуют задачи, не имеющие решения. Для того, чтобы решить игровую задачу, мы должны построить решающее дерево, гарантирующее победу игрока независимо от ответов противника. Такое дерево задает полную стратегию достижения выигрыша: для каждого возможного продолжения, выбранного противником, в дереве стратегии есть ответный ход, приводящий к победе.

13.3. Базовые процедуры поиска в И/ИЛИ–графах

В этом разделе нас будет интересовать *какое-нибудь* решение задачи независимо от его стоимости, поэто-

му проигнорируем пока стоимости связей или вершин И/ИЛИ-графа. Простейший способ организовать поиск в И/ИЛИ-графах средствами Пролога – это использовать переборный механизм, заложенный в самой пролог-системе. Оказывается, что это очень просто сделать, потому что процедурный смысл Пролога это и есть не что иное, как поиск в И/ИЛИ-графе. Например, И/ИЛИ-граф рис. 13.4 (без учета стоимостей дуг) можно описать при помощи следующих предложений:

```
a :- b.          % a - ИЛИ-вершина с двумя преемниками
a :- c.          % b и c
b :- d, e.       % b - И-вершина с двумя преемниками d и e
c :- h.
c :- f, g.
f :- h, i.
d. g. h.         % d, g и h - целевые вершины
```

Для того, чтобы узнать, имеет ли эта задача решение, нужно просто спросить:

```
?- a.
```

Получив этот вопрос, пролог-система произведет поиск в глубину в дереве рис. 13.4 и после того, как пройдет через все вершины подграфа, соответствующего решающему дереву рис. 13.4(b), ответит «да».

Преимущество такого метода программирования И/ИЛИ-поиска состоит в его простоте. Но есть и недостатки:

- Мы получаем ответ «да» или «нет», но не получаем решающее дерево. Можно было бы восстановить решающее дерево при помощи трассировки программы, но такой способ неудобен, да его и недостаточно, если мы хотим иметь возможность явно обратиться к решающему дереву как к объекту программы.

- В эту программу трудно вносить добавления, связанные с обработкой стоимостей.

- Если наш И/ИЛИ-граф – это граф общего вида, содержащий циклы, то пролог-система, следуя стратегии в глубину, может войти в бесконечный рекурсивный цикл.

Попробуем постепенно исправить эти недостатки. Сначала определим нашу собственную процедуру поиска в глубину для И/ИЛИ-графов.

Прежде всего мы должны изменить представление И/ИЛИ-графов. С этой целью введем бинарное отношение, изображаемое инфиксным оператором '--->'. Например, вершина *a* с двумя ИЛИ-преемниками будет представлена предложением

```
a ---> или : [b,c].
```

Оба символа '--->' и ':' - инфиксные операторы, которые можно определить как

```
:- op( 600, xfx, --->).
:- op( 500, xfx, :).
```

Весь И/ИЛИ-граф рис.13.4 теперь можно задать при помощи множества предложений

```
a ---> или : [b,c].
b ---> и : [d,e].
c ---> и : [f,g].
e ---> или : [h].
f ---> или : [h,i].

цель(d).   цель(g).   цель(h).
```

Процедуру поиска в глубину в И/ИЛИ-графах можно построить, базируясь на следующих принципах:

Для того, чтобы решить задачу вершины В, необходимо придерживаться приведенных ниже правил:

(1) Если В - целевая вершина, то задача решается тривиальным образом.

(2) Если вершина В имеет ИЛИ-преемников, то нужно решить одну из соответствующих задач-преемников (пробовать решать их одну за другой, пока не будет найдена задача, имеющая решение).

(3) Если вершина В имеет И-преемников, то нужно решить все соответствующие задачи (пробовать решать их одну за другой, пока они не будут решены все).

Если применение этих правил не приводит к решению, считать, что задача не может быть решена.

Соответствующая программа выглядит так:

```
решить( Верш) :-
    цель( Верш).

решить( Верш) :-
    Верш ---> или : Вершины, % Верш - ИЛИ-вершина
    принадлежит( Верш1, Вершины),
                % Выбор преемника Верш1 вершины Верш
решить( Верш1).

решить( Верш) :-
    Верш ---> и : Вершины, % Верш - И-вершина
    решитьвсе( Вершины).
                % Решить все задачи-преемники
решитьвсе([]).

решитьвсе( [Верш|Вершины]) :-
    решить( Верш),
    решитьвсе( Вершины).
```

Здесь **принадлежит** - обычное отношение принадлежности к списку.

Эта программа все еще имеет недостатки:

- она не порождает решающее дерево, и
- она может зацикливаться, если И/ИЛИ-граф имеет соответствующую структуру (циклы).

Программу нетрудно изменить с тем, чтобы она порождала решающее дерево. Необходимо так подправить отношение **решить**, чтобы оно имело два аргумента:

```
решить( Верш, РешДер).
```

Решающее дерево представим следующим образом. Мы имеем три случая:

(1) Если **Верш** - целевая вершина, то соответствующее решающее дерево и есть сама эта вершина.

(2) Если **Верш** - ИЛИ-вершина, то решающее дерево имеет вид

```
Верш ---> Поддерево
```

где **Поддерево** - это решающее дерево для одного из преемников вершины **Верш**.

(3) Если **Верш** - И-вершина, то решающее дерево имеет вид

```
Верш ---> и : Поддеревья
```

где **Поддеревья** - список решающих деревьев для всех преемников вершины **Верш**.

```
% Поиск в глубину для И/ИЛИ-графов
% Процедура решить(Верш,РешДер) находит решающее дерево для
% некоторой вершины в И/ИЛИ-графе

решить( Верш, Верш) :- % Решающее дерево для целевой
    цель( Верш).        % вершины - это сама вершина

решить( Верш, Верш ---> Дер) :-
    Верш ---> или : Вершины,% Верш - ИЛИ-вершина
    принадлежит( Верш1, Вершины),
                % Выбор преемника Верш1 вершины Верш
    решить( Верш1, Дер).

решить( Верш, Верш ---> и : Деревья) :-
    Верш ---> и : Вершины, % Верш - И-вершина
    решитьвсе( Вершины, Деревья).
                % Решить все задачи-преемники

решитьвсе( [], []).

решитьвсе( [Верш|Вершины],[Дер | Деревья]) :-
    решить( Верш, Дер),
    решитьвсе( Вершины, Деревья).

отобр( Дер) :-              % Отобразить решающее дерево
    отобр( Дер, 0), !.      % с отступом 0

отобр( Верш ---> Дер, Н) :-
                % Отобразить решающее дерево с отступом Н
    write( Верш), write( '--->'),
    Н1 is Н + 7,
    отобр( Дер, Н1), !.

отобр( и : [Д], Н) :-
                % Отобразить И-список решающих деревьев
    отобр( Д, Н).

отобр( и : [Д|ДД], Н) :-
                % Отобразить И-список решающих деревьев
    отобр( Д, Н),
    tab( Н),
    отобр( и : ДД, Н), !.

отобр( Верш, Н) :-
    write( Верш), nl.
```

Рис. 13.8. Поиск в глубину для И/ИЛИ-графов. Эта программа может зацикливаться. Процедура **решить** находит решающее дерево, а процедура **отобр** показывает его пользователю. В процедуре **отобр** предполагается, что на вывод вершины тратится только один символ.

Например, при поиске в И/ИЛИ-графе рис. 13.4 первое найденное решение задачи, соответствующей самой верхней вершине *a*, будет иметь следующее представление:

```
a ---> b ---> и : [d,c ---> h]
```

Три формы представления решающего дерева соответствуют трем предложениям отношения **решить**. Поэтому все, что нам нужно сделать для изменения нашей исходной программы **решить**, – это подправить каждое из этих трех предложений, просто добавив в каждое из них решающее дерево в качестве второго аргумента. Измененная программа показана на рис. 13.8. В нее также введена дополнительная процедура **отобр** для отображения решающих деревьев в текстовой форме. Например, решающее дерево рис. 13.4 будет отпечатано процедурой **отобр** в следующем виде:

```
a ---> b ---> d
          e ---> h
```

Программа рис. 13.8 все еще сохраняет склонность к вхождению в бесконечные циклы. Один из простых способов избежать бесконечных циклов – это следить за текущей глубиной поиска и не давать программе заходить за пределы некоторого ограничения по глубине. Это можно сделать, введя в отношение **решить** еще один аргумент:

```
решить( Верш, РешДер, МаксГлуб)
```

Как и раньше, вершиной **Верш** представлена решаемая задача, а **РешДер** – это решение этой задачи, имеющее глубину, не превосходящую **МаксГлуб**. **МаксГлуб** – это допустимая глубина поиска в графе. Если **МаксГлуб** = 0, то двигаться дальше запрещено, если же **МаксГлуб** > 0, то поиск распространяется на преемников вершины **Верш**, причем для них устанавливается меньший предел по глубине, равный **МаксГлуб** – 1. Это дополнение легко ввести в программу рис. 13.8. Например, второе предложение процедуры **решить** примет вид:

```
решить( Верш, Верш ---> Дер, МаксГлуб) :-
    МаксГлуб > 0,
    Верш ---> или : Вершины,% Верш – ИЛИ-вершина
    принадлежит( Верш1, Вершины),
```

```
                % Выбор преемника Верш1 вершины Верш
    Глуб1 is МаксГлуб - 1,   % Новый предел по глубине
    решить( Верш1, Дер, Глуб1).
                % Решить задачу-преемник с меньшим ограничением
```

Нашу процедуру поиска в глубину с ограиичением можно также использовать для имитации поиска в ширину. Идея состоит в следующем: многократно повторять поиск в глубину каждый раз все с большим значением ограничения до тех пор, пока решение не будет найдено. То есть попробовать решить задачу с ограничением по глубине, равным 0, затем - с ограничением 1, затем - 2 и т.д. Получаем следующую программу:

```
имитация_в_ширину( Верш, РешДер) :-
    проба_в_глубину( Верш, РешДер, 0).
% Проба поиска с возрастающим ограничением, начиная с 0

проба_в_глубину( Верш, РешДер, Глуб) :-
    решить( Верш, РешДер, Глуб);
    Глуб1 is Глуб + 1,      % Новый предел по глубине
    проба_в_глубину( Верш, РешДер, Глуб1).
                % Попытка с новым ограничением
```

Недостатком имитации поиска в ширину является то, что при каждом увеличении предела по глубине программа повторио просматривает верхнюю область пространства поиска.

Упражнения

13.1. Закончите составление программы поиска в глубину (с ограничением) для И/ИЛИ-графов, иамеченную в настоящем разделе.

13.2. Определите на Прологе И/ИЛИ-пространство для задачи «ханойская башня» и примеиите к иему процедуры поиска настоящего раздела.

13.3. Рассмотрите какую-нибудь простую детермииированную игру двух лиц с полной информацией и дайте определение ее И/ИЛИ-представления. Используйте программу поиска в И/ИЛИ-графах для построения выигрывающих стратегий в форме И/ИЛИ-деревьев.

13.4. Поиск с предпочтением в И/ИЛИ-графах

13.4.1. Эвристические оценки и алгоритм поиска

Базовые процедуры поиска предыдущего раздела производят систематический и полный просмотр И/ИЛИ-дерева, не руководствуясь при этом какими-либо эвристиками. Для сложных задач подобные процедуры весьма не эффективны из-за большой комбинаторной сложности пространства поиска. В связи с этим возникает необходимость в эвристическом управлении поиском, направленном на уменьшение комбинаторной сложности за счет исключения бесполезных альтернатив. Управление эвристиками, излагаемое в настоящем разделе, будет основано на численных эвристических оценках «трудности» задач, входящих в состав И/ИЛИ-графа. Программу, которую мы составим, можно рассматривать как обобщение программы поиска с предпочтением в пространстве состояний гл. 12.

Начнем с того, что сформулируем критерий оптимальности, основанный на стоимостях дуг И/ИЛИ-графа. Во-первых, мы расширим наше представление И/ИЛИ-графов, дополнив его стоимостями дуг. Например, И/ИЛИ-граф рис. 13.4 можно представить следующими предложениями:

```
a ---> или : [b/1, c/3].
b ---> и : [d/1, e/1].
c ---> и : [f/2, g/1].
e ---> или : [h/6].
f ---> или : [h/2, i/3].

цель( d). цель( g). цель( h).
```

Стоимость решающего дерева мы определим как сумму стоимостей его дуг. Цель оптимизации – найти решающее дерево минимальной стоимости. Как и раньше, иллюстрацией служит рис. 13.4.

Будет полезным определить *стоимость вершины* И/ИЛИ-графа как стоимость оптимального решающего дерева для этой вершины. Стоимость вершины, определенная таким образом, соответствует «трудности» соответствующей задачи.

Мы будем предполагать, что стоимости вершин И/ИЛИ-графа можно оценить (не зная соответствующих решающих деревьев) при помощи эвристической функции h. Эти оценки будут использоваться для управления поиском. Наша программа поиска начнет свою работу со стартовой вершины и, распространяя поиск из уже просмотренных вершин на их преемников, будет постепенно наращивать дерево поиска. Этот процесс будет строить дерево даже в том случае, когда сам И/ИЛИ-граф не является деревом; при этом граф будет разворачиваться в дерево за счет дублирования своих отдельных частей.

Для продолжения поиска будет всегда выбираться «наиболее перспективное» решающее дерево-кандидат. Каким же образом используется функция h для оценки степени перспективности решающего дерева-кандидата или, точнее, вершины-кандидата – корня этого дерева?

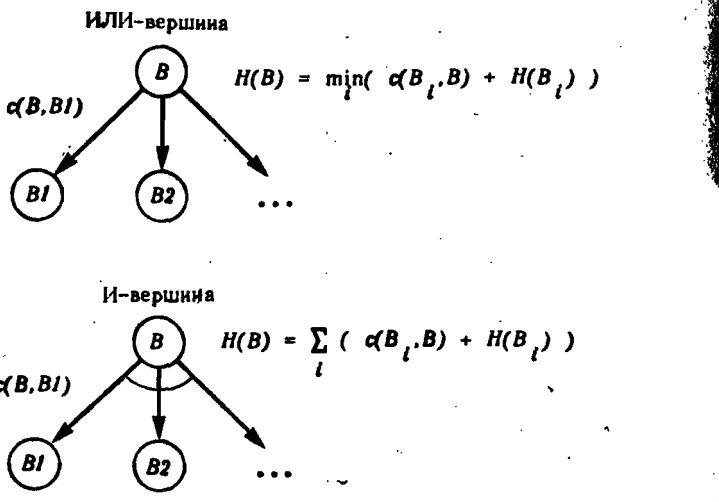


Рис. 13.9. Получение оценки H трудности задач И/ИЛИ-графа.

Обозначим через $H(B)$ оценку трудности вершины B. Для самой верхней вершины текущего дерева поиска $H(B)$ просто совпадает с $h(B)$. С другой стороны, для оценки внутренней вершины дерева поиска нам не обязательно использовать непосредственно значение h, поскольку у нас есть некоторая дополнительная информация об этой вершине: мы знаем ее преемников. Следовательно, как показано на рис. 13.9, мы можем приближенно оценить трудность внутренней ИЛИ-вершины как

$$H(B) = \min_i (c(B,B_i) + H(B_i))$$

где $c(B,B)$ – стоимость дуги, ведущей из B в B_i.

Взятие минимума в этой формуле оправдано тем обстоятельством, что для того, чтобы решить задачу B, нам нужно решить только одну из ее задач-преемников.

Трудность И-вершины B можно приближенно оценить так:

$$H(B) = \sum_i (c(B,B_i) + H(B_i))$$

Будем называть H-оценку внутренней вершины «возвращенной» (backed-up) оценкой.

Более практичной с точки зрения использования в нашей программе поиска является другая величина F, которую можно определить в терминах H следующим образом. Пусть $B1$ – вершина-предшественник вершины B в дереве поиска, причем стоимость дуги, ведущей из $B1$ в B, равна $c(B1,B)$, тогда положим

$$F(B) = c(B1, B) + H(B)$$

Пусть $B1$ – родительская вершина вершины B, а B_1, B_2, ... – ее дочерние вершины, тогда, в соответствии с определениями F и H, имеем

$$F(B) = c(B1, B) + \min_i F(B_i), \quad \text{если } B \text{ – ИЛИ-вершина}$$

$$F(B) = c(B1, B) + \sum_i F(B_i), \quad \text{если } B \text{ – И-вершина}$$

Хотя стартовая вершина A и не имеет предшественника, будем считать, что стоимость ведущей в нее

(виртуальной) дуги равна 0. Если положить h равным 0 для всех вершин И/ИЛИ-дерева, то для любого найденного оптимального решающего дерева окажется, что его стоимость, т.е. сумма стоимостей его дуг, в точности равна $F(A)$.

На любой стадии поиска каждый преемник ИЛИ-вершины соответствует некоторому альтернативному решающему дереву-кандидату. Процесс поиска всегда принимает решение продолжать просмотр того дерева-кандидата, для которого F-оценка минимальна. Вернемся еще раз к рис. 13.4 и посмотрим, как будет вести себя процесс поиска на примере И/ИЛИ-графа, изображенного на этом рисунке. В начале дерево поиска состоит всего из одной вершины – стартовой вершины a, далее дерево постепенно «растет» до тех пор, пока не будет найдено решающее дерево. На рис. 13.10, показан ряд «мгновенных снимков», сделанных в процессе роста дерева поиска. Для простоты мы предположим, что $h = 0$ для всех вершин. Числа, приписанные вершинам на рис. 13.10 – это их F-оценки (разумеется, по мере накопления информации в процессе поиска они изменяются). Ниже даются некоторые пояснительные замечания к рис. 13.10.

После распространения поиска из первоначального дерева (снимок A) получается дерево B. Вершина a – это ИЛИ-вершина, поэтому мы имеем два решающих дерева-кандидата: b и c. Поскольку $F(b) = 1 < 3 = F(c)$, для продолжения поиска выбирается альтернатива b. Насколько далеко может зайти процесс роста поддерева b? Этот процесс может продолжаться до тех пор, пока не произойдет одно из двух событий:

(1) F-оценка вершины b станет больше, чем F-оценка ее конкурента c, или
(2) обнаружится, что найдено решающее дерево.

В связи с этим, начиная просмотр поддерева-кандидата b, мы устанавливаем верхнюю границу для $F(b)$: $F(b) \leq 3 = F(c)$. Сначала порождаются преемники d и e вершины b (снимок C), после чего F-оценка b возрастает до 3. Так как это значение не превосходит верхнюю границу, рост дерева-кандидата с корнем в b продолжается. Вершина d оказывается целевой вершиной, а после распространения поиска из вершины e на один шаг получаем дерево, показанное на снимке D. В этот момент выясняется, что $F(b) = 9 > 3$, и рост дерева b

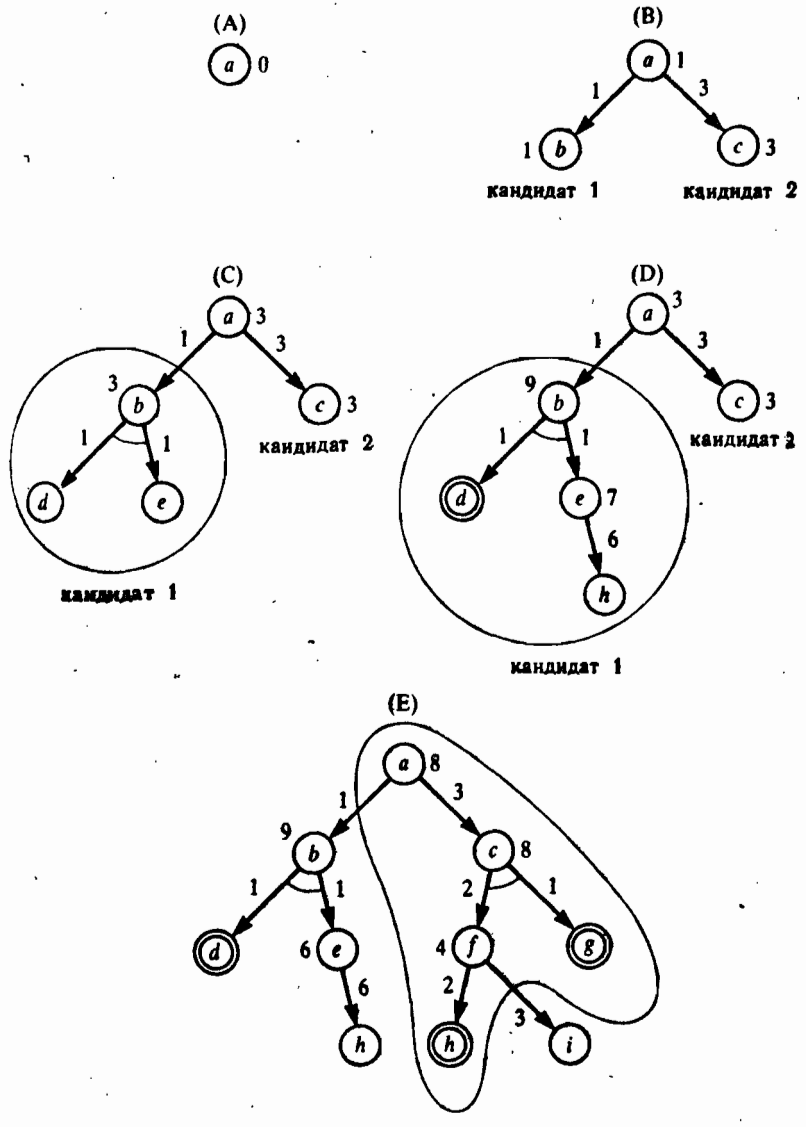


Рис. 13.10. Трассировка процесса поиска с предпочтением в И/ИЛИ-графе ($h = 0$) при решении задачи рис. 13.4.

прекращается. В результате процесс поиска не успевает «осознать», что h – это тоже целевая вершина и что порождено решающее дерево. Вместо этого происходит переключение активности на конкурирующую альтернативу c. Поскольку в этот момент $F(b) = 9$, устанавливается верхняя граница для $F(c)$, равная 9. Дерево-кандидат с корнем c наращивается (с учетом установленного ограничения) до тех пор, пока не возникает ситуация, показанная на снимке Е. Теперь процесс поиска обнаруживает, что найдено решающее дерево (включающее в себя целевые вершины h и g), на чем поиск заканчивается. Заметьте, что в качестве результата процесс поиска выдает наиболее дешевое из двух возможных решающих деревьев, а именно решающее дерево рис. 13.4(с).

13.4.2. Программа поиска

Программа, в которой реализованы идеи предыдущего раздела, показана на рис. 13.12. Прежде, чем мы перейдем к объяснению отдельных деталей этой программы, давайте рассмотрим тот способ представления дерева поиска, который в ней используется.

Существует несколько случаев, как показано на рис. 13.11. Различные формы представления поискового дерева возникают как комбинации следующих возможных вариантов, относящихся к размеру дерева и к его «решающему статусу».

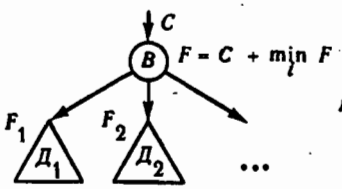
- Размер:
  (1) дерево состоит из одной вершины (листа) или
  (2) оно имеет корень и (непустые) поддеревья.
- Решающий статус:
  (1) обнаружено, что дерево соответствует решению задачи(т. е. является решающим деревом) или
  (2) оно все еще решающее дерево-*кандидат*.

Основной функтор, используемый для представления дерева, указывает, какая из комбинаций этих воз-

Случай1: Дерево поиска, состоящее из одного листа

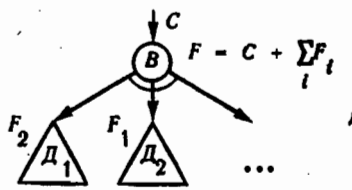
лист(B, F, C)

Случай2: Дерево поиска с ИЛИ-поддеревьями



дер(B,F,C, или:[Д1,Д2,...])

Случай3: Дерево поиска с И-поддеревьями

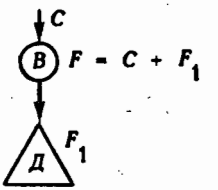


дер(B,F,C, и : [Д1,Д2,...])

Случай4: Лист-решение

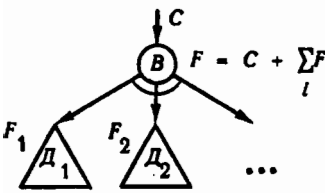
решлист(B, F)

Случай5: Решающее дерево с корнем в ИЛИ-вершине



решдер(В, F, Д)

Случай6: Решающее дерево с корнем в И-вершине



решдер(В, F, и: [Д1,Д2,...])

Рис. 13.11. Представление дерева поиска.

можностей имеется в виду. Это может быть одна из следующих комбинаций:

лист решлист дер решдер

Далее, в представление дерева входят все или некоторые из следующих объектов:

- корневая вершина дерева,
- *F*-оценка дерева,
- стоимость С дуги И/ИЛИ-графа, ведущей в корень дерева,
- список поддеревьев,
- отношение (И или ИЛИ) между поддеревьями.

Список поддеревьев всегда упорядочен по возрастанию *F*-оценок. Поддеревья, являющиеся решающими деревьями, помещаются в конец списка.

Обратимся теперь к программе рис. 13.12. Отношение самого высокого уровня – это

и_или(Верш, РешДер)

где **Верш** – стартовая вершина. Программа строит решающее дерево (если таковое существует), рассчитывая на то, что оно окажется оптимальным решением. Будет ли это решение в действительности самым дешевым, зависит от той функции h, которую использует алгоритм. Существует теорема, в которой говорится о том, как оптимальность решения зависит от h. Эта теорема аналогична теореме о допустимости алгоритма поиска с предпочтением в пространстве состояний (гл. 12). Обозначим через $C(B)$ стоимость оптимального решающего дерева для вершины B. Если для каждой вершины B И/ИЛИ-графа эвристическая оценка $h(B) \leq C(B)$, то гарантируется, что процедура **и_или** найдет оптимальное решение. Если же h не удовлетворяет этому условию, то найденное решение может оказаться субоптимальным. Существует тривиальная эвристическая функция, удовлетворяющая условию оптимальности, а именно $h = 0$ для всех вершин. Ее недостатком является отсутствие эвристической силы.

Основную роль в программе рис. 13.12 играет отношение

расширить(Дер, Предел, Дер1, ЕстьРеш)

Дер и **Предел** – его «входные» аргументы, а **Дер1** и **ЕстьРеш** – «выходные». Аргументы имеют следующий смысл:

Дер – дерево поиска, подлежащее расширению.

Предел – предельное значение F-оценки, при котором еще разрешено наращивать дерево **Дер**.

ЕстьРеш – индикатор, значения которого указывают на то, какой из следующих трех случаев имеет место:

(1) **ЕстьРеш** = да: **Дер** можно «нарастить» (с учетом ограничения **Предел**) таким образом, чтобы образовалось решающее дерево **Дер1**.

(2) **ЕстьРеш** = нет: дерево **Дер** можно расширить до состояния **Дер1**, для которого F-оценка превосходит **Предел**, но прежде чем F-оценка превзошла **Предел**, решающее дерево не было обнаружено.

(3) **ЕстьРеш** = никогда: **Дер** не содержит решения.

В зависимости от случая **Дер1** – это либо решающее

дерево, либо **Дер**, расширенное до момента перехода через **Предел**; если **ЕстьРеш** = **никогда**, то переменная **Дер1** неинициализирована.

Процедура

расширспис(Деревья, Предел, Деревья1, ЕстьРеш)

аналогична процедуре **расширить**. Так же, как и в процедуре **расширить**, **Предел** задает ограничение на рост дерева, а **ЕстьРеш** – это индикатор, указывающий, каков результат расширения («да», «нет» или «никогда»). Первый аргумент – это, на этот раз, *список* деревьев (И-список или ИЛИ-список):

Деревья = или:[Д1,Д2,...] или
Деревья = и : [Д1,Д2,...]

Процедура **расширспис** выбирает из списка **Деревья** наиболее перспективное дерево (исходя из *F*-оценок). Так как деревья в списке упорядочены, таким деревом является первый элемент списка. Наиболее перспективное дерево подвергается расширению с новым ограничением **Предел1**. Значение **Предел1** зависит от **Предел**, а также от других деревьев списка. Если **Деревья** – это ИЛИ-список, то **Предел1** устанавливается как наименьшая из двух величин: **Предел** и *F*-оценка следующего по «качеству» дерева из списка **Деревья**. Если **Деревья** – это И-дерево, то **Предел1** устанавливается равным **Предел** минус сумма *F*-оценок всех остальных деревьев из списка. Значение переменной **Деревья1** зависит от случая, задаваемого индикатором **ЕстьРеш**. Если **ЕстьРеш** = **нет**, то **Деревья1** – это то же самое, что и список **Деревья**, причем наиболее перспективное дерево расширено с учетом ограничения **Предел1**. Если **ЕстьРеш** = **да**, то **Деревья1** – это решение для всего списка **Деревья** (найденное без выхода за границы значения **Предел**). Если **ЕстьРеш** = **никогда**, то переменная **Деревья1** неинициализирована.

Процедура **продолжить**, вызываемая после расширения списка деревьев, решает, что делать дальше, в зависимости от результата срабатывания процедуры **расширить**. Эта процедура либо строит решающее дерево, либо уточняет дерево поиска и продолжает процесс его наращивания, либо выдает сообщение «никогда» в случае, когда было обнаружено, что список деревьев не содержит решения.

```
/*  ПРОГРАММА И/ИЛИ-ПОИСКА С ПРЕДПОЧТЕНИЕМ

Эта программа порождает только одно решение. Гарантируется,
что это решение самое дешевое при условии, что используемая
эвристическая функция является нижней гранью реальной стои-
мости решающих деревьев.

Дерево поиска имеет одну из следующих форм:

дер( Верш, F, C, Поддеревья)        дерево-кандидат
лист( Верш, F, C)                   лист дерева поиска
решдер( Верш, F, Поддеревья)        решающее дерево
решлист( Верш, F)                   лист решающего дерева

C - стоимость дуги, ведущей в Верш
F = C + H, где H - эвристическая оценка оптимального решаю-
                   щего дерева с корнем Верш

Список Поддеревья упорядочен таким образом, что
(1)   решающие поддеревья находятся в конце списка;
(2)   остальные поддеревья расположены в порядке возрастания
      F-оценок

*/

:- op( 500, xfx, :).

:- op( 600, xfx, --->).

и_или( Верш, РешДер) :-
    расширить( лист(Верш,0,0), 9999, РешДер, да).
                    % Предполагается, что 9999 > любой F-оценки

% Процедура расширить( Дер, Предел, НовДер, ЕстьРеш)
% расширяет Дер в пределах ограничения Предел
% и порождает НовДер с "решающим статусом" ЕстьРеш.

% Случай 1: выход за ограничение

расширить( Дер, Предел, Дер, нет) :-
    f( Дер, F), F > Предел, !.          % Выход за ограничение

% В остальных случаях F ≤ Предел
% Случай 2: встретилась целевая вершина

расширить(лист(Верш,F,C),_,решлист(Верш,F),да) :-
    цель( Верш), !.

% Случай 3: порождение преемников листа

расширить(лист(Верш,F,C),Предел,НовДер,ЕстьРеш) :-
```

```
    расшлист( Верш, С, Дер1), !,
    расширить( Дер1, Предел, НовДер, ЕстьРеш);
    ЕстьРеш = никогда, !.          % Нет преемников, тупик

% Случай 4: расширить дерево

расширить(дер(Верш,F,С,Поддеревья),
          Предел,НовДер,ЕстьРеш) :-
  Предел1 is Предел - С,
  расширспис(Поддеревья,Предел1,НовПоддер,ЕстьРеш1),
  продолжить(ЕстьРеш1,Верш,С,НовПоддер,Предел,
                              НовДер,ЕстьРеш).

% расширспис( Деревья, Предел, Деревья1, ЕстьРеш)
% расширяет деревья из заданного списка с учетом
% ограничения Предел и выдает новый список Деревья1
% с "решающим статусом" ЕстьРеш.

расширспис(Деревья,Предел,Деревья1,ЕстьРеш) :-
   выбор(Деревья,Дер,ОстДер,Предел,Предел1),
   расширить( Дер, Предел1, НовДер, ЕстьРеш1),
   собрать(ОстДер,НовДер,ЕстьРеш1,Деревья1,ЕстьРеш).

% "продолжить" решает, что делать после расширения
% списка деревьев

продолжить(да,Верш,С,Поддеревья,_,
                    решдер(Верш,F,Поддеревья),да):-
   оценка( Поддеревья, Н), F is С + Н, !.

продолжить( никогда, _, _, _, _, _, никогда) :- !.

продолжить(нет,Верш,С,Поддеревья,Предел,
                              НовДер,ЕстьРеш) :-
   оценка( Поддеревья, Н), F is С + Н, !,
   расширить(дер(Верш,F,С,Поддеревья),Предел,
                              НовДер,ЕстьРеш

% "собрать" соединяет результат расширения дерева со списком
деревьев

собрать(или : _,Дер,да,Дер,да):-!.%Есть решение ИЛИ-списка

собрать(или : ДД,Дер,нет,или : НовДД,нет) :-
   встав( Дер, ДД, НовДД), !. % Нет решения ИЛИ-списка

собрать( или : [], _, никогда, _, никогда) :- !.
                                % Больше нет кандидатов

собрать( или:ДД, _, никогда, или:ДД, нет) :- !.
                                % Есть еще кандидаты
```

```
собрать( и : ДД,Дер,да,и : [Дер Э ДД],да ) :-
    всереш( ДД), !.                    % Есть решение И-списка

собрать( и : _, _, никогда, _, никогда) :- !.
                                       % Нет решения И-списка

собрать( и : ДД, Дер, ДаНет, и : НовДД, нет) :-
    встав( Дер, ДД, НовДД), !.% Пока нет решения И-списка

% "расшлист" формирует дерево из вершины и ее преемников

расшлист(Верш,С,дер(Верш,F,С,Оп : Поддеревья)) :-
    Верш ---> Оп : Преемники,
    оценнть( Преемники, Поддеревья),
    оценка( Оп : Поддеревья, Н), F is С + Н.

оценить( [], []).

оценить( [Верш/С | ВершиныСтоим], Деревья) :-
    h( Верш, Н), F is С + Н,
    оценить( ВершиныСтоим, Деревья1),
    встав( лист( Верш, F, С), Деревья1, Деревья).

% "всереш" проверяет, все ли деревья в списке "решены"

всереш( []).

всереш( [Дер | Деревья] ) :-
    реш( Дер),
    всереш( Деревья).

реш( решдер( _, _, _) ).

реш( решлист( _, _) ).

f( Дер, F) :-                          % Извлечь F-оценку дерева
    arg( 2, Дер, F), !.                % F - это 2-й аргумент Дер

% встав( Дер, ДД, НовДД) вставляет Дер в список
% деревьев ДД; результат - НовДД

встав( Д, [], [Д] ) :- !.

встав( Д, [Д1 | ДД], [Д, Д1 | ДД] ) :-
    реш( Д1), !.

встав( Д, [Д1 | ДД], [Д1 | ДД1] ) :-
    реш( Д),
    встав(Д, ДД, ДД1), !.

встав( Д, [Д1 | ДД], [Д, Д1 | ДД] ) :-
f( Д, F), f( Д1, F1), F =< F1, !.

встав( Д, [Д1 | ДД], [ Д1 | ДД1] ) :-
    встав( Д, ДД, ДД1).
```

```
% "оценка" находит "возвращенную" F-оценку И/ИЛИ-списка деревьев
оценка( или :[Дер | _], F) :-
                % Первое дерево ИЛИ-списка - наилучшее
    f( Дер, F), !.

оценка( и :[], 0) :- !.

оценка( и : [Дер1 | ДД], F) :-
    f( Дер1, F1),
    оценка( и : ДД, F2),
    F is F1 + F2, !.

оценка( Дер, F) :-
    f( Дер, F).

% Отношение выбор(Деревья,Лучшее,Остальные,Предел,Предел1):
% Остальные - И/ИЛИ-список Деревья без его "лучшего" дерева
% Лучшее; Предел - ограничение для списка Деревья, Предел1 -
% ограничение для дерева Лучшее

выбор(Оп : [Дер],Дер,Оп : [],Предел,Предел) :- !.
                                % Только один кандидат

выбор(Оп : [Дер|ДД],Дер,Оп : ДД,Предел,Предел1) :-
    оценка( Оп : ДД, F),
    ( Оп = или, !, мин( Предел, F,Предел1);
      Оп = и, Предел1 is Предел - F).

мин( A, B, A) :- A < B, !.

мин( A, B, B).
```

Рис. 13.12. Программа поиска с предпочтением в И/ИЛИ-графе.

Еще одна процедура

собрать(ОстДер,НовДер,ЕстьРеш1,НовДеревья,ЕстьРеш)

связывает между собой несколько объектов, с которыми работает **расширспис**. **НовДер** – это расширенное дерево, взятое из списка деревьев процедуры **расширспис**, **ОстДер** – остальные, не измененные деревья из этого списка, а **ЕстьРеш1** указывает на «решающий статус» дерева **НовДер**. Процедура **собрать** имеет дело с несколькими случаями в зависимости от значения **ЕстьРеш1**, а также от того, является ли список деревьев И-списком или ИЛИ-списком. Например, предложение

собрать(или : _, Дер, да, Дер, да).

означает: в случае, когда список деревьев – это ИЛИ-список и при только что проведенном расширении получено решающее дерево, считать, что задача, соответствующая всему списку деревьев, также решена, а ее решающее дерево и есть само дерево **Дер**. Остальные случаи легко понять из текста процедуры **собрать**.

Для отображения решающего дерева можно определить процедуру, аналогичную процедуре **отобр** (рис. 13.8). Оставляем это читателю в качестве упражнения.

13.4.3. Пример отношений, определяющих конкретную задачу: поиск маршрута

Давайте теперь сформулируем задачу нахождения маршрута как задачу поиска в И/ИЛИ-графе, причем сделаем это таким образом, чтобы наша формулировка могла бы быть непосредственно использована процедурой и_или рис. 13.12. Мы условимся, что карта дорог будет представлена при помощи отношения

связь(Гор1, Гор2, Р)

означающего, что между городами **Гор1** и **Гор2** существует непосредственная связь, а соответствующее расстояние равно **Р**. Далее, мы допустим, что существует отношение

клпункт(Гор1-Гор2, Гор3)

имеющее следующий смысл: для того, чтобы найти маршрут из **Гор1** в **Гор2**, следует рассмотреть пути, проходящие через **Гор3** (**Гор3** – это «ключевой пункт» между **Гор1** и **Гор2**). Например, на карте рис. 13.1 *f* и *g* – это ключевые пункты между *a* и *z*:

клпункт(a-z, f). **клпункт(a-z, g).**

Мы реализуем следующий принцип построения маршрута:

Для того, чтобы найти маршрут между городами X и Z, необходимо:

(1) если между X и Z имеются ключевые пункты Y1, Y2, ..., то найти один из путей:

- путь из X в Z через Y1, или
- путь из X в Z через Y2, или

...

(2) если между X и Z нет ключевых пунктов, то найти такой соседний с X город Y, что существует маршрут из Y в Z.

Таким образом, мы имеем два вида задач, которые мы будем представлять как

(1)	**X–Z**	найти маршрут из X в Z
(2)	**X–Z через Y**	найти маршрут из X в Z, проходящий через Y

Здесь 'через' – это инфиксный оператор более высокого приоритета, чем '–', и более низкого, чем '--->'. Теперь можно определить соответствующий И/ИЛИ-граф явным образом при помощи следующего фрагмента программы:

```
:- op( 560, xfx, через)

% Правила задачи X–Z, когда между X и Z
% имеются ключевые пункты,
% стоимости всех дуг равны 0

X–Z ---> или : СписокЗадач

:- bagof( (X–Z через Y)/0, клпункт( X–Z, Y),
          СписокЗадач), !.

% Правила для задачи X–Z без ключевых пунктов

X–Z ---> или : СписокЗадач

:- bagof( (Y–Z)/P, связь( X, Y, P), СписокЗадач).

% Сведение задачи типа "через" к подзадачам,
% связанным отношением И

X–Z через Y ---> и : [( X–Y)/0, (Y–Z)/0].

цель( X–X)          % Тривиальная задача: попасть из X в X
```

Функцию h можно определить, например, как расстояние, которое нужно преодолеть при воздушном сообщении между городами.

Упражнение

13.4. Напишите процедуру

отобр2(РешДер)

для отображения решающего дерева, найденного программой **и_или** рис. 13.12. Формат отображения пусть будет аналогичен тому, что применялся в процедуре **отобр** (рис. 13.8), так что процедуру **отобр2** можно получить, внеся в **отобр** изменения, связанные с другим представлением деревьев. Другая полезная модификация – заменить в **отобр** цель **write(Верш)** на процедуру, определяемую пользователем

печверш(Верш, Н)

которая выведет **Верш** в удобной для пользователя форме, а также конкретизирует **Н** в соответствии с количеством символов, необходимом для представления **Верш** в этой форме. В дальнейшем **Н** будет использоваться как величина отступа для поддеревьев.

Резюме

- И/ИЛИ-граф – это формальный аппарат для представления задач. Такое представление является наиболее естественным и удобным для задач, которые разбиваются на независимые подзадачи. Примером могут служить игры.
- Вершины И/ИЛИ-графа бывают двух типов: И-вершины и ИЛИ-вершины.
- Конкретная задача определяется стартовой вершиной и целевым условием. Решение задачи представляется решающим деревом.
- Для моделирования оптимизационных задач в И/ИЛИ-граф можно ввести стоимости дуг и вершин.
- Процесс решения задачи, представленной И/ИЛИ-графом, включает в себя поиск в графе. Стратегия поиска в глубину предусматривает систематический просмотр графа и легко про-

граммируется. Однако эта стратегия может привести к неэффективности из-за комбинаторного взрыва.

- Для оценки трудности задач можно применить эвристики, а для управления поиском – принцип эвристического поиска с предпочтением. Эта стратегия более трудна в реализации.
- В данной главе были разработаны прологовские программы для поиска в глубину и поиска с предпочтением в И/ИЛИ-графах.
- Были введены следующие понятия:

  И/ИЛИ-графы
  И-дуги, ИЛИ-дуги
  И-вершины, ИЛИ-вершины
  решающий путь, решающее дерево
  стоимость дуг и вершин
  эвристические оценки в И/ИЛИ-графах
  «возвращенные» оценки
  поиск в глубину в И/ИЛИ-графах
  поиск с предпочтением в И/ИЛИ-графах

Литература

И/ИЛИ-графы и связанные с ними алгоритмы поиска являются частью классических механизмов искусственного интеллекта для решения задач и реализации машинных игр. Ранним примером прикладной задачи, использующей эти методы, может служить программа символического интегрирования (Slagle 1963). И/ИЛИ-поиск используется в самой пролог-системе. Общее описание И/ИЛИ-графов и алгоритма можно найти в учебниках по искусственному интеллекту (Nilsson 1971; Nilsson 1980). Наша программа поиска с предпочтением – это один из вариантов алгоритма, известного под названием АО*. Формальные свойства АО*-алгоритма (включая его допустимость) изучались несколькими авторами. Подробный обзор полученных результатов можно найти в книге Pearl (1984).

Nilsson N.J. (1971). *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill.

Nilsson N.J. (198 0). *Principles of Artificial Intelligence*. Tioga; also Springer-Verlag.

Pearl J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Slagle J.R. (1963). A heuristic program that solves symbolic integration problems in freshman calculus. In: *Computers and Thought* (E. Feigenbaum, J. Feldman, eds.). McGraw-Hill.

14 ЭКСПЕРТНЫЕ СИСТЕМЫ

Экспертная система – это программа, которая ведет себя подобно эксперту в некоторой проблемной области. Она должна иметь способность к *объяснению* своих решений и тех рассуждений, на основе которых эти решения были приняты. Часто от экспертной системы требуют, чтобы она могла работать с неточной и неполной информацией.

Для того, чтобы построить экспертную систему, мы должны создать механизмы, обеспечивающие выполнение следующих функций: решение задач, взаимодействие с пользователем и работа в условиях неопределенности. В данной главе мы разработаем и реализуем основные идеи построения экспертных систем.

14.1. Функции, выполняемые экспертной системой

Экспертная система – это программа, которая ведет себя подобно эксперту в некоторой, обычно узкой, прикладной области. Типичные применения экспертных систем включают в себя такие задачи, как медицинская диагностика, локализация неисправностей в оборудовании и интерпретация результатов измерений. Экспертные системы должны решать задачи, требующие для своего решения экспертных знаний в некоторой конкретной области. В той или иной форме экспертные системы должны обладать этими знаниями. Поэтому их также называют *системами, основанными на знаниях*. Однако не всякую систему, основанную на знаниях, можно рассматривать как экспертную. Экспертная система должна также уметь каким-то обра-

зом *объяснять* свое поведенне н свон решения пользователю, так же, как это делает эксперт-человек. Это особенно необходнмо в областях, для которых характерна неопределенность, неточность ннформации (напрнмер, в медицннской диагностике). В этих случаях способность к объясненню нужна для того, чтобы повысить степень доверня пользователя к советам системы, а также для того, чтобы дать возможность пользователю обнаружнть возможный дефект в рассуждениях системы. В связн с этим в экспертных системах следует предусматривать дружественное взаимодействие с пользователем, которое делает для пользователя процесс рассуждения системы «прозрачным».

Часто к экспертным системам предъявляют дополнительное требование – способность нметь дело с неопределенностью н неполнотой. Информация о поставленной задаче может быть неполной илн ненадежной; отношения между объектамн/ предметной области могут быть приближенными. Например, может не быть полной уверенности в наличии у пацнента некоторого симптома илн в том, что данные, полученные при нзмеренин, верны; лекарство *может* стать прнчиной осложнення, хотя *обычно* этого не пронсходнт. Во всех этнх случаях необходимы рассуждения с нспользованием вероятностного подхода.

В самом общем случае для того, чтобы построить экспертную систему, мы должны разработать механизмы выполнения следующнх функций системы:

- *решение задач* с использованнем знаний о конкретной предметной области – возможно, прн этом возникнет необходнмость *иметь дело с неопределенностью*
- *взаимодействие с пользователем*, включая объяснение намереинй и решений системы во время и после окончання процесса решения задачи.

Каждая из этих функций может оказаться очень сложной и зависит от прнкладной области, а также от различных практнческнх требований. В процессе разработки и реализации могут вознпкать разнообразные трудные проблемы. В данной главе мы ограничимся наметками основных идей, подлежащих в дальнейшем детализации н усовершенствованию.

14.2. Грубая структура экспертной системы

При разработке экспертной системы принято делить ее на три основных модуля, как показано на рис. 14.1:

(1) база знаний,
(2) машина логического вывода,
(3) интерфейс с пользователем.

База знаний содержит знания, относящиеся к конкретной прикладной области, в том числе отдельные факты, правила, описывающие отношения или явления, а также, возможно, методы, эвристики и различные идеи, относящиеся к решению задач в этой прикладной области. *Машина логического вывода* умеет активно использовать информацию, содержащуюся в базе знаний. *Интерфейс с пользователем* отвечает за бесперебойный обмен информацией между пользователем и системой; он также дает пользователю возможность наблюдать за процессом решения задач, протекающим в машине логического вывода. Принято рассматривать машину вывода и интерфейс как один крупный модуль, обычно называемый *оболочкой экспертной системы*, или, для краткости, просто *оболочкой*.

В описанной выше структуре собственно знания отделены от алгоритмов, использующих эти знания. Такое разделение удобно по следующим соображениям. База знаний, очевидно, зависит от конкретного при-

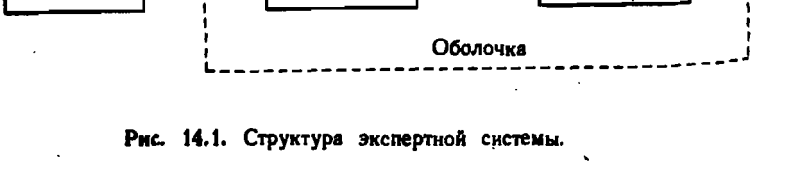


Рис. 14.1. Структура экспертной системы.

ложения. С другой стороны, оболочка, по крайней мере в принципе, независима от приложений. Таким образом, разумный способ разработки экспертной системы для нескольких приложений сводится к созданию универсальной оболочки, после чего для каждого приложения достаточно подключить к системе новую базу знаний. Разумеется, все эти базы знаний должны удовлетворять одному и тому же формализму, который оболочка «понимает». Практический опыт показывает, что для сложных экспертных систем наш сценарий с одной оболочкой и многими базами знаний работает не так гладко, как бы этого хотелось, за исключением тех случаев, когда прикладные области очень близки. Тем не менее даже если переход от одной прикладной области к другой требует модификации оболочки, то по крайней мере основные принципы ее построения обычно удается сохранить.

В этой главе мы намерены разработать относительно простую оболочку, при помощи которой, несмотря на ее простоту, мы сможем проиллюстрировать основные идеи и методы в области экспертных систем. Мы будем придерживаться следующего плана:

(1) Выбрать формальный аппарат для представления знаний.

(2) Разработать механизм логического вывода, соответствующий этому формализму.

(3) Добавить средства взаимодействия с пользователем.

(4) Обеспечить возможность работы в условиях неопределенности.

14.3. Правила типа «если—то» для представления знаний

В качестве кандидата на использование в экспертной системе можно рассматривать, в принципе, любой непротиворечивый формализм, в рамках которого можно описывать знания о некоторой проблемной области. Однако самым популярным формальным языком

представления знаний является язык правил типа «если-то» (или кратко: «если-то»-правил), называемых также *продукциями*. Каждое такое правило есть, вообще говоря, некоторое условное утверждение, но возможны и различные другие интерпретации. Вот примеры:

- *если* предварительное условие P *то* заключение (вывод) C
- *если* ситуация S *то* действие A
- *если* выполнены условия C1 и C2 *то* не выполнено условие C

«Если-то»-правила обычно оказываются весьма естественным выразительным средством представления знаний. Кроме того, они обладают следующими привлекательными свойствами:

- *Модульность*: каждое правило описывает небольшой, относительно независимый фрагмент знаний.
- Возможность *инкрементного наращивания*: добавление новых правил в базу знаний происходит относительно независимо от других правил.
- *Удобство модификации* (как следствие модульности): старые правила можно изменять и заменять на новые относительно независимо от других правил.
- Применение правил способствует *прозрачности* системы.

Последнее свойство – это важное, отличительное свойство экспертных систем. Под прозрачностью мы понимаем способность системы к объяснению принятых решений и полученных результатов. Применение «если-то»-правил облегчает получение ответов на следующие основные типы вопросов пользователя:

(1) Вопросы типа «как»: *Как* вы пришли к этому выводу?

(2) Вопросы типа «почему»: *Почему* вас интересует эта информация?

Механизмы, основанные на «если-то»-правилах, для формирования ответов на подобные вопросы мы обсудим позже.

если
1 тип инфекции – это первичная бактериемия и
2 материал для посева был отобран стерильно, и
3 предполагаемые ворота инфекции – желудочно-кишечный тракт
то
имеются веские аргументы (0.7) за то,
что инфекционный агент является бактерией

Рис. 14.2. "Если-то"-правило медицинской консультативной системы MYCIN (Shortliffe, 1976). Параметр 0.7 показывает степень доверия этому правилу.

«Если-то»-правила часто применяют для определения логических отношений между понятиями предметной области. Про чисто логические отношения можно сказать, что они принадлежат к «категорическим знаниям», «категорическим» – потому, что соответствующие утверждения всегда, абсолютно верны. Однако в некоторых предметных областях, таких, как медицинская диагностика, преобладают «мягкие» или вероятностные знания. Эти знания являются «мягкими» в том смысле, что говорить об их применимости к любым практическим ситуациям можно только до некоторой степени («часто, но не всегда»). В таких случаях используют модифицированные «если-то»-правила, дополняя их логическую интерпретацию вероятностной оценкой. Например:

если условие A *то* заключение B *с уверенностью* F

Рис. 14.2, 14.3 и 14.4 дают представление о разнообразии способов, которыми знания могут быть выражены при помощи «если-то»-правил. На этих рисунках приведены примеры правил из трех различных систем, основанных на знаниях: медицинской консультативной системы MYCIN, системы AL/X для диагностики неисправностей в оборудовании и системы AL3 для решения шахматных задач.

Вообще говоря, если вы хотите разработать серьезную экспертную систему для некоторой выбранной вами предметной области, вы должны провести консультации с экспертами в этой области и многое узнать о ней сами. Достигнуть определенного поня-

мания предметной области после общения с экспертами и чтения литературы, а затем облечь это понимание в форму представления знаний в рамках выбранного формального языка – это искусство, называемое *инженерией знаний*. Как правило, это сложная задача, требующая больших усилий, чего мы не можем себе позволить в данной книге. Но какая-нибудь предметная область и какая-нибудь база данных нам необходимы в качестве материала для экспериментов. С практической точки зрения нам для этой цели вполне подойдет «игрушечная» база знаний. На рис. 14.5 показана часть такой базы знаний. Она состоит из простых правил, помогающих идентифицировать животных по их основным признаками в предположении, что задача идентификации ограничена только небольшим числом разных животных.

Правила, содержащиеся в базе знаний, имеют вид

ИмяПравила : если Условие то Заключение

где **Заключение** – это простое утверждение, а

если
давление в V-01 достигло уровня открытия выпускного клапана
то
выпускной клапан в V-01 открылся
[N=0.005, S=400]
если
давление в V-01 не достигло уровня открытия выпускного клапана и выпускной клапан в V-01 открылся
то
преждевременное открытие выпускного клапана (сместилась установка порогового давления)
[N=0.001, S=2000]

Рис. 14.3. Два правила из демонстрационной базы знаний системы AL/X для диагностики неисправностей (Reiter 1980). N и S – величины "необходимости" и "достаточности", детально описанные в разд. 14.7. Величина S указывает степень, с которой условие влечет за собой заключение (вывод). Величина N указывает, до какой степени истинность условия необходима для того, чтобы заключение было истинным.

если
1 существует гипотеза *H*, что план *P* ведет к успеху, и
2 существуют две гипотезы
H1, что план *P1* опровергает план *P*, и
H2, что план *P2* опровергает план *P*, и
3 имеют место факты: гипотеза *H1 ложна* и гипотеза *H2 ложна*

то
1 породить гипотезу *H3*, что составной план *"P1 или P2"* опровергает план *P*, и
2 породить факт: *из H3 следует не(H)*

Рис. 14.4. Правило уточнения плана из системы AL3 для решения шахматных задач (Bratko 1982).

Условие – это набор простых утверждений, соединенных между собой операторами **и** и **или**. Мы также разрешим в части условия использовать оператор **не**, хотя и с некоторыми оговорками. При надлежащем прологовском определении этих операторов (как это сделано на рис. 14.5) правила станут синтаксически верными предложениями Пролога. Заметим, что оператор **и** связывает операнды сильнее, чем **или**, что соответствует обычным соглашениям.

```
% Небольшая база знаний для идентификации животных
:- op( 100, xfx,[имеет, 'кормит детенышей',
                 'не может',ест,откладывает,это]).
:- op( 100, xf, [плавает, летает, хорошо]).

прав1:   если
             Животное имеет шерсть
             или
             Животное 'кормит детенышей' молоком
         то
             Животное это млекопитающее.
прав2:   если
             Животное имеет перья
             или
             Животное летает и
             Животное откладывает яйца
         то
             Животное это птица.
```

```
прав3:   если
             Животное это млекопитающее и
             ( Животное ест мясо
               или
               Животное имеет 'острые зубы' и
               Животное имеет когти и
               Животное имеет
                'глаза, направленные вперед' )
         то
             Животное это хищник.
прав4:   если
             Животное это хищник и
             Животное имеет
                'рыжевато-коричневый цвет' и
             Животное имеет 'темные пятна'
         то
             Животное это гепард.
прав5:   если
             Животное это хищник и
             Животное имеет
                'рыжевато-коричневый цвет' и
             Животное имеет 'черные полосы'
         то
             Животное это тигр.
прав6:   если
             Животное это птица и
             Животное 'не может' летать и
             Животное плавает
         то
             Животное это пингвин.
прав7:   если
             Животное это птица и
             Животное летает хорошо
         то
             Животное это альбатрос.

факт: X это животное :-
    принадлежит(X,[гепард,тигр,пингвин,альбатрос]).

можно_спросить( _ 'кормит детенышей'_,
    'Животное' 'кормит детенышей' 'Чем').

можно_спросить( _ летает, 'Животное' летает).

можно_спросить( _ откладывает яйца,
    'Животное' откладывает яйца).

можно_спросить( _ ест _, 'Животное' ест 'Что').
```

```
можно_спросить(_ имеет _,'Животное' имеет 'Нечто').

можно_спросить( _ 'не может' _,
    'Животное' 'не может' 'Что делать').

можно_спросить( _ плавает, 'Животное' плавает).

можно_спросить( _ летает хорошо,
    'Животное' летает хорошо).
```

Рис. 14.5. Простая база знаний для идентификации животных. Заимствовано из Winston (1984). Отношение "можно_спросить" определяет вопросы, которые можно задавать пользователю. Операторы **если**, **то**, **и**, **или** определены на рис. 14.10.

Рассмотрим еще одну небольшую базу знаний, которая может помочь локализовать неисправности в простой электрической схеме, состоящей из электрических приборов и предохранителей. Электрическая схема показана на рис. 14.6. Вот одно из возможных правил:

если
    лампа1 включена *и*
    лампа1 не работает *и*
    предохранитель1 заведомо цел
то
    лампа1 заведомо неисправна.

Вот другой пример правила:

если
    радиатор работает
то
    предохранитель1 заведомо цел.

Эти два правила опираются на некоторые факты (относящиеся к нашей *конкретной* схеме), а именно что *лампа1* соединена с *предохранитель1* и что *лампа1* и *радиатор* имеют общий предохранитель. Для другой схемы нам понадобится еще один набор правил. Поэтому было бы лучше сформулировать правила в более общем виде (используя прологовские переменные) так, чтобы они были применимы к любой схеме, а затем уже дополнять их информацией о конкретной схеме. Например, вот одно из полезных правил: если прибор включен, но не работает, а соответствующий предохранитель цел, то прибор неисправен. На наш

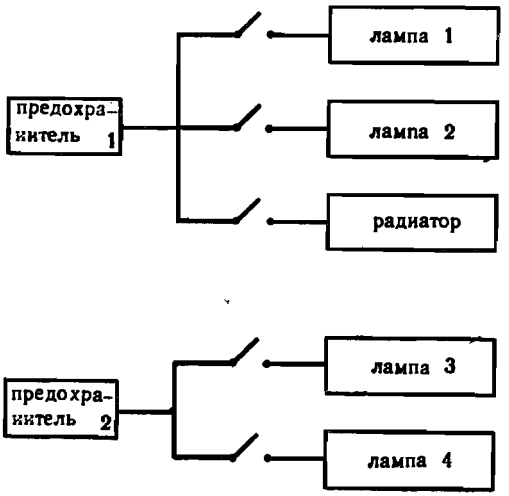


Рис. 14.6. Соединения между предохранителями и приборами в простой электрической схеме.

формальный язык это транслируется так:

```
правило_поломки:
        если
            Прибор включен и
            не (Прибор работает) и
            Прибор соединен с Предохранитель и
            Предохранитель заведомо цел
        то
            Прибор заведомо неисправен.
```

База знаний такого рода показана на рис. 14.7.

Упражнения

14.1. Рассмотрите «если-то»-правила рис. 14.2–14.4 и транслируйте их в нашу систему обозначений для правил. Предложите расширение нотации, чтобы, при необходимости, можно было работать с оценками уверенности.

```
% Небольшая база знаний для локализации неисправностей в
% электрической схеме

% Если прибор включен, но не работает, и предохранитель цел,
% то прибор неисправен.

правило_поломки:
                если
                    вкл( Прибор) и
                    прибор( Прибор) и
                    не работает( Прибор) и
                    соед( Прибор, Предохр) и
                    доказано( цел( Предохр) )
                то
                    доказано( неиспр( Прибор) ).

% Если устройство работает, то его предохранитель цел

правило_цел_предохр:
                если
                    соед( Прибор, Предохр) и
                    работает( Прибор)
                то
                    доказано( цел( Предохр) ).

% Если два различных прибора подключены к одному и тому же
% предохранителю, оба включены и не работают, то предохранитель
% сгорел.
% ЗАМЕЧАНИЕ: предполагается, что из двух приборов неисправных
% не более одного!

правило_предохр:
            если
                соед( Прибор1, Предохр) и
                вкл( Прибор1) и
                не работает( Прибор1) и
                общ_предохр( Прибор2, Прибор1) и
                вкл( Прибор2) и
                не работает( Прибор2)
            то
                доказано( сгорел( Предохр) ).

правило_общ_предохр:
            если
                соед( Прибор1, Предохр) и
                соед( Прибор2, Предохр) и
                различны( Прибор1, Прибор2)
            то
```

```
                общ_предохр( Прибор1, Прибор2).
факт: различны( X, Y) :- not (X=Y).

факт: прибор( радиатор).
факт: прибор( лампа1).
факт: прибор( лампа2).
факт: прибор( лампа3).
факт: прибор( лампа4).

факт: соед( лампа1, предохр1).
факт: соед( лампа2, предохр1).
факт: соед( радиатор, предохр1).
факт: соед( лампа3, предохр2).
факт: соед( лампа4, предохр2).

можно_спросить( вкл( П), вкл( 'Прибор') ).

можно_спросить(работает( П),работает('Прибор')).
```

Рис. 14.7. База знаний для локализации неисправностей в схеме, показанной на рис. 14.6.

14.2. Придумайте какую-нибудь задачу принятия решений и сформулируйте соответствующие знания в форме «если-то»-правил. Можете рассмотреть, например, планирование отпуска, предсказание погоды, простой медицинский диагноз и лечение и т. п.

14.4. Разработка оболочки

Если мы посмотрим на правила наших двух маленьких баз знаний рис. 14.5 и 14.7, мы сразу увидим, что они по своему смыслу эквивалентны правилам Пролога. Однако, с точки зрения синтаксиса Пролога, эти правила в том виде, как они написаны, соответствуют всего лишь фактам. Для того, чтобы заставить их работать, самое простое, что может прийти в голову, это переписать их в виде настоящих прологовских правил. Например:

```
Животное это млекопитающее :-
    Животное имеет шерсть;
    Животное 'кормит детенышей' молоком.

Животное это хищник :-
    Животное это млекопитающее,
    Животное ест мясо.

...
```

Теперь эта программа сможет подтвердить, что тигр по имени Питер – это действительно тигр, если мы добавим в нее некоторые из свойств Питера (в виде прологовских фактов):

```
питер имеет шерсть.
питер ленив.
питер большой.
питер имеет 'рыжевато-коричневый цвет'.
питер имеет 'черные полосы'.
питер ест мясо.
```

Тогда мы можем спросить:

```
?- питер это тигр.
yes

?- питер это гепард.
no
```

Хотя пролог-система и отвечает на вопросы, используя для этого нашу базу знаний, нельзя сказать, что ее поведение вполне соответствует поведению эксперта. Это происходит по крайней мере по двум причинам:

(1) Мы не можем попросить систему объяснить свой ответ; например, *как* она установила, что Питер это тигр, и *почему* Питер это *не* гепард.

(2) Прежде, чем задать вопрос, нужно ввести в систему всю необходимую информацию (в виде прологовских фактов). Но тогда пользователь, возможно, введет какую-нибудь лишнюю информацию (как в нашем примере) или же упустит какую-нибудь информацию, имеющую решающее значение. В первом случае будет проделана ненужная работа, а во втором – система будет давать неверные ответы.

Для того, чтобы исправить эти два недостатка, мы нуждаемся в более совершенном способе взаимодействия между пользователем и системой во время и после завершения процесса рассуждений. Поставим себе целью добиться того, чтобы система взаимодействовала с пользователем так, как в следующем примере диалога (ответы пользователя даются полужирным шрифтом, реплики пролог-системы – курсивом):

Пожалуйста, спрашивайте:
питер это тигр.

Это правда: питер имеет шерсть?
да.

Это правда: питер ест мясо?
нет.

Это правда: питер имеет острые зубы?
да.

Это правда: питер имеет когти?
почему.

Чтобы проверить по прав3, что питер это хищник
Чтобы проверить по прав5, что питер это тигр
Это был ваш вопрос

Это правда: питер имеет когти?
да.

Это правда: питер имеет глаза, направленные вперед?
да.

Это правда: питер имеет рыжевато-коричневый цвет?
да.

Это правда: питер имеет черные полосы?
да.

(питер это тигр) это правда
Хотите узнать, как?
да.

```
питер это тигр
 было выведено по прав5 из
  питер это хищник
   было выведено по прав3 из
    питер это млекопитающее
     было выведено по прав1 из
      питер имеет шерсть
       было сказано
     и
     питер имеет острые зубы
      было сказано
     и
     питер имеет когти
      было сказано
     и
     питер имеет глаза, направленные вперед
      было сказано
  и
  питер имеет рыжевато-коричневый цвет
   было сказано
  и
  питер имеет черные полосы
   было сказано
```

Как видно из диалога, система задает пользователю вопросы, касающиеся «примитивной» информации, например:

Это правда: питер ест мясо?

Эту информацию нельзя отыскать в базе знаний или вывести из другой информации. На подобные вопросы пользователь может отвечать двумя способами:

(1) сообщив системе в качестве ответа на вопрос необходимую информацию или
(2) спросив систему, *почему* эта информация необходима.

Последняя из двух возможностей полезна, поскольку она позволяет пользователю заглянуть внутрь системы и увидеть ее текущие намерения. Пользователь спросит «почему» в том случае, когда вопрос системы покажется ему не относящимся к делу либо когда ответ на вопрос системы потребует от него дополнительных усилий. Из объяснений системы пользователь поймет, стоит ли информация, которую запрашивает система, тех дополнительных усилий, которые необ-

ходимо приложить для ее приобретения. Предположим, например, что система спрашивает: «Это животное ест мясо?» Пользователь, не знающий ответа на этот вопрос, поскольку он никогда не видел, как это животное ело что-либо, может решить, что не стоит ждать, пока он застанет животное за едой и убедится, что оно действительно ест мясо.

Для того, чтобы заглянуть внутрь системы и до какой-то степени представить себе протекающий в ней процесс рассуждений, можно воспользоваться прологовскими средствами трассировки. Но эти средства в большинстве случаев окажутся недостаточно гибкими для наших целей. Поэтому, вместо того, чтобы воспользоваться собственным механизмом интерпретации Пролога, который не сможет справиться с нужным нам способом взаимодействия с пользователем, мы создадим свое средство интерпретации в виде специальной надстройки над пролог-системой. Этот новый интерпретатор будет включать в себя средства для взаимодействия с пользователем.

14.4.1. Процесс рассуждений

Наш интерпретатор будет принимать вопрос и искать на него ответ. Язык правил допускает, чтобы в условной части правила была И/ИЛИ-комбинация условий. Вопрос на входе интерпретатора может быть такой же комбинацией подвопросов. Поэтому процесс поиска ответов на эти вопросы будет аналогичен процессу поиска в И/ИЛИ-графах, который мы обсуждали в гл. 13.

Ответ на заданный вопрос можно найти несколькими способами в соответствии со следующими принципами:

Для того, чтобы найти ответ *Отв* на вопрос *В*, используйте одну из следующих возможностей:

- если *В* найден в базе знаний в виде факта, то *Отв* – это «*В* это правда»

- если в базе знаний существует правило вида

  «если *Условие* то *B*»,

  то для получения ответа *Отв* рассмотрите *Условие*
- если вопрос *B* можно задавать пользователю, спросите пользователя об истинности *B*
- если *B* имеет вид *B1 и B2*, то рассмотрите *B1*, а затем,
  если *B1* ложно, то положите *Отв* равным «*B* это ложь», в противном случае рассмотрите *B2* и получите *Отв* как соответствующую комбинацию ответов на вопросы *B1* и *B2*
- если *B* имеет вид *B1 или B2*, то рассмотрите *B1*, а затем,
  если *B1* истинно, то положите *Отв* равным «*B1* это правда», в противном случае рассмотрите *B2* и получите *Отв* как соответствующую комбинацию ответов на вопросы *B1* и *B2*.

Вопросы вида

не B

обрабатываются не так просто, и мы обсудим их позже.

14.4.2. Формирование ответа на вопрос «почему»

Вопрос «*почему*» возникает в ситуации, когда система просит пользователя сообщить ей некоторую информацию, а пользователь желает знать, *почему* эта информация необходима. Допустим, что система спрашивает:

a – это правда?

В ответ пользователь может спросить:

почему?

Объяснение в этом случае выглядит примерно так:

Потому, что

Я могу использовать *a*,
чтобы проверить по правилу $П_a$, что *b*, и

Я могу использовать *b*,
чтобы проверить по правилу $П_b$, что *c*, и

Я могу использовать *c*,
чтобы проверить по правилу $П_c$, что *d*, и

...

Я могу использовать *y*,
чтобы проверить по правилу $П_y$, что *z*, и

z – это ваш исходный вопрос.

Объяснение – это демонстрация того, как система намерена использовать информацию, которую она хочет получить от пользователя. Намерения системы демонстрируются в виде цепочки правил и целей, соединяющей эту информацию с исходным вопросом.

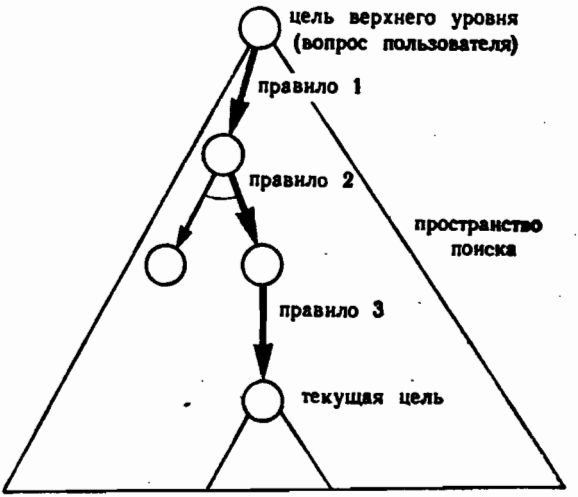


Рис. 14.8. Объяснение типа "почему". На вопрос "Почему вас интересует текущая цель?" дается объяснение в виде цепочки правил и целей, соединяющей текущую цель с исходным вопросом пользователя находящимся в верхушке дерева. Эта цепочка называется трассой.

Будем называть такую цепочку *трассой*. Трассу можно себе представлять как цепочку правил, соединяющую в И/ИЛИ-дереве вопросов текущую цель с целью самого верхнего уровня так, как это показано на рис. 14.8. Таким образом, для формирования ответа на вопрос «почему» нужно двигаться в пространстве поиска от текущей цели вверх вплоть до самой верхней цели. Для того, чтобы суметь это сделать, нам придется в процессе рассуждений сохранять трассу в явном виде.

14.4.3. Формирование ответа на вопрос «как»

Получив ответ на свой вопрос, пользователь возможно захочет увидеть, *как* система пришла к такому заключению. Один из подходящих способов ответить на вопрос «как» – это представить доказательство, т. е. те правила и подцели, которые использовались для достижения полученного заключения. Это доказательство в случае нашего языка записи правил имеет вид решающего И/ИЛИ-дерева. Поэтому наша машина логического вывода будет не просто отвечать на вопрос, соответствующий цели самого верхнего уровня – этого нам недостаточно, а будет выдавать в качестве ответа решающее И/ИЛИ-дерево, составленное из имен правил и подцелей. Затем это дерево можно будет отобразить на выходе системы в качестве объяснения типа «как». Объяснению можно придать удобную для восприятия форму, если каждое поддерево печатать с надлежащим отступом, например:

```
питер это хищник
  было выведено по прав3 из
    питер это млекопитающее
      было выведено по прав1 из
        питер имеет шерсть
          было сказано
    и
    питер ест мясо
      было сказано
```

14.5. Реализация

Теперь мы приступим к реализации нашей оболочки, следуя тем идеям, которые обсуждались в предыдущем разделе. На рис. 14.9 показаны основные объекты, которыми манипулирует оболочка. **Цель** – это вопрос, подлежащий рассмотрению; **Трасса** – это цепочка, составленная из «целей-предков» и правил, находящихся между вершиной **Цель** и вопросом самого верхнего уровня; **Ответ** – решающее дерево типа И/ИЛИ для вершины **Цель**.

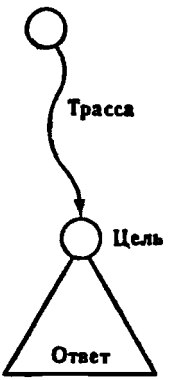


Рис. 14.9. Отношение **рассмотреть(Цель, Трасса, Ответ)**. **Ответ** – это И/ИЛИ решающее дерево для целевого утверждения **Цель**.

Основными процедурами оболочки будут:

рассмотреть(Цель, Трасса, Ответ)

Эта процедура находит ответ **Ответ** на вопрос **Цель**. Процедура

ответпольз(Цель, Трасса, Ответ)

порождает решения для тех вопросов **Цель**, которые можно задавать пользователю. Она спрашивает пользователя об истинности утверждения **Цель**, а также отвечает на вопросы «почему». Процедура

выдать(Ответ)

выводит результат и отвечает на вопросы «как». Все эти процедуры приводятся в действие процедурой-драйвером **эксперт**.

14.5.1. Процедура рассмотреть

Центральной процедурой оболочки является процедура

рассмотреть(Цель, Трасса, Ответ)

которая будет находить ответ **Ответ** на заданный вопрос **Цель**, используя принципы, намеченные в общих чертах в разд. 14.4.1: найти **Цель** среди фактов базы знаний, или применить правило из базы знаний, или спросить пользователя, или же обработать **Цель** как И/ИЛИ-комбинацию подцелей.

Аргументы имеют следующий смысл и следующую структуру:

Цель вопрос, подлежащий рассмотрению, представленный как И/ИЛИ-комбинация простых утверждений, например

X имеет перья или X летает или
X откладывает яйца

Трасса цепочка, составленная из целей-предков и правил, расположенных между **Цель** и исходной целью самого верхнего уровня. Представляется как список, состоящий из элементов вида

Цель по Прав

что означает: **Цель** рассматривалась с использованием правила **Прав**. Например, пусть исходной целью будет «питер это тигр», а текущей целью – «питер ест мясо». В соответствии с базой знаний рис. 14.5 имеем трассу

[(питер это хищник) по прав3,
(питер это тигр) по прав5]

Смысл ее можно выразить так:

Я могу использовать «питер ест мясо» для того, чтобы проверить по прав3, что

«пнтер это хнщинк».
Далее, я могу нспользовать «пнтер это хнщинк» для того, чтобы провернть по прав5, что «пнтер это тнгр».

Ответ решающее И/ИЛИ-дерево для вопроса **Цель**. Общая форма представления для объекта **Ответ**:

Заключение было Найдено

где **Найдено** – это обоснование для результата **Заключение**. Следующне трн примера нллюстрируют различные варнанты ответов:

```
(1)   (соед(радиатор,предохр1) это правда) было
                          'найдено как факт'
(2)   (пнтер ест мясо) это ложь было сказано
(3)   (питер это хнщник) это правда было
        ( 'выведено по' прав3 из
      (пнтер это млекопитающее) это правда было
        ( 'выведено по' прав1 из
      (питер имеет шерсть) это правда было сказано)
      и
      (питер ест мясо) это правда было сказано )
```

На рнс. 14.10 показана прологовская программа для процедуры **рассмотреть**. В этой программе реализованы принципы разд. 14.4.1 с использованием только что описанных структур данных.

```
% Процедура
%
% рассмотреть( Цель, Трасса, Ответ)
%
% находит Ответ на вопрос Цель. Трасса – это цепочка
% целей-предков и правил. "рассмотреть" стремится найти
% положительный ответ на вопрос. Ответ "ложь" выдается
% только в том случае, когда рассмотрены все возможности,
% и все онн дали результат "ложь".

:- op( 900, xfx, :).
:- op( 800, xfx, было).
:- op( 870, fx, если).
:- op( 880, xfx, то).
:- op( 550, xfy, или).
:- op( 540, xfy, и).
```

```
:- op( 300, fx, 'выведено по').
:- op( 600, xfx, из).
:- op( 600, xfx, по).

% В программе предполагается,что op(700,xfx,это),op(500,fx,не)
рассмотреть( Цель,Трасса,Цель это правда
                        было 'найдено как факт') :-
   факт :Цель.

% Предполагается, что для каждого типа цели
% существует только одно правило
рассмотреть( Цель, Трасса,
      Цель это ПравдаЛожь
      было 'выведено по' Прав из Ответ) :-
  Прав : если Условие то Цель,
                % Правило, относящееся к цели
  рассмотреть(Условие,[Цель по Прав|Трасса],Ответ),
  истинность( Ответ, ПравдаЛожь).

рассмотреть( Цель1 и Цель2, Трасса, Ответ) :- !,
   рассмотреть( Цель1, Трасса, Ответ1),
   продолжить(Ответ1,Цель1 и Цель2,Трасса,Ответ).

рассмотреть( Цель1 или Цель2, Трасса, Ответ) :-
   рассм_да( Цель1, Трасса, Ответ);
                % Положительный ответ на Цель1
   рассм_да( Цель2, Трасса, Ответ).
                % Положительный ответ на Цель2

рассмотреть(Цель1 или Цель2,Трасса,
                        Ответ1 и Ответ2) :- !,
   not рассм_да( Цель1, Трасса, _),
   not рассм_да( Цель2, Трасса, _),
                % Нет положительного ответа
   рассмотреть( Цель1, Трасса, Ответ1),
                % Ответ1 отрицательный
   рассмотреть( Цель2, Трасса, Ответ2).
                % Ответ2 отрицательный

рассмотреть(Цель,Трасса,
                Цель это Ответ было сказано) :-
   ответпольз(Цель,Трасса,Ответ).% Ответ дан пользователем
рассм_да( Цель, Трасса, Ответ) :-
   рассмотреть( Цель, Трасса, Ответ),
   положительный( Ответ).

продолжить( Ответ1, Цель1 и Цель2, Трасса, Ответ) :-
   положительный( Ответ1),
```

```
    рассмотреть( Цель2, Трасса, Ответ2),
    (положительный(Ответ2),Ответ = Ответ1 и Ответ2;
     отрицательный(Ответ2), Ответ = Ответ2).
продолжить( Ответ1, Цель1 и Цель2, _, Ответ1) :-
    отрицательный( Ответ1).

истинность(Вопрос это ПравдаЛожь было Найдено,
                                    ПравдаЛожь) :- !.

истинность( Ответ1 и Ответ2, ПравдаЛожь) :-
    истинность( Ответ1, правда),
    истинность( Ответ2, правда), !,
    ПравдаЛожь = правда;
    ПравдаЛожь = ложь.

положительный( Ответ) :-
    истинность( Ответ, правда).

отрицательный( Ответ) :-
    истинность( Ответ, ложь).
```

Рис. 14.10. Основная процедура оболочки экспертной системы.

14.5.2. Процедура ответпольз

Прежде чем перейти к написанию процедуры **ответпольз**, давайте рассмотрим одну полезную вспомогательную процедуру

принять(Ответ)

В процессе диалога часто возникает ситуация, когда от пользователя ожидается ответ «да», «нет» или «почему». Процедура **принять** предназначена для того, чтобы извлечь один из этих ответов, понимая его правильно и в тех случаях, когда пользователь применяет сокращения ('д' или 'н') или делает ошибки. Если ответ пользователя непонятен, то **принять** просит дать другой вариант ответа.

```
принять( Ответ) :-
    read( Ответ1),
    означает( Ответ1, Значение), !,
                        % Ответ1 означает что-нибудь?
```

```
    Ответ = Значение;        % Да
    nl, write( 'Непонятно, попробуйте еще раз,% Нет
                пожалуйста'), nl,
    принять( Ответ).         % Новая попытка

означает( да, да).
означает( д, да).
означает( нет, нет).
означает( н, нет).
означает( почему, почему).
означает( п, почему).
```

Следует заметить, что процедурой **принять** нужно пользоваться с осторожностью, так как она содержит взаимодействие с пользователем. Следующий фрагмент программы может служить примером неудачной попытки запрограммировать интерпретацию ответов пользователя:

```
принять( да), интерп_да( ...);
принять( нет), интерп_нет( ...);
...
```

Здесь, если пользователь ответит «нет», то программа попросит его повторить свой ответ. Поэтому более правильный способ такой:

```
принять( Ответ),
( Ответ = да, интерп_да( ...);
  Ответ = нет, интерп_нет( ...);
  ... )
```

Процедура

ответпольз(Цель, Трасса, Ответ)

спрашивает пользователя об истинности утверждения **Цель**. **Ответ** – это результат запроса. **Трасса** используется для объяснения в случае, если пользователь спросит «почему».

Сначала процедура **ответпольз** должна проверить, является ли **Цель** информацией, которую можно запрашивать у пользователя. Это свойство объекта **Цель** задается отношением

можно_спросить(Цель)

которое в дальнейшем будет усовершенствовано. Если спросить можно, то утверждение **Цель** выдается пользователю, который, в свою очередь, указывает

истинно оно или ложно. Если пользователь спросит «почему», то ему выдается **Трасса**. Если утверждение **Цель** истинно, то пользователь укажет также значения содержащихся в нем переменных (если таковые имеются).

Все вышеизложенное можно запрограммировать (в качестве первой попытки) следующим образом:

```
ответпольз( Цель, Трасса, Ответ) :-
    можно_спросить( Цель),   % Можно ли спрашивать
    спросить( Цель, Трасса, Ответ).
            % Задать вопрос относительно утверждения Цель

спросить( Цель, Трасса, Ответ) :-
    показать( Цель),
            % Показать пользователю вопрос
    принять( Ответ1),              % Прочесть ответ
    обработать( Ответ1, Цель, Трасса, Ответ).
            % Обработать ответ

обработать( почему, Цель, Трасса, Ответ) :-
            % Задан вопрос "почему"
    показать_трассу( Трасса),
            % Выдача ответа на вопрос "почему"
    спросить( Цель, Трасса, Ответ).
            % Еще раз спросить

обработать( да, Цель, Трасса, Ответ) :-
            % Пользователь ответил, что Цель истинна
    Ответ = правда,
    запрос_перем( Цель);
            % Вопрос о значении переменных
    спросить( Цель, Трасса, Ответ).
            % Потребовать от пользователя новых решений

обработать( нет, Цель, Трасса, ложь).
            % Пользователь ответил, что Цель ложна

показать( Цель) :-
    nl, write( 'Это правда:'),
    write( Цель), write( ?), nl.
```

Обращение к процедуре **запрос_перем(Цель)** нужно для того, чтобы попросить пользователя указать значение каждой из переменных, содержащихся в утверждении **Цель**:

```
запрос_перем( Терм) :-
  var( Терм), !,              % Переменная?
```

```
    nl, write( Терм), write( '='),
    read( Терм).                % Считать значение переменной

запрос_перем( Терм) :-
    Терм =.. [Функтор | Аргументы],
            % Получить аргументы структуры
    запрос_арг( Аргументы).
            % Запросить значения переменных в аргументах

запрос_арг( []).

запрос_арг( [Терм | Термы]) :-
    запрос_перем( Терм),
    запрос_арг( Термы).
```

Проведем несколько экспериментов с процедурой **ответпольз**. Пусть, например, известно, что пользователя можно спрашивать о наличии бинарного отношения **ест**:

```
можно_спросить( X ест Y).
```

(В приведенных ниже диалогах между пролог-системой и пользователем тексты пользователя даются полужирным шрифтом, а реплики пролог-системы курсивом).

```
?- ответпольз( питер ест мясо, [], Ответ).

    Это правда: питер ест мясо?   % Вопрос пользователю
    да.                           % Ответ пользователя

    Ответ = правда
```

Более интересный пример диалога (с использованием переменных) мог бы выглядеть примерно так:

```
?- ответпольз( Кто ест Что, [], Ответ).

Это правда: _17 ест _18?
            % Пролог дает переменным свои внутренние имена
да.
_17 = питер.
_18 = мясо.

Ответ = правда.
Кто = питер
Что = мясо;       % Возврат для получения других решений
```

Это правда: _17 ест _18?
да.
_17 **= сьюзен.**
_18 **= бананы.**

Ответ = правда
Кто = сьюзен
Что = бананы;

Это правда : _17 ест _18?
нет.

Ответ = ложь

14.5.3. Усовершенствование процедуры ответпольз

Один из недостатков нашей процедуры **ответпольз**, который хорошо виден из приведенного выше диалога, – это появление на выходе системы имен, генерируемых пролог-системой, что выглядит довольно неуклюже. Символы, подобные _17, следовало бы заменить на более осмысленные слова.

Другой, более серьезный дефект этой версии процедуры **ответпольз** состоит в следующем. Если мы еще раз обратимся к **ответпольз**, задав ту же самую цель, то пользователю придется повторно вводить все варианты решений. Поэтому, если наша экспертная система придет в процессе рассуждений к рассмотрению той же самой цели второй раз, то, вместо того, чтобы использовать информацию, уже полученную от пользователя, она проведет с пользователем в точности тот же самый скучный диалог.

Давайте исправим эти два дефекта. Во-первых, улучшение внешнего вида запросов системы будет основано на введении стандартного формата для каждой «запрашиваемой» цели. Для этого в отношение **можно_спросить** мы добавим второй аргумент, который и будет задавать этот формат, как видно из следующего примера:

можно_спросить(X ест Y,'Животное' ест 'Что-то').

При передаче запроса пользователю каждая перемен-

ная вопроса должна быть заменена на ключевое слово, взятое из формата, например:

```
?- ответпольз( X ест Y, [], Ответ).

Это правда: Животное ест Что-то?
да.

Животное = питер.
Что-то = мясо.

Ответ = правда
X = питер
Y = мясо
```

В улучшенной версии процедуры **ответпольз**, показанной на рис. 14.11, такое форматирование запросов выполняется процедурой

формат(Цель, ВнешФормат, Вопрос, Перем0, Перем)

Здесь **Цель** – утверждение, которое нужно форматировать. **ВнешФормат** определяет внешний формат этого утверждения, задаваемый отношением

можно_спросить(Цель, ВнешФормат)

Вопрос – это **Цель**, отформатированная в соответствии с **ВнешФормат**. **Перем** – список переменных, входящих в **Цель**, вместе с соответствующими ключевыми словами (как указано в **ВнешФормат**), причем список **Перем** получается из списка **Перем0** добавлением новых переменных. Например:

```
?- формат( X передает документы Y,
           'Кто' передает 'Что' 'Кому',
           Вопрос, [], Перем).

Вопрос = 'Кто' передает документы 'Кому',
Перем = [ X/'Кто', Y/'Кому'].
```

Второе усовершенствование, состоящее в устранении повторных вопросов к пользователю, будет более трудным. Во-первых, все ответы пользователя следует запоминать, с тем чтобы их можно было отыскать в памяти в более поздний момент времени. Для этого достаточно сделать ответы пользователя элементами некоторого отношения и применить **assert**, например

```
assert( сказано(мери передает документы друзьям,
                правда) ).
```

В ситуации, когда имеется несколько решений, предложенных пользователем для одной и той же цели, в память относительно нее будет записано несколько фактов. Здесь возникает одно осложнение. Допустим, что в нескольких местах программы встречаются различные варианты некоторой цели (отличающиеся именованием переменных). Например:

```
( X имеет Y) и                  % Первый вариант - Цель1
...
( X1 имеет Y1) и                % Второй вариант - Цель2
...
```

Допустим также, что пользователя просят (через механизм возвратов) предложить несколько решений для **Цель1**. Затем процесс рассуждений продвигается вплоть до **Цель2**. Так как у нас уже есть несколько решений для **Цель1**, мы захотим, чтобы система автоматически применила их и к **Цель2** (поскольку очевидно, что они удовлетворяют **Цель2**). Теперь предположим, что система пытается применить эти решения к **Цель2**, но ни одно из них не удовлетворяет некоторой другой цели, расположенной ниже. Система делает возврат к **Цель2** и просит пользователя предложить новые решения. Если пользователь введет еще несколько решений, то их также придется запомнить. И если система в дальнейшем сделает возврат к **Цель1**, то эти новые решения надо будет применить к **Цель1**.

Для того, чтобы правильным образом использовать информацию, вводимую пользователем по запросам из разных точек программы, мы будем снабжать каждую такую информацию специальным индексом. Таким образом, факты, запоминаемые системой, будут иметь вид

сказано(Цель, Истинность, Индекс)

где **Индекс** - это значение счетчика ответов пользователя. Процедура

ответпольз(Цель, Трасса, Ответ)

теперь должна будет отслеживать число решений, уже порожденных механизмом возвратов к моменту обращения к этой процедуре. Это можно сделать при помощи другого варианта процедуры **ответпольз** с четырьмя аргументами:

ответпольз(Цель, Трасса, Ответ, N)

где N – некоторое целое число. Такое обращение к **ответпольз** должно порождать решения для **Цель** с индексами, начиная с N и далее. Обращение

ответпольз(Цель, Трасса, Ответ)

соответствует получению *всех* решений, индексируемых, начиная с 1, поэтому мы имеем следующее соотношение:

```
ответпольз( Цель, Трасса, Ответ) :-
    ответпольз( Цель, Трасса, Ответ, 1).
```

Принцип работы процедуры

ответпольз(Цель, Трасса, Ответ, N)

таков: сначала получить решения для **Цель**, отыскивая в памяти все уже известные решения с индексами, начиная с N и далее. Когда все старые решения исчерпаются, начать задавать вопросы пользователю относительно утверждения **Цель**, записывая полученные таким образом новые решения в память при помощи **assert** и индексируя их должным образом при помощи целых чисел. Когда пользователь сообщит, что больше нет решений, записать в память факт

конец_ответов(Цель)

Если пользователь с самого начала скажет, что решений нет вообще, то записать факт

сказано(Цель, ложь, Индекс)

Находя в памяти те или иные решения, процедура **ответпольз** должна правильно интерпретировать подобную информацию.

Однако существует еще одна трудность. Пользователь может, оставляя некоторые переменные неконкретизированными, указывать общие решения. Если найдено положительное решение, более общее, чем **Цель**, или столь же общее, как **Цель**, то нет смысла продолжать задавать вопросы об утверждении **Цель**, поскольку мы уже имеем более общее решение. Аналогичным образом следует поступить, если обнаружен факт

сказано(Цель, ложь, _)

Программа **ответпольз**, показанная на рис. 14.11,

учитывает все вышеприведенные соображения. В нее введен новый аргумент **Копия** (копия утверждения **Цель**), который используется в нескольких случаях сопоставлений вместо **Цель**, с тем чтобы оставить в неприкосновенности переменные утверждения **Цель**. Эта программа использует также два вспомогательных отношения. Одно из них

конкретный(Терм)

истинно, если **Терм** не содержит переменных. Другое

конкретизация(Терм, Терм1)

означает, что **Терм1** есть некоторая конкретизация (частный случай) терма **Терм**, т. е. **Терм** – это утверждение не менее общее, чем **Терм1**. Например:

конкретизация(X передает информацию Y,
мэри передает информацию Z)

Обе процедуры основаны на еще одной процедуре:

нумпер(Терм, N, M)

Эта процедура «нумерует» переменные, содержащиеся в **Терм**, заменяя каждую из них на некоторый специальный новый терм таким образом, чтобы эти «нумерующие» термы соответствовали числам от N до M–1. Например, пусть эти термы имеют вид

пер/0, пер/1, пер/2, ...

тогда в результате обращения к системе

?- Терм = f(X, t(a,Y,X)), нумпер(Терм,5,M).

мы получим

```
Терм = f( пер/5, t( a, пер/6, пер/5) )
M = 7
```

Отношение, подобное **нумпер**, часто входит в состав пролог-системы в качестве встроенной процедуры. Если это не так, то его можно реализовать программно следующим способом:

```
нумпер( Терм, N, Nплюс1) :-
    var( Терм), !,                    % Переменная?
    Терм = пер/N,
    Nплюс1 is N + 1.
```

```
% Процедура
%
% ответпольз( Цель, Трасса, Ответ)
%
% порождает, используя механизм возвратов, все решения
% для целевого утверждения Цель, которые указал пользователь.
% Трасса - это цепочка целей-предков и правил,
% используемая для объяснения типа "почему".

ответпольз( Цель, Трасса, Ответ) :-
    можно_спросить( Цель, _),  % Можно спросить?
    копия( Цель, Копия),       % Переименование переменных
    ответпольз( Цель, Копия, Трасса, Ответ, 1).

%Не спрашивать второй раз относительно конкретизированной цели

ответпольз( Цель, _, _, _, N) :-
    N > 1,                     % Повторный вопрос?
    конкретный( Цель), !,      % Больше не спрашивать
    fail.

% Известен ли ответ для всех конкретизаций утверждения Цель?

ответпольз( Цель, Копия, _, Ответ, _) :-
    сказано( Копия, Ответ, _),
    конкретизация( Копия, Цель), !.% Ответ известен

%Найти все известные решения для Цель с индексами, начиная с N

ответпольз( Цель, _, _, правда, N) :-
    сказано( Цель, правда, M),
    M >= N.

% Все уже сказано об утверждении Цель?

ответпольз( Цель, Копия, _, Ответ, _) :-
    конец_ответов( Копия),
    конкретизация( Копия, Цель), !,% Уже все сказано
    fail.

% Попросить пользователя дать (еще) решения

ответпольз( Цель, _, Трасса, Ответ, N) :-
    спросить_польз( Цель, Трасса, Ответ, N).

спросить_польз( Цель, Трасса, Ответ, N) :-
    можно_спросить( Цель, ВнешФормат),
    формат( Цель, ВнешФормат, Вопрос, [], Перем),
                               % Получить формат вопроса
    спросить(Цель,Вопрос,Перем,Трасса,Ответ,N).
```

```
спросить(Цель, Вопрос, Перем, Трасса, Ответ, N) :-
    nl,
    ( Перем = [], !,                    % Сформулкровать вопрос
      write( 'Это правда: ');
      write( 'Есть (еще) решения для :' ) ),
    write( Вопрос), write( '?'),
    принять( Ответ1), !,                % Ответ1 = да/нет/почему
    обработать(Ответ1, Цель, Вопрос, Перем,
                                        Трасса, Ответ, N).

обработать(почему, Цель, Вопрос, Перем,
                                        Трасса, Ответ, N):-
    выд_трассу( Трасса),
    спросить(Цель, Вопрос, Перем, Трасса, Ответ, N).

обработать(да, Цель, _, Перем, Трасса, правда, N) :-
    след_индекс( Инд),
                % Получить новый индекс для "сказано"
    Инд1 is Инд + 1,
    ( запрос_перем( Перем),
      assertz( сказано( Цель, правда, Инд) );
                                        % Запись решения
    копия( Цель, Копия),                % Копирование цели
    ответпольз(Цель, Копия, Трасса, Ответ, Инд1) ).
                                        % Есть еще решения?

обработать( нет, Цель, _, _, _, ложь, N) :-
    копия( Цель, Копия),
    сказано( Копия, правда, _), !,
                % 'нет' означает: больше нет решений
    assertz( конец_ответов( Цель) ),
                % Отметить конец ответов
    fail;
    след_индекс( Инд),
                % Следующий свободный индекс для "сказано"
    assertz( сказано( Цель, ложь, Инд) ).
                % 'нет' означает: нет ни одного решения

формат(Пер, Имя, Имя, Перем, [Пер/Имя | Перем]) :-
    var( Пер), !.
формат( Атом, Имя, Атом, Перем, Перем) :-
    atomic( Атом), !,
    atomic( Имя).
формат( Цель, Форм, Вопрос, Перем0, Перем) :-
    Цель =.. [Функтор | Арг1],
    Форм =.. [Функтор | Форм1],
    формвсе( Арг1, Форм1, Арг2, Перем0, Перем),
    Вопрос =.. [Функтор | Арг2].
```

```
формвсе( [], [], [], Перем, Перем).
формвсе([Х|СпХ],[Ф|СпФ],[В|СпВ],Перем0,Перем) :-
    формвсе( СпХ, СпФ, СпВ, Перем0, Перем1),
    формат( Х, Ф, В, Перем1, Перем).

запрос_перем( []).
запрос_перем( [Переменная/Имя | Переменные]) :-
    nl, write( Имя), write( '='),
    read( Переменная),
    запрос_перем( Переменные).

выд_трассу( []) :-
    nl, write( 'Это был ваш вопрос'), nl.
выд_трассу( [Цель по Прав | Трасса] ) :-
    nl, write( 'Чтобы проверить по' ),
    write( Прав), write( ', что'),
    write( Цель),
    выд_трассу( Трасса).

конкретный( Терм) :-
    иумпер( Терм, 0, 0).          % Нет переменных в Терм'е

% конкретизация( Т1, Т2) означает, что Т2 - конкретизация Т1,
% т.е. терм Т1 - более общий, чем Т2, или той же степени
% общности, что и Т2

конкретизация( Терм, Терм1) :-
                % Терм1 - частный случай Терм'а
    копия( Терм1, Терм2),
                % Копия Терм1 с новыми переменными
    иумпер( Терм2, 0, _), !,
    Терм = Терм2. % Успех, если Терм1 - частный случай Терм2

копия( Терм, НовТерм) :-
                % Копия Терм'а с новыми переменными
    asserta( copy( Терм) ),
    retract( copy( НовТерм) ), !.

посл_индекс( 0).          % Начальный индекс для "сказано"

след_индекс( Инд) :-  % Следующий индекс для "сказано"
    retract( посл_индекс( ПослИнд) ), !,
    Инд is ПослИнд + 1,
    assert( посл_индекс( Инд) ).
```

Рис. 14.11. Оболочка экспертной системы: Вопросы к пользователю и ответы на вопросы "почему".

```
нумпер( Терм, N, M) :-
    Терм =.. [Функтор|Аргументы],  % Структура или атом
    нумарг( Аргументы, N, M).
                % Пронумеровать переменные в аргументах
нумарг( [], N, N) :- !.
нумарг( [X | Спис], N, M) :-
    нумпер( X, N, N1),
    нумарг( Спис, N1, M).
```

14.5.4. Процедура выдать

Процедура

выдать(Ответ)

приведенная на рис. 14.12, показывает пользователю окончательный результат консультационного сеанса и дает объяснения типа «как». **Ответ** включает в себя как ответ на вопрос пользователя, так и дерево вывода, демонстрирующее *как* система пришла к такому заключению. Сначала процедура выдать представляет пользователю свое заключение. Затем, если пользователь пожелает узнать, *как* это заключение достигнуто, то печатается дерево вывода в некоторой удобной для восприятия форме – это и есть объяснение типа «как». Форма объяснения показана в примере разд. 14.4.3.

14.5.5. Драйвер верхнего уровня

И наконец, для того, чтобы иметь удобный доступ к оболочке из интерпретатора Пролога, нам необходима процедура, выполняющая функцию «драйвера». На рис. 14.13 показано, как могла бы выглядеть предназначенная для этой цели процедура **эксперт**. Драйвер **эксперт** производит запуск трех основных модулей оболочки (рис. 14.10 – 14.12) и координирует их работу. Например:

```
% Выдача заключения консультационного сеанса и
% объяснения типа "как"
выдать( Ответ) :-
  nl, заключение( Ответ),
  nl, write( 'Хотите узнать, как?'),
  принять( Ответ1),
  ( Ответ1 = да, !, отобр( Ответ);
    true) .                    % Показ решающего дерева

заключение( Ответ1 и Ответ2) :- !,
  заключение( Ответ1), write( 'и'),
  заключение( Ответ2).

заключение( Заключение было Найдено) :-
  write( Заключение).

% "отобр" отображает полное решающее дерево

отобр( Решение) :-
  nl, отобр( Решение, 0), !.                % Отступ 0

отобр( Ответ1 и Ответ2, H) :- !,            % Отступ H
  отобр( Ответ1, H),
  tab( H), write( 'и'), nl,
  отобр( Ответ2, H).

отобр( Ответ был Найден, H) :-              % Отступ H
  tab( H), печответ( Ответ),      % Показ заключения
  nl, tab( H),
  write( 'было'),
  отобр1( Найден, H).             % Показ доказательства

отобр1( Выведено из Ответ, H) :- !,
  write( Выведено), write( 'из'),% Показ имени правила
  nl, H1 is H + 4,
  отобр( Ответ, H1).              % Показ "предшественника"

отобр1( Найдено, _) :-
            % Найдено = 'сказано' или 'найдено как факт'
  write( Найдено), nl.

печответ( Цель это правда) :- !,
  write( Цель).    % На выходе 'это правда' опускается

печответ( Ответ) :-                % Отрицательный ответ
  write( Ответ).
```

Рис. 14.12. Оболочка экспертной системы:
Отображение окончательного результата и объяснение типа "как".

```
?- эксперт.
Пожалуйста, спрашивайте: % Пригласнение пользователю
X это животное и голивф это X.% Вопрос пользователя
Это правда: голиаф имеет шерсть?
...
```

14.5.6. Одно замечанна по поводу программы—оболочки

В некоторых местах нашей программы-оболочки обиаруживается недостаток той «декларативной ясности», которая так характерна для программ, написанных на Прологе. Причина состоит в том, что нам пришлось предусмотреть в этой программе довольно жесткое управление процессом функционирования оболочки. Ведь, согласно нашему замыслу, экспертная система должна была не только находить ответы на вопросы, ио и делать это некоторым разумным с точки зрения пользователя способом. В связи с этим нам пришлось реализовать вполне определенное *поведение* системы в процессе решения задач, а не просто некоторое отношение ввода-вывода. В результате получилась программа более процедуриого характера, чем обычио. Все это может послужить прнмером ситуации, когда, не нмея возможности рассчитывать на собственные процедурные механизмы Пролога, мы вынуждены взять на себя детальное описание процедурного поведения системы.

14.5.7. Цели с отрицанием

Использование знака отрицання в левых частях правил, а следовательно, н в вопросах, обрабатываемых процедурой рассмотреть, представляется естественным и его следует разрешить. В качестве первой попыткн можно предложить следующий способ работы с отрицанием целей:

```
% Процедура-драйвер верхнего уровня
эксперт :-
  принять_вопрос( Вопрос),
                % Ввести вопрос пользователя
  ( ответ_да( Вопрос);
                % Попытка найти положительный ответ
    ответ_нет( Вопрос) ).
  % Если нет положительного ответа, то найти отрицательный
ответ_да( Вопрос) :-
                % Искать положительный ответ на Вопрос
  статус( отрицательный),
                % Пока еще нет положительного ответа
  рассмотреть( Вопрос, [], Ответ),      % Трасса пуста
  положительный( Ответ),    % Искать положительный ответ
  стутус( положительный),
                % Найден положительный ответ
  выдать( Ответ), nl,
  write( 'Нужны еще решения?' ),
  принять( Ответ1),     % Прочесть ответ пользователя
  Ответ1 = нет.
        % В противном случае возврат к "рассмотреть"
ответ_нет(Вопрос):-
        % Искать отрицательный ответ на Вопрос
  retract( пока_нет_положительного_решения), !,
                % Не было положительного решения?
  рассмотреть( Вопрос, [], Ответ),
  отрицательный( Ответ),
  выдать(Ответ), nl,
  write( 'Нужны еще решения?' ),
  принять( Ответ1),
  Ответ1 = нет.
        % В противном случае - возврат к "рассмотреть"
статус( отрицательный) :-
  assert( пока_нет_положительного_решения).

статус( положительный) :-
  retract(пока_нет_положительного_решения),!;true.

принять_вопрос( Вопрос) :-
  nl, write( 'Пожалуйста, спрашивайте:'), nl,
  read( Вопрос).
```

Рис. 14.13. Оболочка экспертной системы: драйвер. Обращение к оболочке из Пролога при помощи процедуры **эксперт**.

```
рассмотреть( не Цель, Трасса, Ответ) :- !,
    рассмотреть( Цель, Трасса, Ответ1),
    обратить( Ответ1, Ответ).
        % Получить обратное истинностное значение

обратить( Цель это правда было Найдено,
          ( не Цель) это ложь было Найдено).

обратить( Цель это ложь было Найдено,
          ( не Цель) это правда было Найдено).
```

Если Цель конкретизирована, то все в порядке, если же нет, то возникают трудности. Рассмотрим, например, такой диалог:

```
?- эксперт.

Пожалуйста, спрашивайте:
не ( X ест мясо).

Есть (еще) решения для : Животное
да.
Животное = тигр.
```

В этот момент система даст ответ:

не (тигр ест мясо) это ложь

Такой ответ нас не может удовлетворить. Источник затруднения следует искать в том, какой смысл мы вкладываем в вопросы типа

не (X ест мясо)

В действительности мы хотим спросить: «Существует ли такой X, что X не ест мяса?» Однако процедура рассмотреть (так как мы ее определили) проинтерпретирует этот вопрос следующим образом:

(1) Существует ли такой X, что X ест мясо?
(2) Да, тигр ест мясо.

Итак,

(3) не (тигр ест мясо) это ложь.

Короче говоря, интерпретация такова: «Правда ли, что никакой X не ест мясо?» Положительный ответ мы получим, только если *никто* не ест мяса. Можно также сказать, что процедура рассмотреть отвечает на вопрос так, как будто X находится под знаком кванторa *всеобщности*:

для *всех* X: не (X ест мясо)?

а не квантора существования, в чем и состояло наше намерение:

для *некоторого* X: не (X ест мясо)?

Если рассматриваемый вопрос конкретизирован, то проблемы исчезают. В противном случае правильный способ работы с отрицаниями становится более сложным. Например, вот некоторые из возможных правил:

Для того, чтобы рассмотреть (*не Цель*), рассмотрите *Цель*, а затем:

- если *Цель* это ложь, то (*не Цель*) это правда;
- если *Цель'* – это некоторое решение для *Цель*, и *Цель'* – утверждение той же степени общности, что и *Цель*, то (*не Цель*) это ложь;
- если *Цель'* – это некоторое решение для *Цель*, и *Цель'* – более конкретное утверждение, чем *Цель*, то об утверждении (*не Цель*) нельзя сказать ничего определенного.

Можно избежать всех этих осложнений, если потребовать, чтобы отрицания стояли только перед конкретизированными целями. Если правила базы знаний формулировать должным образом, то часто удается удовлетворить этому условию. Нам это удалось в «правиле поломки» (рис. 14.7):

```
правило_поломки:
        если
            вкл( Прибор) и
            прибор( Прибор) и      % Конкретизация
            не работает( Прибор) и
            соед( Прибор, Предохр) и
            доказано( цел( Предохр) )
        то
            доказано( неиспр( Прибор) ).
```

Здесь условие

прибор(Прибор)

«защищает» следующее за ним условие

не работает(Прибор)

от неконкретизированной переменной.

Упражнение

14.3. База знаний может, в принципе, содержать цнклы. Напрнмер:

прав1: если бутылка_пуста то джон_пьян.
прав2: если джон_пьян то бутылка_пуста.

Работая с подобной базой знаний, наша процедура **рассмотреть** может зациклиться на обработке одних и тех же целей. Внесите в процедуру **рассмотреть** изменения, предотвращающие зацикливание. Используйте для этого объект **Трасса**. Однако соблюдайте осторожность: если текущая цель *сопоставима* с одной из предыдущих целей, то такую снтуацню следует рассматривать как цикл только в том случае, когда текущая цель имеет большую степень общности, чем предыдущая.

14.6. Работа с неопределенностью

14.6.1. Степень достоверности

Наша оболочка экспертной системы, описанная в предыдущем разделе, может работать только с такими вопросами (утверждениями), которые либо истинны, либо ложны. Предметные областн, в которых на любой вопрос можно ответить «правда» или «ложь», называются *категорическими*. Наши правила базы знания (также, как и данные) были категорическими, это были «категорические импликации». Однако многие области экспертных знаний не являются категорическими. Как правило, в заключениях эксперта много догадок (впрочем, высказанных с большой уверенностью), которые обычно верны, но могут быть и исключения. Как данные, относящиеся к конкретной задаче, так и импликации, содержащиеся в правилах, могут быть не вполне определенными. Неопределенность можно промоделировать, приписывая утвержде-

нням некоторые характеристики, отличные от *«истина»* и *«ложь»*. Характеристики могут иметь свое внешнее выражение в форме дескрипторов, таких, как, например, *верно*, *весьма вероятно*, *вероятно*, *маловероятно*, *невозможно*. Другой способ: степень уверенности может выражаться в форме действительного числа, заключенного в некотором интервале, например между 0 и 1 или между –5 и +5. Такую числовую характеристику называют по-разному – «коэффициент определенности», «степень доверия» или «субъективная уверенность». Более естественным было бы использовать вероятности (в математическом смысле слова), но попытки применить их на практике приводят к трудностям. Происходит это по следующим причинам:

- Экспертам, по-видимому, неудобно мыслить в терминах вероятностей. Их оценки правдоподобия не вполне соответствуют математическому определению вероятностей.
- Работа с вероятностями, корректная с точки зрения математики, потребовала бы или какой-нибудь недоступной информации, или каких-либо упрощающих допущений, не вполне оправданных с точки зрения практического приложения.

Поэтому, даже если выбранная мера правдоподобия лежит в интервале 0 и 1, более правильным будет называть ее из осторожности «субъективной уверенностью, подчеркивая этим, что имеется в виду оценка, данная экспертом. Оценки эксперта не удовлетворяют всем требованиям теории вероятностей. Кроме того, вычисления над такими оценками могут отличаться от исчисления вероятностей. Но, несмотря на это, они могут служить вполне адекватной моделью того, как человек оценивает достоверность своих выводов.

Для работы в условиях неопределенности было придумано множество различных механизмов. Мы будем рассматривать здесь механизм, используемый в системах Prospector и AL/X для минералогической разведки и локализации неисправностей соответственно. Следует заметить, что модель, применяемая в системе Prospector, несовершенна как с теоретической,

так и с практической точек зрения. Однако она использовалась на практике, она проста и может служить хорошей иллюстрацией при изложении основных принципов, а потому вполне подойдет нам, по крайней мере для первого знакомства с этой областью. С другой стороны, известно, что даже в значительно более сложных моделях не обходятся без трудностей.

14.6.2. Модель Prospector'а

Достоверность событий моделируется с помощью действительных чисел, заключенных в интервале между 0 и 1. Для простоты изложения мы будем называть их «вероятностями», хотя более точный термин – «субъективная уверенность». Отношения между событиями можно представить графически в форме «сети вывода». На рис. 14.14 показан пример сети вывода. События изображаются прямоугольниками, а отношения между ними – стрелками. Овалами изображены комбинации событий (И, ИЛИ, НЕ).

Мы будем считать, что отношения между событиями (стрелки) являются своего рода «мягкими импликациями». Пусть имеются два события *E* и *H*, и пусть информация о том, что имело место событие *E*, оказывает влияние на нашу уверенность в том, что произошло событие *H*. Если это влияние является «категорической импликацией», то можно просто написать

если *E* то *H*

В случае же «мягкой импликации» это отношение может быть менее определенным, так что ему можно приписать некоторую «силу», с которой оно действует:

если *E* то *H* с силой S

Та сила, с которой достоверность *E* влияет на уверенность в *H*, моделируется в системе Prospector при помощи двух параметров:

N = «коэффициент необходимости»
S = «коэффициент достаточности»

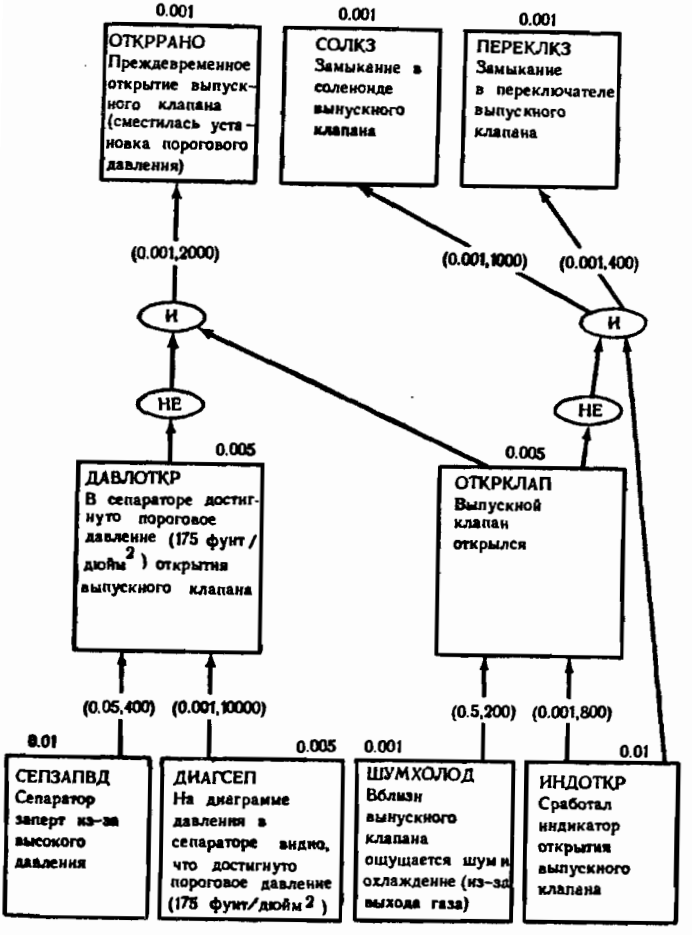


Рис. 14.14. Сеть вывода системы AL/X (заимствовано из Reiter (1980)). Числа, приписанные прямоугольникам, – априорные вероятности событий; числами на стрелках задается "сила" отношений между событиями.

В сети вывода это изображается так:

$$E \xrightarrow[(N,S)]{} H$$

Два события, участвующие в отношении, часто называют «фактом» и «гипотезой» соответственно. Допустим, что мы проверяем гипотезу H. Тогда мы будем искать такой факт E, который мог бы подтвердить либо опровергнуть эту гипотезу. S говорит нам, в какой степени *достаточно* факта E для подтверждения гипотезы H; N – насколько *необходим* факт E для подтверждения гипотезы H. Если факт E имел место, то чем больше S, тем больше уверенности в H. С другой стороны, если не верно, что имел место факт E, то чем больше N, тем менее вероятно, что гипотеза H верна. В случае, когда степень достоверности E находится где-то между полной достоверностью и невозможностью, степень достоверности H определяется при помощи интерполяции между двумя крайними случаями. Крайние случаи таковы:

(1) известно, что факта E не было
(2) известно, что факт E имел место
(3) ничего неизвестно относительно E

Для каждого события H сети вывода существует априорная вероятность $p_0(H)$ – (безусловная) вероятность события H в состоянии, когда неизвестно ни одного положительного или отрицательного факта. Если становится известным какой-нибудь факт E, то вероятность H меняет свое значение с $p_0(H)$ на $p(H|E)$. Величина изменения зависит от «силы» стрелки, ведущей из E в H. Итак, мы начинаем проверку гипотез, принимая их априорные вероятности. В дальнейшем происходит накопление информации о фактах, что находит свое отражение в изменении вероятностей событий сети. Эти изменения распространяются по сети от события к событию в соответствии со связями между событиями. Например, рассмотрим рис. 14.14 и предположим, что получена информация о срабатывании индикатора открытия выпускного клапана. Эта информация повлияет на нашу уверенность в том, что выпускной клапан открылся, что, в свою очередь, повлияет на уверенность в том, что сместилась установка порогового давления.

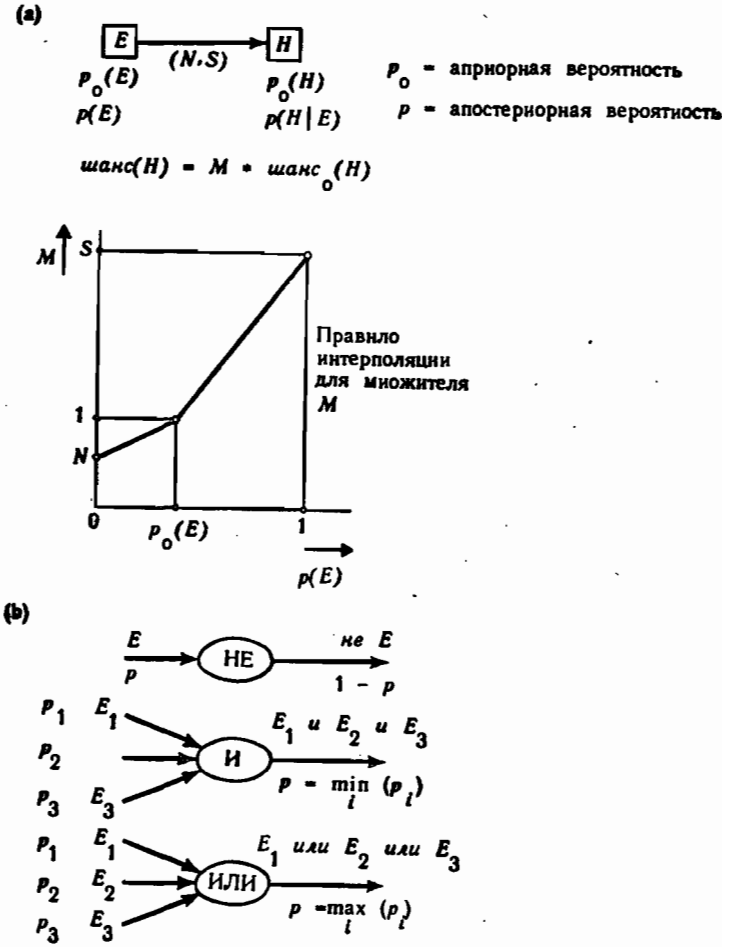


Рис. 14.15. Правила распространения вероятностей по сети, принятые в системах Prospector и AL/X: (a) "мягкая импликация" с силой (N,S); (b) логические комбинации отношений.

На рис. 14.15 показан один из способов реализации этого эффекта распространения информации по сети. Часть вычислений производится не над вероят-

ностями, а над *шансами*. Это удобно, хотя в принципе и не обязательно. Между шансами и вероятностями имеет место простое соотношение:

шанс = *вер*/(1 − *вер*)
вер = *шанс*/(1 + *шанс*)

Пусть между E и H существует отношение «мягкой импликации», тогда, в соответствии с рис. 14.15,

шанс ($H|E$) = M * *шанс* (H)

где множитель M определяется априорной и апостериорной вероятностями с учетом силы (N, S) связи между E и H. Предполагается, что правила Prospector'а (рис. 14.15) для вычисления вероятностей логических комбинаций событий (использующие *min* и *max*) правильно моделируют поведение человека при оценке субъективной уверенности в таких составных событиях.

14.6.3. Принципы реализации

Давайте сначала расширим правила языка, с тем чтобы получить возможность работать с неопределенностью. К каждому правилу мы можем добавить «силовой модификатор», определяемый двумя неотрицательными действительными числами S и N. Вот соответствующий формат:

```
ИмяПравила:   если
                 Условие
              то
                 Заключение
              с
                 Сила( N, S).
```

Примеры правил рис. 14.14 можно изобразить в этой форме так:

```
прав1 :   если
             не давлоткр и
             открклап
          то
             открклрано
          с
             сила( 0.001, 2000).
```

```
прав2 : если
            сепзапвд
        то
            давлоткр
        с
            сила( 0.05, 400).
```

Для того, чтобы произвести соответствующее расширение оболочки экспертной системы (разд. 14.5), нам понадобится внести изменения в большинство процедур. Давайте сосредоточимся только на одной из них, а именно на процедуре

рассмотреть(Цель, Трасса, Ответ)

Мы предположим, что утверждение **Цель** не содержит переменных (как это сделано в Prospector'е и в AL/X). Это сильно упростит дело (особенно в процедуре **ответпольз**). Таким образом, **Цель** будет логической комбинацией элементарных утверждений. Например:

не давлоткр и открклап

Цепочку целей-предков и правил **Трасса** можно представить таким же способом, как это сделано в разд. 14.5. Однако форму представления объекта **Ответ** придется модифицировать для того, чтобы включить в нее вероятности. Цель и ее вероятность можно соединить в один терм следующим образом:

Цель : Вероятность

Получим такой пример объекта **Ответ**:

индоткр : 1 было сказано

Смысл ответа: пользователь сообщил системе, что событие **индоткр** произошло, и что это абсолютно достоверно.

Представление объекта **Ответ** требует еще одной модификации, в связи с тем, что в одно и то же событие могут вести несколько независимых связей, которые все окажут влияние на вероятность этого события - его шанс будет помножен (рис. 14.15) на все множители. В этом случае **Ответ** будет содержать список всех ветвей вывода заключения. Приведем пример ответа такого рода для сети рис. 14.14 (для наглядности расположенный на нескольких строках):

```
давлоткр : 1 было 'выведено по'
    [ прав2 из сепзапвд : 1 было сказано,
      прав5 из диагсеп : 1 было сказано ]
```

Процедура **рассмотреть**, выдающая ответы в такой форме, показана на рис. 14.16. Она обращается к предикату

импликация(P0, P, Сила, Вер0, Вер)

соответствующему отношению «мягкой импликации» (см. рис. 14.15). P0 – априорная вероятность события *E*, а P – его апостериорная вероятность. **Сила** – сила импликации, представленная как

сила(N, S)

Вер0 и **Вер** – соответственно априорная и апостериорная вероятности гипотезы *H*.

Следует заметить, что наша реализация очень проста, она обеспечивает только изменение вероятностей при распространении информации по сети вывода и иногда ведет себя недостаточно разумно. Никакого внимания не уделяется отбору для анализа наиболее важной в данный момент информации. В более сложной версии следовало бы направлять процесс поиска ответа в сторону наиболее существенных фактов. Кроме того, необходимо стремиться к тому, чтобы пользователю задавалось как можно меньше вопросов.

Наконец, несколько замечаний относительно новой версии процедуры **ответпольз**. Она будет проще, чем процедура рис. 14.11, так как в запросах, передаваемых пользователю, уже не будет переменных. На этот раз пользователь в качестве ответа введет некоторую вероятность (вместо «да» или «нет»). Если пользователю ничего неизвестно о событии, содержащемся в вопросе, то вероятность этого события не изменится. Пользователь может также задать вопрос «почему» и получить изображение объекта **Трасса** в качестве объяснения. Кроме того, следует разрешить пользователю задавать вопрос: «Какова текущая вероятность моей гипотезы?» Тогда, если он устал вводить новую информацию (или у него мало времени), он может прекратить консультационный сеанс, довольствуясь ответом системы, полученным на основании неполной информации.

```
% Процедура
% рассмотреть( Цель, Трасса, Ответ)
%
% находит степень правдоподобия утверждения "цель это правда".
% Оценка правдоподобия содержится в объекте Ответ. Трасса - это
% цепочка целей-предшественников и правил, которую можно
% использовать в объяснении типа "почему"
рассмотреть( Цель, Трасса, (Цель:Вер) было
                'выведено по ' ПравОтв) :-
  bagof(Прав:если Условие то Цель с Сила,Правила),
                % Все правила, относящиеся к цели
  априори(Цель,Вер0),
                % Априорная вероятность цели
  модиф(Вер0,Правила,Трасса,Вер,ПравОтв).
                % Модифицировать априорные вероятности

рассмотреть( Цель1 и Цель2, Трасса,
             ( Цель1 и Цель2 : Вер было 'выведено из'
             ( Ответ1 и Ответ2) ) :-
  !,
  рассмотреть( Цель1, Трасса, Ответ1),
  рассмотреть( Цель2, Трасса, Ответ2),
  вероятность( Ответ1, В1),
  вероятность( Ответ2, В2),
  мин( В1, В2, Вер).

рассмотреть( Цель1 или Цель2, Трасса,
             (Цель или Цель2:Вер) было 'выведено из'
             '(Ответ1 и Ответ2) ) :-
  !,
  рассмотреть( Цель1, Трасса, Ответ1),
  рассмотреть( Цель2, Трасса, Ответ2),
  вероятность( Ответ1, В1),
  вероятность( Ответ2, В2),
  макс( В1, В2, Вер).

рассмотреть( не Цель, Трасса,
       (не Цель:Вер) было 'выведено из' Ответ) :-
  !,
  рассмотреть( Цель, Трасса, Ответ),
  вероятность( Ответ, В),
  обратить( В, Вер).

рассмотреть(Цель,Трасса,(Цель:Вер) было сказано) :-
  ответпольз(Цель,Трасса,Вер).
                % Ответ, выведенный пользователем
```

```
% Отношение
%
% модиф( Вер0, Правила, Трасса, Вер, ПравОтв)
%
% Существует Цель с апрнорной вероятностью Вер0. Правила имеют
% отношение к утверждению Цель; суммарное влияние этих правил
% (точнее, их условных частей) на Вер0 приводит к тому,
% что Вер0 заменяется на апостернорную вероятность Вер;
% Трасса - список целей-предков и правил, использовавшихся
% при выводе утверждения Цель;
% ПравОтв - результаты анализа условных частей
% правил из списка Правила.

модиф( Вер0, [], Трасса, Вер0, []).
                    % Нет правил - нет модификации

модиф( Вер0,
       [ Прав : если Усл то Цель с Сила | Правила],
       Трасса, Вер, [Прав из Ответ | ПравОтв] ):-
  рассмотреть(Усл,[Цель по Прав | Трасса],Ответ),
                    % Условие из первого правила
  априорн( Усл, В0),
  вероятность( Ответ, В),
  импликация( В0, В, Сила, Вер0, Вер1),
                    % "Мягкая" импликация
  модиф( Вер1, Правила, Трасса, Вер, ПравОтв).
```

Рис. 14.16. Определение степени правдоподобия гипотезы при помощи распространения информации об оценке уверенности по сети вывода.

14.7. Заключительные замечания

Нашу оболочку экспертной системы можно развивать в целом ряде направлений. В данный момент уместно сделать несколько критических замечаний и высказать предложения по усовершенствованию нашей программы.

В нашей программе, являющейся упрощенной реализацией, не уделено достаточного внимания вопросам эффективности. В более эффективной реализации потребовалось бы использовать более сложные струк-

туры данных, ввести индексирование или иерархическую структуризацию множества правил и т. п.

Наша процедура **рассмотреть** подвержена зацикливанию в тех случаях, когда в правилах базы знаний «циклически» упоминается одна и та же цель. Этот недостаток легко исправить, предусмотрев в **рассмотреть** соответствующий контроль, т. е. проверку, не является ли текущая цель частным случаем некоторой цели, уже введенной в состав объекта **Трасса**.

Наше объяснение типа «как» выводит дерево доказательства целиком. В случае больших деревьев, удобнее было бы вывести только верхнюю часть дерева, а затем дать пользователю возможность «гулять» по остальной части дерева по своему желанию. Тогда пользователь смог бы просматривать дерево выборочным образом, используя команды, такие как «Вниз по ветви 1», «Вниз по ветви 2», ..., «Вверх », «Достаточно».

В объяснениях типа «как» и «почему» наша оболочка ссылается на правила, указывая их имена, и не показывает их в явном виде. Необходимо, чтобы во время консультационного сеанса пользователь мог, по желанию, запрашивать те или иные правила и получать их явные изображения.

Известно, что придать диалогу с пользователем естественный характер при помощи умелой постановки вопросов – сложная задача. Наш способ ее решения работает только в определенных пределах и во многих случаях приводит к самым разным проблемам, например:

Это правда: сьюзен летает?
нет.

Это правда: сьюзен летает хорошо?

Конечно же нет, раз она совсем не **летает**! Другой пример:

Есть (еще) решения для: Кто-нибудь летает?
да.
Кто-нибудь = **птица**.

Это правда: альбатрос летает?

Для того, чтобы справиться с подобными нежелательными эффектами, следует ввести в экспертную систе-

му дополнительные отношения между понятиями вместе с механизмами их обработки. Обычно эти новые отношения задают иерархию объектов и их свойств.

Возможно еще одно усовершенствование процедуры взаимодействия с пользователем, предусматривающее планирование оптимальной стратегии постановки вопросов. Целью оптимизации является минимизация количества вопросов, которые необходимо задать пользователю для достижения некоторого окончательного логического заключения. Разумеется, возникнут различные варианты таких стратегий, и то, какая из них окажется оптимальной, будет зависеть от ответов пользователя. Принятие решения о выборе той или иной альтернативной стратегии можно основывать на *априорных* вероятностях, являющихся вероятностными оценками «стоимостей» альтернатив. Величины оценок, возможно, придется пересчитывать после каждого ответа пользователя.

Существует еще одна величина, поддающаяся оптимизации: длина цепочки вывода. Такая оптимизация позволила бы давать более простые объяснения типа «как». Сложность объяснений можно также уменьшить за счет селективного подхода к правилам. Некоторые из правил можно было бы не включать в состав объектов **Трасса** и **Ответ**, порождаемых процедурой **рассмотреть**. С этой целью необходимо указывать в базе знаний, какие из правил «трассируемы», а следовательно, должны появляться в объяснениях, а какие можно опускать.

В «разумной» экспертной системе следует предусмотреть вероятностные механизмы, заставляющие ее концентрировать свое внимание на наиболее правдоподобных гипотезах среди всех конкурирующих между собой гипотез. Такая экспертная система должна запрашивать у пользователя ту информацию, которая позволила бы распознать наилучшую среди наиболее правдоподобных гипотез.

Наша экспертная система была классификационного или «анализирующего» типа, в противоположность системам «синтезирующего» типа, в которых ставится задача *построить* что-либо. В последнем случае результат работы – это план действий, предпринимаемых для выполнения этой задачи, например план действий робота, компьютерная конфигурация, удовлетворяющая заданным требованиям, или форсированная комбинация в шахматах. Наш пример, относящийся к

локализации неисправностей, можно естественным образом расширить, чтобы включить в рассмотрение действия. Например, если система не может прийти к определенному выводу, поскольку приборы выключены, она даст рекомендацию: «Включить лампу 3». Здесь сразу возникнет задача построения оптимального плана: минимизировать число действий, необходимых для достижения окончательного вывода.

Проекты

Завершите программирование нашей оболочки в части, касающейся неопределенной информации (процедура **ответпольз** и другие).
Рассмотрите перечисленные выше критические замечания, а также возможные расширения нашей экспертной системы. Разработайте и реализуйте соответствующие усовершенствования.

Резюме

- Обычно от экспертных систем требуют выполнения следующих функций:

  решение задач в заданной предметной области,
  объяснение процесса решения задач,
  работа с неопределенной и неполной информацией.

- Удобно считать, что экспертная система состоит из двух модулей: оболочки и базы знаний. Оболочка в свою очередь состоит из механизма логического вывода и интерфейса с пользователем.

- При создании экспертной системы необходимо принять решения о выборе формального языка представления знаний, механизма логического вывода, средств взаимодействия с пользователем и способа работы в условиях неопределенности.

- «Если-то»-правила, или продукции являются наиболее часто применяемой формой представления знаний в экспертных системах.

- Оболочка, разработанная в данной главе, интерпретирует «если–то»–правила, обеспечивает выдачу объяснений типа «как» и «почему» и запрашивает у пользователя необходимую информацию.
- Машина логического вывода была расширена для работы с неопределенной информацией.
- В данной главе были обсуждены следующие понятия:

  экспертные системы
  база знаний, оболочка,
  машина логического вывода
  «если–то»–правила, продукции
  объяснения типа «как» и «почему»
  категорические знания, неопределенные знания
  сеть вывода,
  распространение оценок достоверности по сети

Литература

Книга Michie (1979) – это сборник статей, относящихся к различным аспектам экспертных систем и инженерии знаний. Две ранние экспертные системы, оказавшие большое влияние на развитие этой области, MYCIN и Prospector, описаны в Shortliffe (1976) и Duda et al (1979). Книга Buchanan and Shortliffe (1984) является хорошим сборником статей, посвященных результатам экспериментов с системой MYCIN. Weiss and Kulikowski (1984) описывают свой практический опыт разработки экспертных систем. Вопрос о работе в условиях неопределенности еще нельзя считать вполне решенным: в статье Quinlan (1983) сравниваются различные подходы к этой проблеме. Способ разработки нашей экспертной системы до некоторой степени аналогичен описанному в Hammond (1984). Некоторые примеры, использовавшиеся в тексте, заимствованы из Winston (1984), Shortliffe (1976), Duda *et al.* (1979), Bratko (1982) и Reiter (1980).

Bratko I. (1982). Knowledge-based problem-solving in AL3. In: *Machine Intelligence 10* (J.E. Hayes, D. Michie, Y.H. Pao, eds.). Ellis Horwood.

Buchanan B.G. and Shortliffe E.H. (1984, eds.). *Rule-based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.* Addison-Wesley.

Duda R., Gasschnig J. and Hart P. (1979). Model design in the Prospector consultant system for mineral exploration. In: *Expert Systems in the Microelectronic Age* (D. Michie, ed.). Edinburgh University Press.

Hammond P. (1984). Micro-PROLOG for Expert Systems. In: *Micro-PROLOG: Programming in Logic* (K.L. Clark, F.G. McCabe, eds.). Prentice-Hall.

Michie D. (1979, ed.). *Expert Systems in the Microelectronic Age.* Edinburgh University Press.

Quinlan J.R. (1983). Inferno: a cautious approach to uncertain reasoning. *The Computer Journal* **26**: 255–270.

Reiter J. (1980). AL/X: An Expert System Using Plausible Inference. Oxford: Intelligent Terminals Ltd.

Shortliffe E. (1976). *Computer-based Medical Consultations: MYCIN.* Elsevier.

Weiss S.M. and Kulikowski C.A. (1984). *A Practical Guide to Designing Expert Systems.* Chapman and Hall.

Winston P. H. (1984). *Artificial Intelligence* (second edition). Addison-Wesley. [Имеется перевод первого издания: Уинстон П. Искусственный интеллект. – М.: Мир, 1980.]

15 ИГРЫ

В этой главе мы рассмотрим методы программирования игр двух лиц с полной информацией (таких, как шахматы). Для игр, представляющих интерес, деревья возможных продолжений слишком велики, чтобы можно было говорить о полном переборе, поэтому необходимы какие-то другие подходы. Один из таких методов, основанный на минимаксном принципе, имеет эффективную реализацию, известную под названием «альфа-бета алгоритм». В дополнение к этому стандартному методу, мы разработаем в этой главе программу на основе Языка Советов (Advice Language), который дает возможность вносить в шахматную программу знания о типовых ситуациях. Этот довольно подробный пример может послужить еще одной иллюстрацией того, насколько хорошо Пролог приспособлен для реализации систем, основанных на знаниях.

15.1. Игры двух лиц с полной информацией

Игры, которые мы собираемся обсуждать в данной главе, относятся к классу так называемых игр двух лиц с полной информацией. Примерами таких игр могут служить шахматы, шашки и т.п. В игре участвуют два игрока, которые ходят по очереди, причем оба они обладают полной информацией о текущей игровой ситуации (это определение исключает большинство карточных игр). Игра считается оконченной, если достигнута позиция, являющаяся согласно правилам игры «терминальной» (конечной), например ма-

товая позиция в шахматах. Правилами игры также устанавливается, каков исход игры в этой терминальной позиции.

Для игр такого рода возможно представление в виде *дерева игры* (или *игрового дерева*). Вершины этого дерева соответствуют ситуациям, а дуги – ходам. Начальная ситуация игры – это корневая вершина; листьями дерева представлены терминальные позиции.

В большинстве игр этого типа возможны следующие исходы: *выигрыш*, *проигрыш* и *ничья*. Мы будем рассматривать здесь игры, имеющие только два возможных исхода – *выигрыш* и *проигрыш*. Игры, в которых возможна ничья, можно упрощенно считать играми с двумя исходами – *выигрыш* и *не-выигрыш*. Двух участников игры мы будем называть «игроком» и «противником». «Игрок» может выиграть в некоторой нетерминальной позиции с ходом игрока («позиции игрока»), если в ней существует *какой-нибудь* разрешенный ход, приводящий к выигрышу. С другой стороны, некоторая нетерминальная позиция с ходом противника («позиция противника») является выигранной для игрока, если *все* разрешенные ходы из этой позиции ведут к позициям, в которых возможен выигрыш. Эти правила находятся в полном соответствии с представлением задач в форме И/ИЛИ-дерева, которое мы обсуждали в гл. 13. Между понятиями, относящимися к И/ИЛИ-деревьям, и понятиями, используемыми в играх, можно установить взаимное соответствие следующим образом:

позиции игры	вершины, задачи
терминальные позиции выигрыша	целевые вершины, тривиально решаемые задачи
терминальные позиции проигрыша	задачи, не имеющие решения
выигранные позиции	задачи, имеющие решение
позиции игрока	ИЛИ-вершины
позиции противника	И-вершины

Очевидно, что аналогичным образом понятия, относящиеся к поиску в И/ИЛИ-деревьях, можно переосмыслить в терминах поиска в игровых деревьях.

Ниже приводится простая программа, которая определяет, является ли некоторая позиция игрока выигранной.

```
выигр( Поз) :-
  терм_выигр( Поз).
                    % Терминальная выигранная позиция

выигр( Поз) :-
  not терм_проигр( Поз),
  ход( Поз, Поз1),                % Разрешенный ход в Поз1
  not ( ход( Поз1, Поз2),
        not выигр( Поз2) ).
    % Ни один из ходов противника не ведет к не-выигрышу
```

Здесь правила игры встроены в предикат **ход(Поз, Поз1)**, который порождает все разрешенные ходы, а также в предикаты **терм_выигр(Поз)** и **терм_проигр(Поз)**, которые распознают терминальные позиции, являющиеся, согласно правилам игры, выигранными или проигранными. В последнем из правил программы, содержащем двойное отрицание (**not**), говорится: не существует хода противника, ведущего к ие выигранной позиции. Другими словами: *все* ходы противника приводят к позициям, выиграиным с точки зрения игрока.

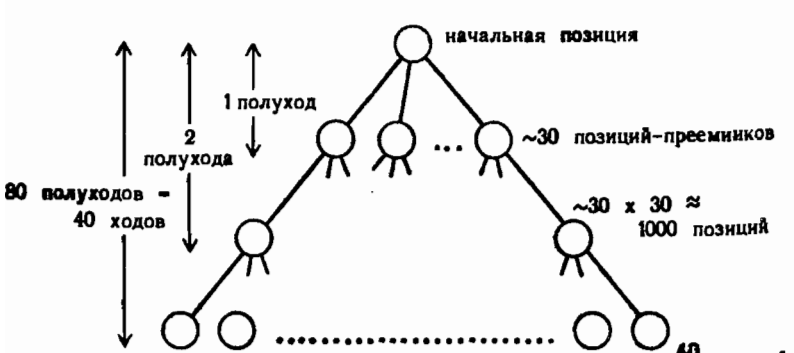


Рис. 15.1. Сложность игровых деревьев в шахматах. Оценки основаны на том, что в каждой шахматной позиции существуют приблизительно 30 разрешенных ходов и что терминальные позиции расположены на глубине 40 ходов. Один ход равен двум полуходам (по одному полуходу с каждой стороны).

Так же, как и аналогичная программа поиска в И/ИЛИ-графах, приведенная выше программа использует стратегию в глубину. Кроме того, в ней не исключается возможность зацикливания на одних и тех же позициях. Попытка устранить этот недостаток может привести к осложнениям, поскольку правила некоторых из игр допускают такое повторение позиций. Правда, разрешение повторять позиции часто носит условный характер, например по шахматным правилам после троекратного повторения позиции может быть объявлена ничья.

Программа, которую мы составили, демонстрирует основные принципы программирования игр. Но практически приемлемая реализация таких сложных игр, как шахматы или го, потребовала бы привлечения значительно более мощных методов. Огромная комбинаторная сложность этих игр делает наш наивный переборный алгоритм, просматривающий дерево вплоть до терминальных игровых позиций, абсолютно непригодным. Этот вывод иллюстрирует (на примере шахмат) рис. 15.1: пространство поиска имеет астрономические размеры – около 10^{120} позиций. Можно возразить, что в дереве на рис. 15.1 встречаются одинаковые позиции. Однако было показано, что число различных позиций дерева поиска находится далеко за пределами возможностей вычислительных машин обозримого будущего.

Проект

Напишите программу для какой-нибудь простой игры (такой, как *ним*), использующую упрощенный алгоритм поиска в И/ИЛИ-дереве.

15.2. Минимаксный принцип

Для игр, представляющих интерес, полный просмотр игрового дерева невозможен, поэтому были разработаны другие методы, предусматривающие просмотр только части дерева игры. Среди этих методов

существует страндартный метод поиска, используемый в игровых (особенио в шахматных) программах и основанный на *минимаксном* принципе. Дерево игры просматривается только вплоть до некоторой глубины (обычно на несколько ходов), а затем для всех концевых вершин дерева поиска вычисляются оценки при помощи некоторой оценочной функции. Идея состоит в том, чтобы, получив оцеики этих терминальных поисковых вершин, не продвигаться дальше и получить тем самым экономию времени. Далее, оценки терминальных позиций распространяются вверх по дереву поиска в соответствии с минимаксным прииципом. В результате все вершины дерева поиска получают свои оценки. И наконец, игровая программа, участвующая в некоторой реальной игре, делает свой ход – ход, ведущий из исходной (корневой) позиции в наиболее перспективного (с точки зрения оценки) ее преемника.

Обратите внимание на то, что мы здесь делаем определенное различие между «деревом игры» и «деревом поиска». Дерево поиска – это только часть дерева игры (его верхняя часть), т. е. та его часть, которая была явным образом порождеиа в процессе поиска. Таким образом, терминальные поисковые позиции совсем не обязательно должиы совпадать с терминальными позициями самой игры.

Очень многое зависит от оценочной функции, которая для большинства игр, представляющих интерес, является приближенной эвристической оценкой шансов на выигрыш одного из участников игры. Чем выше оценка, тем больше у него шаисов выиграть и чем ниже оценка, тем больше шансов иа выигрыш у его противника. Поскольку один из участников игры всегда стремится к высоким оценкам, а другой – к нпзким, мы дадим им имена МАКС и МИН соответственно. МАКС всегда выбирает ход с максимальной оценкой; в противоположность ему МИН всегда выбирает ход с минимальной оценкой. Пользуясь этим принципом (*минимаксным* принципом) и зная значения оценок для всех вершин «подножья» дерева поиска, можно определить оценки всех остальных вершин дерева. На рис. 15.2 показано, как это делается. На этом рисунке видно, что уровни позиций с ходом МАКС'а чередуются с уровнями позиций с ходом МИН'а. Оценки вершин нижнего уровня определяются при помощи оценочной фуикции. Оценки всех внутренних вершин

можно определить, двигаясь снизу вверх от уровня к уровню, пока мы не достигнем корневой вершины. В результате, как видно из рис. 15.2, оценка корня оказывается равной 4, и, соответственно, лучшим ходом МАКС'а из позиции a – a–b. Лучший ответ МИН'а на этот ход – b–d, и т.д. Эту последовательность ходов называют также *основным вариантом*. Основной вариант показывает, какова «минимаксно-оптимальная» игра для обоих участников. Обратите внимание на то, что оценки всех позиций, входящих в основной вариант, совпадают.

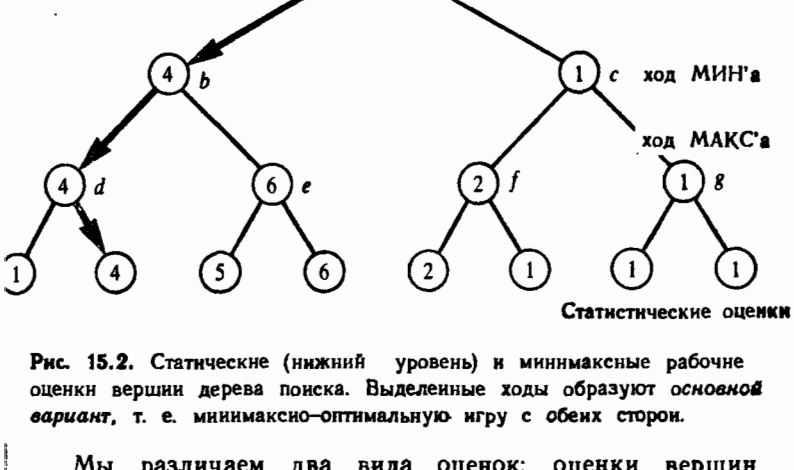


Рис. 15.2. Статические (нижний уровень) и минимаксные рабочие оценки вершин дерева поиска. Выделенные ходы образуют *основной вариант*, т. е. минимаксно-оптимальную игру с обеих сторон.

Мы различаем два вида оценок: оценки вершин нижнего уровня и оценки внутренних вершин (рабочие оценки). Первые из них называются также «статическими», так как они вычисляются при помощи «статической» оценочной функции, в противоположность рабочим оценкам, получаемым «динамически» при распространении статических оценок вверх по дереву.

Правила распространения оценок можно сформулировать следующим образом. Будем обозначать статическую оценку позиции P через $v(P)$, а ее рабочую оценку – через $V(P)$. Пусть $P_1, \ldots, P_n$ – разрешенные преемники позиции P. Тогда соотношения меж-

ду статическими и рабочими оценками можно записать так:

$V(P) = v(P)$

если P – терминальная позиция дерева поиска (n=0)

$V(P) = \max_i V(P_i)$

если P – позиция с ходом МАКС'а

$V(P) = \min_i V(P_i)$

если P – позиция с ходом МИН'а

```
% Минимаксная процедура: минимакс( Поз, ЛучшПоз, Оц)
% Поз – позиция, Оц – ее минимаксная оценка;
% лучший ход из Поз ведет в позицию ЛучшПоз

минимакс( Поз, ЛучшПоз, Оц) :-
  ходы( Поз, СписПоз), !,
                  % СписПоз – список разрешенных ходов
  лучш( СписПоз, ЛучшПоз, Оц);
  стат_оц( Поз, Оц).          % Поз не имеет преемников

лучш( [Поз], Поз, Оц) :-
  минимакс( Поз, _, Оц), !.

лучш( [Поз1 | СписПоз], ЛучшПоз, ЛучшОц) :-
  минимакс( Поз1, _, Оц1),
  лучш( СписПоз, Поз2, Оц2),
  выбор( Поз1, Оц1, Поз2, Оц2, ЛучшПоз, ЛучшОц).

выбор( Поз0, Оц0, Поз1, Оц1, Поз0, Оц0) :-
  ход_мина( Поз0), Оц > Оц1, !;
  ход_макса( Поз0), Оц < Оц1, !.

выбор( Поз0, Оц0, Поз1, Оц1, Поз1, Оц1).
```

Рис. 15.3. Упрощенная реализация минимаксного принципа.

Программа на Прологе, вычисляющая минимаксную рабочую оценку для некоторой заданной позиции, показана на рис. 15.3. Основное отношение этой программы –

минимакс(Поз, ЛучшПоз, Оц)

где **Оц** – минимаксная оценка позиции **Поз**, а **ЛучшПоз** – наилучшая позиция-преемник позиции **Поз** (лучший ход, позволяющий достигнуть оценки **Оц**). Отношение

ходы(Поз, СписПоз)

задает разрешенные ходы игры: **СписПоз** – это список разрешенных позиций-преемников позиции **Поз**. Предполагается, что цель **ходы** имеет неуспех, если **Поз** является терминальной поисковой позицией (листом дерева поиска). Отношение

лучш(СписПоз, ЛучшПоз, ЛучшОц)

выбирает из списка позиций-кандидатов **СписПоз** «наилучшую» позицию **ЛучшПоз**. **ЛучшОц** – оценка позиции **ЛучшПоз**, а следовательно, и позиции **Поз**. Под «наилучшей» оценкой мы понимаем либо максимальную, либо минимальную оценку, в зависимости от того, с чьей стороны ожидается ход.

15.3. Альфа–бета алгоритм: эффективная реализация минимаксного принципа

Программа, показанная на рис. 15.3, производит просмотр в глубину дерева поиска, систематически обходя *все* содержащиеся в нем позиции вплоть до терминальных; она вычисляет статические оценки *всех* терминальных позиций. Как правило, для того, чтобы получить правильную минимаксную оценку корневой вершины, совсем не обязательно проделывать эту работу полностью. Поэтому алгоритм поиска можно сделать более экономным. Его можно усовершенствовать, используя следующую идею. Предположим, что у нас есть два варианта хода. Как только мы узнали, что один из них явно хуже другого, мы можем принять правильное решение, не выясняя, на сколько *в точности* он хуже. Давайте используем этот принцип для сокращения дерева поиска рис. 15.2. Процесс поиска протекает следующим образом:

(1) Начинаем с позиции *a*.
(2) Переходим к *b*.

(3) Переходим к *d*.

(4) Берем максимальную из оценок преемников позиции *d*, получаем *V(d)* = 4.

(5) Возвращаемся к *b* и переходим к *e*.

(6) Рассматриваем первого преемника позиции *e* с оценкой 5. В этот момент МАКС (который как раз и должен ходить в позиции *e*) обнаруживает, что ему гарантирована в позиции *e* оценка не меньшая, чем 5, независимо от оценок других (возможно, более предпочтительных) вариантов хода. Этого вполне достаточно для того, чтобы МИН, даже не зная точной оценки позиции *e*, понял, что для него в позиции *b* ход в *e* хуже, чем ход в *d*.

На основании приведенного выше рассуждения мы можем пренебречь вторым преемником позиции *e* и приписать *e приближенную* оценку 5. Приближенный характер этой оценки не окажет никакого влияния на оценку позиции *b*, а следовательно, и позиции *a*.

На этой идее основан знаменитый *альфа–бета алгоритм*, предназначенный для эффективной реализации минимаксного принципа. На рис. 15.4 показан результат работы альфа–бета алгоритма, примененного к нашему дереву рис. 15.2. Из рис. 15.4 видно, что некоторые из рабочих оценок стали приближениями. Однако этих приближенных оценок оказалось достаточно для того, чтобы определить точную оценку корневой позиции. Сложность поиска уменьшилась до пяти обращений к оценочной функции по сравнению с восемью обращениями (в первоначальном дереве поиска рис. 15.2).

Как уже говорилось раньше, ключевая идея альфа-бета отсечения состоит в том, чтобы найти ход не обязательно лучший, но «достаточно хороший» для того, чтобы принять правильное решение. Эту идею можно формализовать, введя два граничных значения, обычно обозначаемых через *Альфа* и *Бета*, между которыми должна заключаться рабочая оценка позиции. Смысл этих граничных значений таков: *Альфа* – это самое маленькое значение оценки, которое к настоящему моменту уже гарантировано для игрока

МАКС; *Бета* – это самое большое значение оценки, на которое МАКС пока еще может надеяться. Разумеется, с точки зрения МИН'а, *Бета* является самым худшим значением оценки, которое для него уже гарантировано. Таким образом, действительное значение оценки (т. е. то, которое нужно найти) всегда лежит между *Альфа* и *Бета*. Если же стало известно, что оценка некоторой позиции лежит вне интервала *Альфа–Бета*, то этого достаточно для того, чтобы сделать вывод: данная позиция не входит в основной вариант. При этом точное значение оценки такой позиции знать не обязательно, его надо знать только тогда, когда оценка лежит между *Альфа* и *Бета*. «Достаточно хорошую» рабочую оценку $V(P, Альфа, Бета)$ позиции P по отношению к *Альфа* и *Бета* можно определить формально как любое значение, удовлетворяющее следующим ограничениям:

$$V(P, Альфа, Бета) \leq Альфа \quad \text{если} \quad V(P) \leq Альфа$$
$$V(P, Альфа, Бета) = V(P) \quad \text{если} \quad Альфа < V(P) < Бета$$
$$V(P, Альфа, Бета) \geq Бета \quad \text{если} \quad V(P) \geq Бета$$

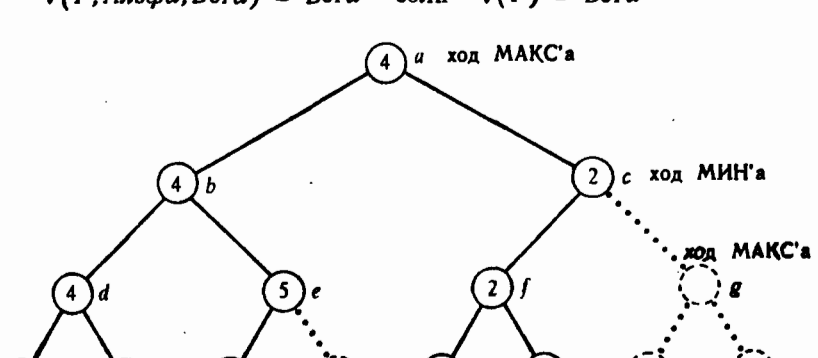


Рис. 15.4. Дерево рис. 15.2 после применения альфа-бета алгоритма. Пунктиром показаны ветви, отсеченные альфа-бета алгоритмом для экономии времени поиска. В результате некоторые из рабочих оценок стали приближенными (вершины *c*, *e*, *f*; сравните с рис. 15.2). Однако этих приближенных оценок достаточно для вычисления точной оценки корневой вершины и построения основного варианта.

Очевидно, что, умея вычислять «достаточно хорошую» оценку, мы всегда можем вычислить точную оценку корневой позиции *P*, установив границы интервала следующим образом:

V(P, -бесконечность, +бесконечность) = V(P)

На рис. 15.5 показана реализация альфа-бета алгоритма в виде программы на Прологе. Здесь основное отношение –

альфабета(Поз, Альфа, Бета, ХорПоз, Оц)

где **ХорПоз** – преемник позиции **Поз** с «достаточно хорошей» оценкой **Оц**, удовлетворяющей всем указанным выше ограничениям:

Оц = *V(Поз, Альфа, Бета)*

Процедура

прибл_лучш(СписПоз, Альфа, Бета, ХорПоз, Оц)

находит достаточно хорошую позицию **ХорПоз** в списке позиций **СписПоз**; **Оц** – приближенная (по отношению к **Альфа** и **Бета**) рабочая оценка позиции **ХорПоз**.

Интервал между *Альфа* и *Бета* может сужаться (но не расширяться!) по мере углубления поиска, происходящего при рекурсивных обращениях к альфа-бета процедуре. Отношение

нов_границы(Альфа, Бета, Поз, Оц, НовАльфа, НовБета)

определяет новый интервал **(НовАльфа, НовБета)**. Он всегда уже, чем старый интервал **(Альфа, Бета)**, или равен ему. Таким образом, чем глубже мы оказываемся в дереве поиска, тем сильнее проявляется тенденция к сжатию интервала *Альфа–Бета*, и в результате оценивание позиций на более глубоких уровнях происходит в условиях более тесных границ. При более узких интервалах допускается большая степень «приблизительности» при вычислении оценок, а следовательно, происходит больше отсечений ветвей дерева. Возникает интересный вопрос: насколько велика экономия, достигаемая альфа-бета алгоритмом по сравнению с программой минимаксного полного перебора рис. 15.3?

Эффективность альфа-бета процедуры зависит от порядка, в котором просматриваются позиции. Всегда лучше первыми рассматривать самые сильные ходы с каждой из сторон. Легко показать на примерах, что

```
% Альфа-бета алгоритм
альфабета( Поз, Альфа, Бета, ХорПоз, Оц) :-
    ходы( Поз, СписПоз), !,
    прибл_лучш( СписПоз, Альфа, Бета, ХорПоз, Оц);
    стат_оц( Поз, Оц).

прибл_лучш([Поз | СписПоз], Альфа, Бета, ХорПоз, ХорОц):-
    альфабета( Поз, Альфа, Бета, _, Оц),
    дост_хор(СписПоз, Альфа, Бета, Поз, Оц, ХорПоз, ХорОц).

дост_хор( [], _, _, Поз, Оц, Поз, Оц) :- !.
                        % Больше нет кандидатов

дост_хор( _, Альфа, Бета, Поз, Оц, Поз, Оц) :-
    ход_мина( Поз), Оц > Бета, !;
                        % Переход через верхнюю границу
    ход_макса( Поз), Оц < Альфа, !.
                        % Переход через нижнюю границу

дост_хор(СписПоз, Альфа, Бета, Поз, Оц, ХорПоз, ХорОц):-
    нов_границы(Альфа, Бета, Поз, Оц, НовАльфа, НовБета),
                        % Уточнить границы
    прибл_лучш(СписПоз, НовАльфа, НовБета, Поз1, Оц1),
    выбор( Поз, Оц, Поз1, Оц1, ХорПоз, ХорОц).

нов_границы( Альфа, Бета, Поз, Оц, Оц, Бета) :-
    ход_мина( Поз), Оц > Альфа, !.
                        % Увеличение нижней границы

нов_границы( Альфа, Бета, Поз, Оц, Альфа, Оц) :-
    ход_макса( Поз), Оц < Бета, !.
                        % Уменьшение верхней границы

нов_границы( Альфа, Бета, _, _, Альфа, Бета).

выбор( Поз, Оц, Поз1, Оц1, Поз, Оц) :-
    ход_мина( Поз), Оц > Оц1, !;
    ход_макса( Поз), Оц < Оц1, !.

выбор( _, _, Поз1, Оц1, Поз1, Оц1).
```

Рис. 15.5. Реализация альфа-бета алгоритма.

возможен настолько неудачный порядок просмотра, что альфа-бета алгоритму придется пройти через *все* вершины, которые просматривались минимаксным алгоритмом полного перебора. Это означает, что в худ-

шем случае альфа-бета алгоритм не будет иметь никаких преимуществ. Однако, если порядок просмотра окажется удачным, то экономия может быть значительной. Пусть N – число терминальных поисковых позиций, для которых вычислялись статические оценки алгоритмом минимаксного полного перебора. Было доказано, что в лучшем случае, когда самые сильные ходы всегда рассматриваются первыми, альфа-бета алгоритм вычисляет статические оценки только для N позиций.

Этот результат имеет один практический аспект, связанный с проведением турниров игровых программ. Шахматной программе, участвующей в турнире, обычно дается некоторое определенное время для вычисления очередного хода, и доступная программе глубина поиска зависит от этого времени. Альфа-бета алгоритм сможет пройти при поиске *вдвое глубже* по сравнению с минимаксным полным перебором, а опыт показывает, что применение той же оценочной функции, но на большей глубине приводит к более сильной игре.

Экономию, получаемую за счет применения альфа-бета алгоритма, можно также выразить в терминах более эффективного коэффициента ветвления дерева поиска (т. е. числа ветвей, исходящих из каждой внутренней вершины). Пусть игровое дерево имеет единый коэффициент ветвления, равный b. Благодаря эффекту отсечения альфа-бета алгоритм просматривает только некоторые из существующих ветвей и тем самым уменьшает коэффициент ветвления. В результате коэффициент b превратится в $\sqrt{b}$ (в лучшем случае). В шахматных программах, использующих альфа-бета алгоритм, достигается коэффициент ветвления, равный 6, при наличии 30 различных вариантов хода в каждой позиции. Впрочем, на этот результат можно посмотреть и менее оптимистично: несмотря на применение альфа-бета алгоритма, после каждого продвижения вглубь на один полуход число терминальных поисковых вершин увеличивается примерно в 6 раз.

Проект

Рассмотрите какую-нибудь игру двух лиц (например, какой-нибудь нетривиальный вариант крестиков-ноликов). Напишите отношения, задающие правила

этой игры (разрешенные ходы и терминальные позиции). Предложите статическую оценочную функцию, пригодную для использования в игровой программе, основанной на альфа-бета алгоритме.

15.4. Минимаксные игровые программы: усовершенствования и ограничения

Минимаксный принцип и альфа-бета алгоритм лежат в основе многих удачных игровых программ, чаще всего шахматных. Общая схема подобной программы такова: произвести альфа-бета поиск из текущей позиции вплоть до некоторого предела по глубине (диктуемого временными ограничениями турнирных правил). Для оценки терминальных поисковых позиций использовать подобранную специально для данной игры оценочную функцию. Затем выполнить на игровой доске наилучший ход, найденный альфа-бета алгоритмом, принять ответный ход противника и запустить тот же цикл с начала.

Таким образом, две основных составляющих игровой программы – это альфа-бета алгоритм и эвристическая оценочная функция. Для того, чтобы создать действительно хорошую программу для такой сложной игры, как шахматы, необходимо внести в эту базовую схему много различных усовершенствований. Ниже приводится краткое описание некоторых из стандартных приемов.

Многое зависит от оценочной функции. Если бы мы располагали абсолютно точной оценочной функцией, мы могли бы ограничить поиск рассмотрением только непосредственных преемников текущей позиции, фактически исключив перебор. Но для таких игр, как шахматы, любая оценочная функция, имеющая практически приемлемую вычислительную сложность, по необходимости будет всего лишь эвристической оценкой. Такая оценка базируется на «статических» свойствах позиции (например, на количестве фигур) и в одних позициях работает надежнее, чем в других. Допустим, например, что мы имеем именно такую оценочную функцию, основанную на соотношении

материала, и представим себе позицию, в которой у белых лишний конь. Ясно, что оценка будет в пользу белых. Здесь все в порядке, если позиция «спокойная» и черные не располагают какой-либо сильной угрозой. Но, с другой стороны, если на следующем ходу черные могут взять белого ферзя, то такая оценка может привести к фатальному просмотру из-за своей неспособности к *динамическому* восприятию позиции. Очевидно, что в спокойных позициях мы можем доверять такой статической оценке в большей степени, чем в активных позициях, когда с каждой из сторон имеются непосредственные угрозы взятия фигур. Поэтому статическую оценку следует использовать только для спокойных позиций. Что же касается активных позиций, то здесь существует такой стандартный прием: следует продолжить поиск из активной позиции за пределы ограничения по глубине и продолжать его до тех пор, пока не встретится спокойная позиция. В частности, таким образом производится просчет размена фигур в шахматах.

Еще одно усовершенствование – *эвристическое отсечение* (ветвей). Целью его является достижение большей предельной глубины поиска за счет отбрасывания менее перспективных продолжений. Этот метод позволяет отсекать ветви в дополнение к тем, которые отсекаются самим альфа-бета алгоритмом. В связи с этим возникает риск пропустить какое-нибудь хорошее продолжение и неправильно вычислить минимаксную оценку.

Существует еще один прием, называемый *последовательным углублением*. Программа многократно выполняет альфа-бета поиск сначала до некоторой небольшой глубины, а затем, увеличивая предел по глубине при каждой итерации. Процесс завершается, когда истекает время, отведенное для вычисления очередного хода. Выполняется наилучший ход, найденный при наибольшей глубине, достигнутой программой. Этот метод имеет следующие преимущества:

- он облегчает контроль времени: в момент, когда время истекает, всегда имеется некоторый ход – лучший из всех, найденных к настоящему моменту;
- минимаксные оценки, вычисленные во время

предыдущей итерации, можно использовать для предварительного упорядочивания позиций в следующей итерации, что помогает альфа-бета алгоритму следовать стратегии «самые сильные ходы – первыми».

Метод последовательного углубления влечет за собой некоторые накладные расходы (из-за повторного поиска в верхней части игрового дерева), но они незначительны по сравнению с суммарными затратами.

Для наших программ, основанных на описанной выше схеме, существует проблема, известная как «эффект горизонта». Представьте себе шахматную позицию, в которой программе грозит неминуемая потеря коня, однако эту потерю можно отложить, пожертвовав какую-либо менее ценную фигуру, скажем пешку. Эта немедленная жертва сможет отодвинуть потерю коня за пределы доступной глубины поиска (за «горизонт» программы). Не видя грозящей опасности, программа отдаст предпочтение продолжению с жертвой пешки, чтобы избежать быстрой гибели своего коня. В действительности программа потеряет *обе* фигуры – и пешку (без необходимости), и коня. Эффект горизонта можно несколько смягчить за счет углубления поиска вплоть до спокойных позиций.

Существует, однако, более фундаментальное ограничение на возможности минимаксных игровых программ, проистекающее из той ограниченной формы представления знаний, которая в них используется. Это становится особенно заметным при сравнении лучших шахматных программ с шахматными мастерами (людьми). Хорошая программа просматривает миллионы (и даже больше) позиций, прежде чем принимает решение об очередном ходе. Психологические опыты показали, что шахматные мастера, как правило, просматривают десятки (максимум, несколько сотен) позиций. Несмотря на эту явно меньшую производительность, мастера-шахматисты обыгрывают программы без особых усилий. Преимущество их состоит в их знаниях, значительно превосходящих знания шахматных программ. Игры между машинами и сильными шахматистами показали, что огромное превосходство в вычислительной мощности не способно скомпенсировать недостаток знаний.

Знания в минимаксных игровых программах имеют следующие три основные формы:

- оценочная функция
- эвристики для отсечения ветвей
- эвристики для распознавания спокойных позиций

Оценочная функция сводит все разнообразные аспекты игровой ситуации к одному числу, и это упрощение может нанести вред. В противоположность этому хороший игрок обладает пониманием позиции, охватывающим многие «измерения». Вот пример из области шахмат: оценочная функция оценивает позицию как равную и выдает значение 0. Оценка той же позиции, данная мастером-шахматистом, может быть значительно более информативной, а также может указывать на дальнейший ход игры, например: у белых лишняя пешка, но черные имеют неплохие атакующие возможности, что компенсирует материальный перевес, следовательно, шансы равны.

Минимаксные шахматные программы часто хорошо проявляют себя в острой тактической борьбе, когда решающее значение имеет точный просчет форсированных вариантов. Их слабости обнаруживаются в спокойных позициях, так как они не способны к долговременному планированию, преобладающему при медленной, стратегической игре. Из-за отсутствия плана создается внешнее впечатление, что программа все время перескакивает с одной идеи на другую. Особенно это заметно в эндшпилях.

В оставшейся части главы мы рассмотрим еще один подход к программированию игр, основанный на внесении в программу знаний о типовых ситуациях при помощи так называемых «советов».

15.5. Знания о типовых ситуациях и механизм «советов»

В этом разделе рассматривается метод представления знаний о конкретной игре с использованием семейства Языков Советов. Языки Советов (Advice Languages) дают возможность пользователю декларативным способом описывать, какие идеи следует использовать в тех или иных типовых ситуациях. Идеи форму-

лируются в терминах целей и средств, необходимых для их достижения. Интерпретатор Языка Советов определяет при помощи перебора, какая идея «работает» в данной ситуации.

15.5.1. Цели и ограничения на ходы

Основное понятие Языка Советов – «элементарный совет». Элементарный совет содержит указание о том, что следует делать (или *пытаться* делать) в некоторой типовой ситуации. Совет выражается в терминах тех *целей*, которые необходимо достичь, и тех *средств*, которые следует применять для этого. Мы называем участников игры «игроком» и «противником»; совет всегда относится к «игроку». Каждый элементарный совет имеет следующие четыре составные части:

- *главная цель*: цель, к которой нужно стремиться;
- *цель-поддержка*: цель, которая должна постоянно удовлетворяться в процессе достижения главной цели;
- *ограничения на ходы игрока*: предикат, определяющий некоторое подмножество ходов из всех разрешенных ходов игрока (ходы, представляющие интерес с точки зрения достижения указанных целей).
- *ограничения на ходы противника*: предикат, выбирающий ходы, которые должен рассмотреть противник (ходы, препятствующие достижению указанных целей).

Рассмотрим, например, шахматный эндшпиль «король и пешка против короля». Здесь применима следующая очевидная идея: провести пешку в ферзи, продвигая ее вперед. В форме совета это выражается так:

- *главная цель*: провести пешку;
- *цель-поддержка*: не потерять пешку;
- *ходы игрока*: продвигать пешку;
- *ходы противника*: приближаться королем к пешке.

15.5.2. Выполнимость совета

Мы говорим, что элементарный совет *выполним* в данной позиции, если игрок может форсированным образом достигнуть главной цели, указанной в совете, при условии, что:

(1) ни разу не нарушается цель-поддержка;
(2) все ходы игрока удовлетворяют наложенным на них ограничениям;
(3) противнику разрешено делать только те ходы, которые предусмотрены соответствующими ограничениями.

С выполнимостью элементарного совета связано понятие *форсированного дерева*. Форсированное дерево задает детальную стратегию, которая гарантирует достижение главной цели при выполнении всех ограничений, содержащихся в элементарном совете. Таким образом, форсированное дерево указывает, как именно должен ходить игрок при любых ответах противника. Более точно, форсированное дерево T для заданной позиции P и элементарного совета A есть такое поддерево дерева игры, что

- корень дерева T – позиция P;
- все позиции из T удовлетворяют цели-поддержке;
- все терминальные позиции из T удовлетворяют главной цели (что, однако, неверно ни для одной внутренней вершины);
- для каждой внутренней позиции игрока в дереве T указан только один ход, причем он удовлетворяет ограничениям на ходы игрока;
- из каждой внутренней позиции противника исходят все ходы противника (удовлетворяющие соответствующим ограничениям).

Каждый элементарный совет можно рассматривать как описание некоторой небольшой специальной игры, имеющей следующие правила. Участникам игры разрешено ходить в пределах ограничений, наложенных на их ходы; позиция, не удовлетворяющая цели-поддержке, считается выигрышем «противника». Не-

терминальная позиция считается выигранной с точки зрения игрока, если данный элементарный совет в ней выполним. Таким образом, для того, чтобы выиграть в этой игре, игрок должен следовать стратегии, задаваемой форсированным деревом.

15.5.3. Правила и таблицы советов

В Языках Советов отдельные элементарные советы объединяются в полную схему представления знаний, имеющую следующую иерархическую структуру. Элементарный совет является частью «если-то»-правила. Набор «если-то»-правил образует *таблицу советов*. Множество таблиц советов имеет структуру иерархической сети. Каждая таблица советов выполняет роль эксперта в своей узкой области и работает с какой-нибудь специфической подзадачей. Примером такого специализированного эксперта может служить таблица советов, содержащая знания о том, как поставить мат королем и ладьей. Эта таблица вызывается в том случае, когда в процессе игры возникает соответствующее окончание.

Мы рассмотрим здесь упрощенную версию Языка Советов, допускающую только одну таблицу советов. Будем называть эту версию Язык Советов 0 или, для краткости, AL0 (Advice Language 0). Ниже описывается структура языка AL0, синтаксически специально приспособленная для удобной реализации на Прологе.

Программа на AL0 называется *таблицей советов*. Таблица советов представляет из себя *упорядоченное* множество «если-то»-правил. Каждое правило имеет вид:

ИмяПравила: если Условие то СписокСоветов

Условие – это логическое выражение, состоящее из имен предикатов, соединенных между собой логическими связками **и**, **или**, **не**. **СписокСоветов** – список имен элементарных советов. Приведем пример правила под названием «правило_края» из окончания «король и ладья против короля»:

```
правило_края:
  если король_противника_на_краю и короли_рядом
```

```
то [мат_2,потеснить,приблизиться,
                    сохранить_простр,отделить].
```

В этом правиле говорится: если в текущей позиции король противника находится на краю доски, а король игрока расположен близко к королю противника (точнее, расстояние между королями меньше четырех клеток), то попытаться выполнить в указанном порядке предпочтения следующие советы: «мат_2», «потеснить», «приблизиться», «сохранить_простр», «отделить». Элементарные советы расположены в порядке убывания их «притязаний» на успех: сначала попытаться поставнть мат в два хода, если не получится – «потеснить» короля противника в угол и т.д. Обратите внимание на то, что при соответствующем определенни операторов наше правило станет синтаксически корректным предложеиием Пролога.

Для представления элементарных советов в виде прологовских предложений предназначен еще один формат:

```
совет( ИмяСовета,
       ГлавнаяЦель:
       ЦельПоддержка:
       ХодыИгрока:
       ХодыПротивника).
```

Цели представляются как выражения, состоящие из имен предикатов и логических связок **и**, **или**, **не**. Ограничения на ходы сторон – это тоже выражения, состоящие из имен предикатов и связок **и** и **затем**: связка **и** имеет обычный логический смысл, а **затем** задает порядок. Например, ограничение, имеющее внд

```
Огр1 затем Огр2
```

означает: сначала рассмотреть ходы, удовлетворяющие ограничению **Огр1**, а затем – ходы, удовлетворяющие **Огр2**.

Например, элементарный совет, относящийся к мату в два хода в окоичании «король и ладья против короля», записанный в такой синтаксической форме, имеет вид:

```
совет( мат_2,
       мат:
       не потеря_ладьи:
       (глубина = 0) и разреш затем
```

(глубина = 2) и ход_шах :
(глубина = 1) и разреш).

Здесь главная цель – мат, цель-поддержка – не потеря_ладьи. Ограничение на ходы игрока означает: на глубине 0 (т. е. в текущей позиции) попробовать любой разрешенный ход и затем на глубине 2 (следующий ход игрока) пробовать только ходы с шахом. Глубина измеряется в полуходах. Ограничение на ходы противника: любой разрешенный ход на глубине 1.

В процессе игры таблица советов используется многократно вплоть до окончания игры, при этом выполняется следующий основной цикл: построить форсированное дерево, затем играть в соответствии с этим деревом, пока не произойдет выход из него; построить другое форсированное дерево и т.д. Форсированное дерево строится каждый раз таким образом: берется текущая позиция **Поз** и просматриваются одно за другим все правила таблицы советов; для каждого правила сопоставляется **Поз** с предварительным условием этого правила и просмотр прекращается, когда будет обнаружено правило, для которого **Поз** удовлетворяет предварительному условию. В этом случае надо рассмотреть список советов найденного правила: обработать элементарные советы один за другим, пока не будет построено форсированное дерево, представляющее собой детальную стратегию игры в этой позиции.

Следует обратить внимание на существенность того порядка, в котором перечисляются правила в таблице советов. Правило, которое реально используется, – это первое из тех правил, предварительные условия которых согласуются с текущей позицией. Для любой возможной позиции должно существовать по крайней мере одно такое правило. Из него берется список советов, и первый из выполнимых советов списка используется в игре.

Таким образом, таблица советов – это программа в высшей степени непроцедурного характера. Интерпретатор языка АЛО принимает на входе некоторую позицию, а затем, «исполняя» таблицу советов, строит форсированное дерево, определяющее стратегию игры в этой позиции.

15.6. Программа на языке AL0 для игры в шахматном эндшпиле

При реализации какой-либо игровой программы на языке AL0 ее можно для удобства разбить на три модуля:

(1) интерпретатор языка AL0,
(2) таблица советов на языке AL0,
(3) библиотека предикатов, используемых в таблице советов (в том числе предикаты, задающие правила игры).

Эта структура соответствует обычной структуре системы, осиованной на знаниях:

- Интерпретатор AL0 выполняет функцию машины логического вывода.
- Таблица советов вместе с библиотекой предикатов образует базу зиаиий.

15.6.1. Миниатюриый интерпретатор языка *AL0*

Реализация на Прологе миииатюриого, не зависящего от конкретиой игры интерпретатора языка AL0 показана на рис. 15.6. Эта программа осуществляет также взаимодействие с пользователем во время игры. Центральная задача этой программы – использовать знания, записанные в таблице советов, то есть интерпретировать программу иа языке советов AL0 с целью построения форсированиых деревьев и их «исполнения» в процессе игры. Базовый алгоритм порождения форсированных деревьев аналогичен поиску с предпочтением в И/ИЛИ-графах гл. 13, при этом форсированное дерево соответствует решающему И/ИЛИ-дереву. Этот алгоритм также напоминает алгоритм построеиия решающего дерева ответа на вопрос пользователя, примеиявшийся в оболочке экспертной системы (гл. 14).

Программа на рис. 15.6 составлена в предположении, что она играет белыми, а ее противник – черными. Программа запускается процедурой

```
% Миниатюрный интерпретатор языка ALO
%
% Эта программа играет, начиная с заданной позиции,
% используя знания, записанные на языке ALO

:- op( 200, xfy, :).
:- op( 220, xfy, ..).
:- op( 185, fx, если).
:- op( 190, xfx, то).
:- op( 180, xfy, или).
:- op( 160, xfy, и).
:- op( 140, fx, не).


игра( Поз) :-                          % Играть, начиная с Поз
  игра( Поз, nil).
                    % Начать с пустого форсированного дерева
игра( Поз, ФорсДер) :-
  отобр( Поз),
  ( конец_игры( Поз),                  % Конец игры?
      write( 'Конец игры'), nl, !;
    сделать_ход( Поз, ФорсДер, Поз1, ФорсДер1), !,
      игра( Поз1, ФорсДер1) ).

% Игрок ходит в соответствии с форсированным деревом
сделать_ход( Поз, Ход .. ФДер1, Поз1, ФДер1) :-
  чей_ход( Поз, б),                    % Программа играет белыми
  разрход( Поз, Ход, Поз1),
  показать_ход( Ход).

% Прием хода противника
сделать_ход( Поз, ФДер, Поз1, ФДер1) :-
  чей_ход( Поз, ч),
  write( 'Ваш ход:'),
  read( Ход),
  ( разрход( Поз, Ход, Поз1),
      поддер( ФДер, Ход, ФДер1), !;
                    % Вниз по форс. дереву
    write( 'Неразрешенный ход'), nl,
      сделать_ход( Поз, ФДер, Поз1, ФДер1) ).

% Если текущее форсированное дерево пусто, построить новое
сделать_ход( Поз, nil, Поз1, ФДер1) :-
```

```
  чей_ход( Поз, б),
  восст_глуб( Поз, Поз0),
                % Поз0 = Поз с глубиной 0
  стратегия( Поз0, ФДер), !,
                % Новое форсированное дерево
  сделать_ход( Поз0, ФДер, Поз1, ФДер1).

% Выбрать форсированное поддерево, соответствующее Ход'у
поддер( ФДеревья, Ход, Фдер) :-
  принадлежит( Ход .. Фдер, ФДеревья), !.

поддер( _, _, nil).

стратегия( Поз, ФорсДер) :-
                % Найти форс. дерево для Поз
  Прав : если Условие то СписСов,
                % Обращение к таблице советов
  удовл( Условие, Поз, _), !,
                % Сопоставить Поз с предварительным условием
  принадлежит( ИмяСовета, СписСов),
                % По очереди попробовать элем. советы
  nl, write( 'Пробую'), write( ИмяСовета),
  выполн_совет( ИмяСовета, Поз, ФорсДер), !.

выполи_совет( ИмяСовета, Поз, Фдер) :-
  совет( ИмяСовета, Совет),
                % Найти элементарный совет
  выполн( Совет, Поз, Поз, ФДер).
  % "выполн" требует две позиции для сравнивающих предикатов
выполн( Совет, Поз, КориПоз, ФДер) :-
  поддержка( Совет, ЦП),
  удовл( ЦП, Поз, КориПоз),
                % Сопоставить Поз с целью-поддержкой
  выполн1( Совет, Поз, КориПоз, ФДер).

выполн1( Совет, Поз, КорнПоз, nil) :-
  главцель( Совет, ГлЦ),
  удовл( ГлЦ, Поз, КорнПоз), !.
                % Главная цель удовлетворяется

выполн1( Совет, Поз, КорнПоз, Ход .. ФДеревья) :-
  чей_ход( Поз, б), !,          % Программа играет белыми
  ходы_игрока( Совет, ХодыИгрока),
                % Ограничения на ходы игрока
  ход( ХодыИгрока, Поз, Ход, Поз1),
                % Ход, удовлетворяющий ограничению
  выполи( Совет, Поз1, КориПоз, ФДеревья).
```

```
выполн1( Совет, Поз, КорнПоз, ФДеревья) :-
  чей_ход( Поз, ч), !,          % Противнпк играет черными
  ходы_противнпка( Совет, ХодыПр),
  bagof( Ход..Поз1, ход(ХодыПр, Поз, Ход, Поз1), ХПспис),
  выполн_все( Совет, ХПспис, КорнПоз, ФДеревья).
                % Совет выполним во всех преемнпках Поз

выполн_все( _, [], _, []).

выполн_все( Совет, [Ход..Поз | ХПспис], КорнПоз,
                                  [Ход..ФД | ФДД] ) :-
  выполн( Совет, Поз, КорнПоз, ФД),
  выполн_все( Совет, ХПспис, КорнПоз, ФДД).


% Интерпретация главной целн н цели-поддержки:
% цель - это И/ИЛИ/НЕ комбинация имен предикатов

удовл( Цель1 и Цель2, Поз, КорнПоз) :- !,
  удовл( Цель1, Поз, КорнПоз),
  удовл( Цель2, Поз, КорнПоз).

удовл( Цель1 илн Цель2, Поз, КорнПоз) :- !,
  ( удовл( Цель1, Поз, КорнПоз);
    удовл( Цель2, Поз, КорнПоз) ).

удовл( не Цель, Поз, КорнПоз) :- !,
  not удовл( Цель, Поз, КорнПоз ).

удовл( Пред, Поз, КорнПоз) :-
  ( Усл =.. [Пред, Поз];
        % Большинство предикатов не зависит от КорнПоз
    Усл =.. [Пред, Поз, КорнПоз] ),
  call( Усл).

% Интерпретация огранпченнй на ходы

ход( Ходы1 и Ходы2, Поз, Ход, Поз1) :- !,
  ход( Ходы1, Поз, Ход, Поз1),
  ход( Ходы2, Поз, Ход, Поз1).

ход( Ходы1 затем Ходы2, Поз, Ход, Поз1) :- !,
  ( ход( Ходы1, Поз, Ход, Поз1);
    ход( Ходы2, Поз, Ход, Поз1) ).

% Доступ к компонентам элементарного совета

главцель( ГлЦ : _, ГлЦ).

поддержка( ГлЦ : ЦП : _, ЦП).
```

```
ходы_игрока(ГлЦ : ЦП : ХодыИгрока : _, ХодыИгрока).
ходы_противника(ГлЦ : ЦП: ХодыИгр : ХодыПр :_,
                ХодыПр).
принадлежит( X, [X | Спис]).
принадлежит( X, [Y | Спис]) :-
  принадлежит( X, Спис).
```

Рис. 15.6 Миниатюрный интерпретатор языка AL0.

игра(Поз)

где **Поз** – выбранная начальная позиция. Если в позиции **Поз** ходит противник, то программа принимает его ход, в противном случае – «консультируется» с таблицей советов, приложенной к программе, порождает форсированное дерево и делает свой ход в соответствии с этим деревом. Так продолжается до окончания игры, которое обнаруживает предикат конец_игры (например, если поставлен мат).

Форсированное дерево – это дерево ходов, представленное в программе следующей структурой:

Ход..[Ответ1..Фдер1, Ответ2..Фдер2, ...]

Здесь «..» – инфиксный оператор; **Ход** – первый ход «игрока»; **Ответ1**, **Ответ2**, ... – возможные ответы противника; **Фдер1**, **Фдер2**, ... – форсированные поддеревья для каждого из этих ответов.

15.6.2. Программа на языке советов для эндшпиля «король и ладья против короля»

Общий принцип достижения выигрыша королем и ладьей против единственной фигуры противника, короля, состоит в том, чтобы заставить короля отступить к краю доски или, при необходимости, загнать его в угол, а затем поставить мат в несколько ходов. В детальном изложении эта стратегия выглядит так:

Повторять циклически, пока не будет поставлен мат (постоянно проверяя, что не возникла патовая позиция и что нет нападения на не защищенную ладью):

(1) Найти способ поставить королю противника мат в два хода.

(2) Если не удалось, то найти способ уменьшить ту область доски, в которой король противника «заперт» под воздействием ладьи.

(3) Если и это не удалось, то найти способ приблизить своего короля к королю противника.

(4) Если ни один из элементарных советов 1, 2, или 3 не выполним, то найти способ сохранить все имеющиеся к настоящему моменту «достижения» в смысле (2) и (3) (т. е. сделать выжидающий ход).

(5) Если ни одна из целей 1, 2, 3 или 4 не достижима, то найти способ получить позицию, в которой ладья занимает вертикальную или горизонтальную линию, отделяющую одного короля от другого.

Описанные выше принципы реализованы во всех деталях в таблице советов на языке AL0, показанной на рис. 15.7. Эта таблица может работать под управлением интерпретатора рис. 15.6. Рис. 15.8 иллюстрирует смысл некоторых из предикатов, использованных в таблице советов, а также показывает, как эта таблица работает.

В таблице используются следующие предикаты:

Предикаты целей

мат	мат королю противника
пат	пат королю противника
потеря_ладьи	король противника может взять ладью
ладья_под_боем	король противника может напасть

	иа ладью прежде, чем иаш король сможет ее защитить
уменьш_простр	умеиьшилось «жизнеииое пространство» короля противиика, ограиичиваемое ладьей
раздел	ладья занимает вертикальиую или горизонтальиую линию, разделяющую королей
ближе_к_клетке	иаш король приблизился к «критической клетке» (см. рис. 15.9), т.е. маихеттеиовское расстояиие до иее уменьшилось
l_конфиг	«L-конфигурация» (рис. 15.9)
простр_больше_2	«жизнеииое пространство» короля противиика занимает больше двух клеток

Предикаты, ограничивающие ходы

глубиа = N	ход иа глубиие N дерева поиска
разреш	любой разрешеииый ход
ход_шах	ход, объявляющий шах
ход_ладьей	ход ладьей
иет_хода	ии одии ход ие подходит
сиачала_диаг	ход королем, преимущественно по диагоиали

```
% Окончание "король и ладья против короля" на языке AL0

% Правила

правило_края:
    если король_противника_на_краю и короли_рядом
    то [мат_2, потеснить, приблизиться,
        сохранить_простр, отделить_2, отделить_3].
иначе_правило
    если любая_поз
    то [ потесиить, приблизиться, сохраиить_простр,
         отделить_2, отделить_3].
```

```
% Элементарные советы
совет( мат_2,
       мат :
       не потеря_ладья и король_противника_на_краю:
       (глубина = 0) и разреш
       затем (глубина = 2) и ход_шах :
       (глубина = 1) и разреш ).

совет( потеснить,
       уменьш_простр и не ладья_под_боем и
       раздел и не пат :
       не потеря_ладьи :
       (глубина = 0) и ход_ладьей :
       нет_хода ).

совет( приблизиться,
       ближе_к_клетке и не ладья_под_боем и
       (раздел или 1_конфиг) и
       (простр_больше_2 или не наш_король_на_краю):
       не потеря_ладьи :
       (глубина = 0) и сначала_диаг :
       нет_хода ).

совет( сохранить_простр,
       ход_противника и не ладья_под_боем и раздел
       и не_дальше_от_ладьи и
       (простр_больше_2 или не наш_король_на_краю):
       не потеря_ладьи :
       (глубина = 0) и сначала_диаг :
       нет_хода ).

совет( отделить_2,
       ход_противника и раздел и не ладья_под_боем:
       не потеря_ладьи :
       (глубина < 3) и разреш :
       (глубина < 2) и разреш ).

совет( отделить_3,
       ход_противника и раздел и не ладья_под_боем:
       не потеря_ладьи :
       (глубина < 5) и разреш :
       (глубина < 4) и разреш ).
```

Рис. 15.7. Таблица советов на языке ALO для окончания "король и ладья против короля". Таблица состоит из двух правил и шести элементарных советов.

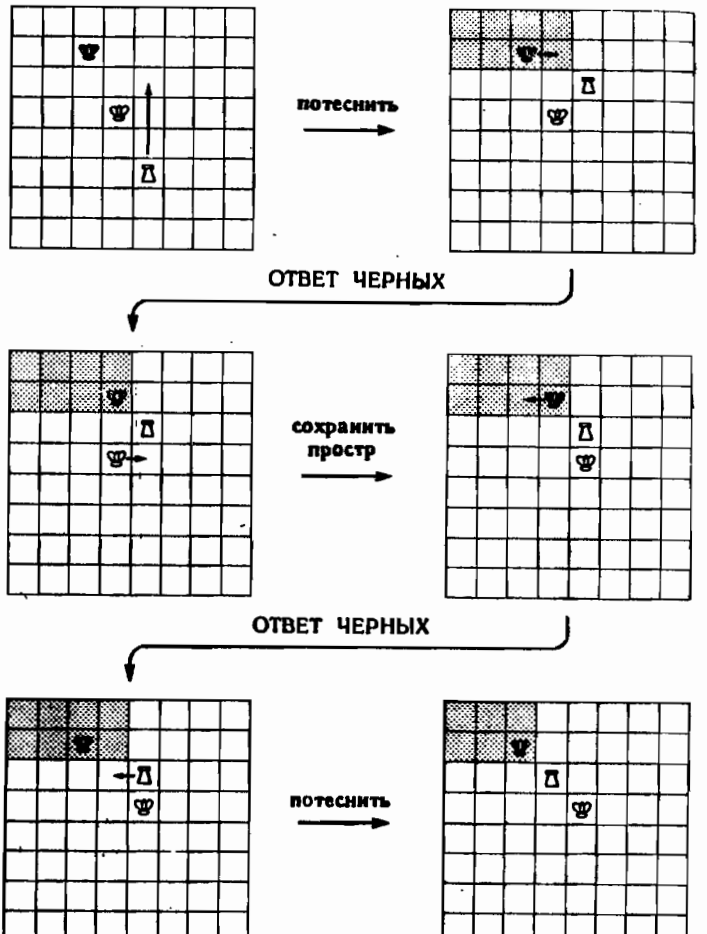


Рис. 15.8. Фрагмент шахматной партии, полученный с использованием таблицы советов рис. 15.7 и иллюстрирующий применение стратегии оттеснения короля в угол доски. В этой последовательности ходов выполнялись элементарные советы: **сохранить простр** (выжидающий ход, сохраняющий "жизненное пространство" черного короля) и **потеснить** (ход, сокращающий "жизненное пространство"). Область, в которой заключен черный король, выделена штриховкой. После выполнения последнего совета **потеснить** эта область сократилась с восьми до шести клеток.

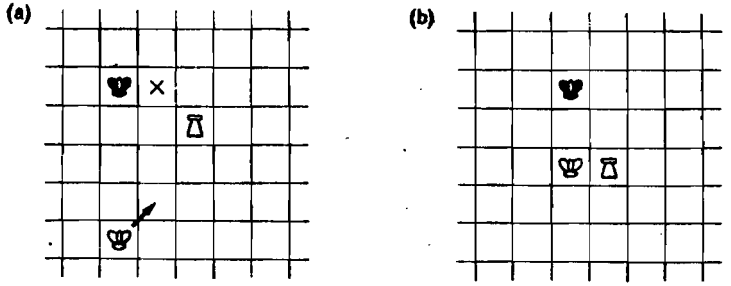


Рис. 15.9. (а) "Критическая клетка" отмечена крестиком. Она используется при маневрировании с целью оттеснить черного короля. Белый король приближается к "критической клетке", двигаясь, как указано на рисунке. (б) Три фигуры образуют конфигурацию, напоминающую букву L.

Аргументами этих предикатов являются либо позиции (в предикатах целей), либо ходы (в предикатах, ограничивающих ходы). Предикаты целей могут иметь один или два аргумента. Первый из аргументов – это всегда текущая вершина поиска; второй аргумент (если он имеется) – корневая вершина дерева поиска. Второй аргумент необходим в так называемых сравнивающих предикатах, которые сравнивают корневую и текущую позиции в том или ином отношении. Например, предикат **уменьш_простр** проверяет, сократилось ли «жизненное пространство» короли противника (рис. 15.8). Эти предикаты вместе с шахматными правилами (применительно к окончанию «король и ладья против короля»), а также процедура для отображения текущего состояния игровой доски (**отобр (Поз)**) запрограммированы на рис. 15.10.

На рис. 15.8 показано, как играет наша программа, основанная на механизме советов. При продолжении игры из последней позиции рис. 15.8 она могла бы протекать так, как в приведенном ниже варианте (в предположении, что «противник» ходит именно так, как указано). Здесь использована алгебраическая шахматная нотация, в которой вертикальные линии пронумерованы, как 'a', 'b', 'c', ..., а горизонтальные – как 1, 2, 3, ... Например, ход ЧК b7 означает: передвинуть черного короля на клетку,

расположенную на пересечении вертикальной линии 'b' с горизонтальной линией 7.

```
...          ЧК b7
БК d5        ЧК c7
БК c5        ЧК b7
БЛ c6        ЧК a7
БЛ b6        ЧК a8
БК b5        ЧК a7
БК c6        ЧК a8
БК c7        ЧК a7
БЛ c6        ЧК a8
БЛ a6        мат
```

Теперь уместно задать некоторые вопросы. Во-первых, является ли наша программа-советчик *корректной* в том смысле, что она ставит мат при любом варианте защиты со стороны противника и при любой начальной позиции, в которой на доске король и ладья против короля? В статье Bratko (1978) приведено формальное доказательство того, что таблица советов, практически совпадающая с таблицей рис. 15.7, действительно является корректной в указанном смысле.

Другой возможный вопрос: является ли программа оптимальной, то есть верно ли, что она ставит мат за минимальное число ходов? Нетрудно показать на примерах, что игру нашей программы в этом смысле нельзя назвать оптимальной. Известно, что оптимальный вариант в этом окончании (т.е. предполагающий оптимальную игру с обеих сторон) имеет длину не более 16 ходов. Хотя наша таблица советов и далека от этого оптимума, было показано, что число ходов наверняка не превосходит 50. Это важный результат в связи с тем, что в шахматах существует «правило 50-ти ходов»: в эндшпилях типа «король и ладья против короля» противник, имеющий преимущество, должен поставить мат не более, чем за 50 ходов; иначе может быть объявлена ничья.

Проект

Рассмотрите какой-нибудь другой простой эндшпиль, например «король и пешка против короля», и напишите для него программу на языке ALO (вместе с определениями соответствующих предикатов).

```
% Библиотека предикатов для окончания
% "король и ладья против короля"

% Позиция представлена стуктурой:
% ЧейХод..Бх : Бу..Лх : Лу..Чх : Чу..Глуб
% ЧейХод - с чьей стороны ход в этой позиции ('б' или 'ч')
% Бх, Бу - координаты белого короля
% Лх, Лу - координаты белой ладьи
% Чх, Чу - координаты черного короля
% Глуб - глубина, на которой находится эта позиция в дереве
% поиска

% Отношения выбора элементов позиции

чей_ход( ЧейХод..._, ЧейХод).
бк(_..БК..._, БК).                    % Белый король
бл(_..._..БЛ..._, БЛ).                % Белая ладья
чк(_..._..._..ЧК..._, ЧК).            % Черный король
глуб(_..._..._..._..Глуб, Глуб).

восст_глуб(ЧХ..Б..Л..Ч..Г, ЧХ..Б..Л..Ч..0).
    % Формируется копия позиции, глубина устанавливается в 0

% Некоторые отношения между клетками доски

сосед_чсл( N, N1) :-   % Соседнее число "в пределах доски"
  ( N1 is N + 1;
    N1 is N - 1 ),
  внутри( N1).

внутри( N) :-
  N > 0, N < 9.

сосед_диаг( X : Y, X1 : Y1) :-
                  % Соседние клетки по диагонали
  сосед_чсл( X, X1), сосед_чсл( Y, Y1).

сосед_верт( X : Y, X : Y1) :-
                  % Соседние клетки по вертикали
  сосед_чсл( Y, Y1).

сосед_гор( X : Y, X1 : Y) :-
                  % Соседние клетки по горизонтали
  сосед_чсл( X, X1).

сосед( S, S1) :-
                  % Соседние клетки (предпочтение - диагонали)
  сосед_диаг( S, S1);
  сосед_гор( S, S1);
```

```
  сосед_верт( S, S1).
конец_игры( Поз) :-
  мат( Поз).

% Предикаты, ограничивающие ходы
% Специализированные генераторы ходов вида:
% ход( Ограничение, Поз, Ход, Поз1)

ход( глубина < Макс, Поз, Ход, Поз1) :-
  глуб( Поз, Г),
  Г < Макс, !.

ход( глубина = Г, Поз, Ход, Поз1) :-
  глуб( Поз, Г), !.

ход( сначала_диаг, б..Б..Л..Ч..Г, Б-Б1,
                   ч..Б1..Л..Ч..Г1) :-
  Г1 is Г + 1,
  сосед( Б, Б1),
                % "сосед" порождает сначала диагональные ходы
  not сосед( Б1, Ч),              % Не попасть под шах
  Б1 \== Л.                       % Не столкнуться с ладьей

ход( ход_ладьей, б..Б..Лх : Лу..Ч..Г, Лх : Лу-Л,
                                  ч..Б..Л..Ч..Г1) :-
  Г1 is Г + 1,
  коорд( I),                      % Число между 1 и 8
  ( Л = Лх : I; Л = I : Лу),
                % По горизонтали или по вертикали
  Л \== Лх : Лу,                  % Обязательно двигаться
  not мешает( Лх : Лу, Б, Л).     % Мешает белый король

ход( ход_шах, Поз, Л-Лх : Лу, Поз1) :-
  бл( Поз, Л),
  чк( Поз, Чх : Чу),
  ( Лх = Чх; Лу = Чу),
                % Ладья и черный король на одной линии
  ход( ход_ладьей, Поз, Л-Лх : Лу, Поз1).

ход( разреш, б..П, М, П1) :-
  ( Огр = сначала_диаг; Огр = ход_ладьей),
  ход( Огр, б..П, М, П1).

ход(разреш,ч..Б..Л..Ч..Г,Ч-Ч1,б..Б..Л..Ч1..Г1) :-
  Г1 is Г + 1,
  сосед( Ч, Ч1),
  not шах( б..Б..Л..Ч1..Г1).
```

```
разрход( Поз, Ход, Поз1) :-
  ход( разреш, Поз, Ход, Поз1).

шах( _..Б..Лх : Лу..Чх : Чу.._) :-
  сосед( Б, Чх : Чу);                % Короли рядом
  (Лх = Чх; Лу = Чу),
  Лх : Лу \== Чх : Чу,               % Нет взятия ладьи
  not мешает( Лх : Лу, Б, Чх : Чу).

мешает( S, S1, S1) :- !.

мешает( X1 : Y, X2 : Y, X3 : Y) :-
  упоряд( X1, X2, X3), !.

мешает( X : Y1, X : Y2, X : Y3) :-
  упоряд( Y1, Y2, Y3).

упоряд( N1, N2, N3) :-
  N1 < N2, N2 < N3;
  N3 < N2, N2 < N1.

коорд( 1). коорд( 2). коорд( 3). коорд( 4).

коорд( 5). коорд( 6). коорд( 7). коорд( 8).

% Преднкаты целей

любая_поз( Поз).

ход_противника( б.._).              % Противник ходит белыми

мат( Поз) :-
  чей_ход( Поз, ч),
  шах( Поз),
  not разрход( Поз, _, _).

пат( Поз) :-
  чей_ход( Поз, ч),
  not шах( Поз),
  not разрход( Поз, _, _).

уменьш_простр( Поз, КорнПоз) :-
  простр( Поз, Пр),
  простр( КорнПоз, КорнПр),
  Пр < КорнПр.

ладья_под_боем( ЧейХод..Б..Л..Ч.._) :-
  расст( Б, Л, Р1),
  расст( Ч, Л, Р2),
  ( ЧейХод = б, !, Р1 > Р2 + 1;
    ЧейХод = ч, !, Р1 > Р2 ).
```

```
ближе_к_клетке( Поз, КорнПоз) :-
   расст_до_клетки( Поз, Р1),
   расст_до_клетки( КорнПоз, Р2),
   Р1 < Р2.
расст_до_клетки( Поз, Мрасст) :-
                        % Манхеттеновское расстояние
   бк( Поз, БК),        % между БК и критической клеткой
   кк( Поз, КК),        % Критическая клетка
   манх_расст( БК, КК, Мрасст).

раздел( _..Бх : Бу..Лх : Лу.. Чх : Чу.._) :-
   упоряд( Бх, Лх, Чх), !;
   упоряд( Бу, Лу, Чу).

l_конфиг( _..Б..Л..Ч.._) :-          % L - конфигурация
   манх_расст( Б, Ч, 2),
   манх_расст( Л, Ч, 3).

не_дальше_от_ладьи( _..Б..Л.._, _..Б1..Л1.._) :-
   расст( Б, Л, Р),
   расст( Б1, Л1, Р1),
   Р =< Р1.

простр_больше_2( Поз) :-
   простр( Поз, Пр),
   Пр > 2.

наш_король_на_краю(_..Х : У.._) :-
                        % Белый король на краю
   (Х = 1, !; Х = 8, !; У = 1, !; У = 8).

король_противника_на_краю(_..Б..Л..Х : У.._) :-
                        % Черный король на краю
   (Х = 1, !; Х = 8, !; У = 1, !; У = 8).

королн_рядом( Поз) :-    % Расстояние между королями < 4
   бк( Поз, БК), чк( Поз, ЧК),
   расст( БК, ЧК, Р),
   Р < 4.

потеря_ладьи( _..Б..Л..Л.._).      % Ладья взята

потеря_ладьи(ч..Б..Л..Ч.._) :-
   сосед( Ч, Л),        % Черный король напал на ладью
   not сосед( Б, Л).    % Белый король не защищает ладью

расст( Х : У, Х1 : У1, Р) :-      % Расстояние до короля
   абс_разн( Х, Х1, Рх),
   абс_разн( У, У1, Ру),
   макс( Рх, Ру, Р).
```

```
абс_разн( A, B, C) :-
  A > B, !, C is A - B;
  C is B - A.

макс( A, B, M) :-
  A >= B, !, M = A;
  M = B.

манх_расст( X : Y, X1 : Y1, P) :-
                          % Манхеттеновское расстояние
  абс_разн( X, X1, Px),
  абс_разн( Y, Y1, Py),
  P is Px + Py.

простр( Поз, Пр) :-
                % Область, в которой "заперт" черный король
  бл( Поз, Лх : Лу),
  чк( Поз, Чх : Чу),
  ( Чх < Лх, СторонаX is Лх - 1;
    Чх > Лх, СторонаX is 8 - Лх ),
  ( Чу < Лу, СторонаY is Лу - 1;
    Чу > Лу, СторонаY is 8 - Лу ),
  Пр is СторонаX * СторонаY, !;
  Пр = 64.        % Ладья и черный король на одной линии

кк( _..Б..Лх : Лу.. Чх : Чу.._, Кх : Ку) :-
                % Критическая клетка
  ( Чх < Лх, !, Кх is Лх - 1; Кх is Лх + 1),
  ( Чу < Лу, !, Ку is Лу - 1; Ку is Лу + 1).


% Процедуры для отображения позиций

отобр( Поз) :-
  nl,
  коорд( Y), nl,
  коорд( X),
  печ_фиг( X : Y, Поз),
  fail.

отобр( Поз) :-
  чей_ход( Поз, ЧХ), глуб( Поз, Г),
  nl, write( 'ЧейХод='), write( ЧХ),
  write( 'Глубина='), write( Г), nl.

печ_фиг( Клетка, Поз) :-
  бк( Поз, Клетка), !, write( 'Б');
  бл( Поз, Клетка), !, write( 'Л');
  чк( Поз, Клетка), !, write( 'Ч');
  write( '.').
```

```
показать_ход( Ход) :-
  nl,  write( Ход),  nl.
```

Рис. 15.10. Библиотека предикатов для окончания "король и ладья против короля".

Резюме

- Игры двух лиц поддаются формальному представлению в виде И/ИЛИ-графов. Поэтому процедуры поиска в И/ИЛИ-графах применимы для поиска в игровых деревьях.
- Простой алгоритм поиска в глубину в игровых деревьях легко программируется, но для игр, представляющих интерес, он не эффективен. Более реалистичный подход – минимаксный принцип в сочетании с оценочной функцией и поиском, ограниченным по глубине.
- Альфа-бета алгоритм является эффективной реализацией минимаксного принципа. Эффективность альфа-бета алгоритма зависит от порядка, в котором просматриваются варианты ходов. Применение альфа-бета алгоритма приводит, в лучшем случае, к уменьшению коэффициента ветвления дерева поиска, соответствующему извлечению из него квадратного корня.
- В альфа-бета алгоритм можно внести ряд усовершенствований. Среди них: продолжение поиска за пределы ограничения по глубине вплоть до спокойных позиций, последовательное углубление и эвристическое отсечение ветвей.
- Численная оценка позиций является весьма ограниченной формой представления знаний о конкретной игре. Более богатый по своим возможностям метод представления знаний должен предусматривать внесение в программу знаний о типовых ситуациях. Язык Советов (Advice Language) реализует такой подход. На этом языке знания представляются в терминах целей и средств для их достижения.

- В данной главе мы составили следующие программы: программная реализация минимаксного принципа и альфа-бета процедуры, интерпретатор языка ALO и таблица советов для окончания «король и ладья против короля».
- Были введены и обсуждены следующие понятия:

  игры двух лиц с полной информацией
  игровые деревья
  оценочная функция, миниимаксный принцип
  статические оценки, рабочие оценки
  альфа-бета алгоритм
  последовательное углубление,
  эвристическое отсечение,
  эвристики для обнаружения спокойных позиций

  Языки Советов

  цели, ограничения, элементарные советы,
  таблица советов

Литература

Минимаксный принцип, реализованный в форме альфа-бета алгоритма, – это наиболее популярный метод в игровом программировании. Особенно часто он применяется в шахматных программах. Минимаксный принцип был впервые предложен Шенноном (Shannon 1950). Возникновение и становление альфа-бета алгоритма имеет довольно запутанную историю. Несколько исследователей независимо друг от друга открыли либо реализовали этот метод полностью или частично. Эта интересная история описана в статье Knuth and Moore (1978). Там же приводится более компактная формулировка альфа-бета алгоритма, использующая вместо минимаксного принципа принцип «него-макса» ("neg-max" principle), и приводится математический анализ производительности алгоритма. Наиболее полный обзор различных минимаксных алгоритмов вместе с их теоретическим анализом содержится в книге Pearl (1984). Существует еще один интересный вопрос, относящийся к миниимаксному принципу. Мы знаем, что статическим оценкам следует доверять только до некоторой степени. Можно ли считать, что рабочие оценки являются более надежными, чем

исходные статические оценки, из которых они получены? В книге Pearl (1984) собран ряд математических результатов, имеющих отношение к ответу на этот вопрос. Приведенные в этой книге результаты, касающиеся распространения ошибок по минимаксному дереву, объясняют, в каких случаях и почему минимаксный принцип оказывается полезным.

Сборник статей Bramer (1983) охватывает несколько аспектов игрового программирования. Frey (1983) – хороший сборник статей по шахматным программам. Текущие разработки в области машинных шахмат регулярно освещаются в серии Advances in Computer Chess и в журнале ICCA.

Метод Языка Советов, позволяющий использовать знания о типовых ситуациях, был предложен Д. Мики. Дальнейшее развитие этого метода отражено в Bratko and Michi (1980 a,b) и Bratko (1982, 1984, 1985). Программа для игры в эндшпиле «король и ладья против короля», описанная в этой главе, совпадает с точностью до незначительных модификаций с таблицей советов, корректность которой была математически доказана в статье Bratko (1978). Ван Эмден также запрограммировал эту таблицу советов на Прологе (van Emden 1982).

Среди других интересных экспериментов в области машинных шахмат, преследующих цель повысить эффективность знаний (а не перебора), следует отметить Berliner (1977), Pitrat (1977) и Wilkins (1980).

Advances in Computer Chess Series (M.R.B. Clarke, ed). Edinburgh University Press (Vols. 1–2), Pergamon Press (Vol. 3).

Berliner M. A. (1977). A representation and some mechanisms for a problem solving chess program. In: *Advances in Computer Chess 1* (M.R.B. Clarke, ed). Edinburgh University Press.

Bramer M. A. (1983, ed). *Computer Game Playing: Theory and Practice.* Ellis Horwood and John Wiley.

Bratko I. (1978) Proving correctness of strategies in the AL1 assertional language. *Information Processing Letters* 7: 223–230.

Bratko I. (1982). Knowledge-based problem solving in AL3. In: *Machine Intelligence 10* (J. Hayes, D.

Michie, J. H. Pao, eds.). Ellis Horwood (an abbreviated version also appears in Bramer 1983).

Bratko I. (1984). Advise and planning in chess end-games. In: *Artificial and Human Intelligence* (S. Amarel, A. Elithorn, R. Banerji, eds.). North-Holland.

Bratko I. (1985). Symbolic derivation of chess patterns. In: *Progress Artificial Intelligence* (L. Steels, J. A. Campbell, eds.). Ellis Horwood and John Wiley.

Bratko I. and Michie D. (1980a). A representation of pattern-knowledge in chess end-games. In: *Advances in Computer Chess* 2 (M.R.B. Clarke, ed). Edinburgh University Press.

Bratko I. and Michie D. (1980b). An advice program for a complex chess programming task. *Computer Journal* **23**: 353-359.

Frey P. W. (1983, ed.). *Chess Skill in Man and Machine* (second edition). Springer-Verlag.

Knuth D. E. and Moore R. W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence* **6**: 93-326.

Pearl J. (1984). Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley.

Pitrat J. (1977). A chess combination program which uses plans *Artificial Intelligence* 8: 275-321.

Shannon C.E. (1950). Programming a computer for playing chess. *Philosophical Magazine* **41**: 256-275. [В сб. Шеннон К. Работы по теории информации и кибернетике. – М.: ИЛ., 1963.]

van Emden M. (1982). Chess end-game advice: a case study in computer utilisation of knowledge. In: *Machine Intelligence 10* (J. Hayes, D. Michie, J.H. Pao, eds). Ellis Hordwood.

Wilkins D.E. (1980). Using patterns and plans in chess. *Artificial Intelligence* **14**: 165-203.

16 ПРОГРАММИРОВАНИЕ В ТЕРМИНАХ ТИПОВЫХ КОНФИГУРАЦИЙ

В этой главе мы будем заниматься системами, ориентированнымн на типовые конфигурацни («образцы»), рассматривая их как некоторый специальный подход к программированию. Языком, ориентированным на образцы, можно считать и сам Пролог. Мы реализуем небольшой интерпретатор для простых программ этого типа и постараемся передать дух такого «конфигурационного» программирования на нескольких примерах.

16.1. Архитектура, ориентированная на типовые конфигурации

16.1.1. Основные понятия

Под *системами, ориентированными на типовые конфигурации* (*образцы*), мы будем понимать программные системы специальной архитектуры. Для некоторых конкретных типов задач такая архитектура дает преимущества по сравнению с традиционным способом организации. Среди задач, которые естественным образом вписываются в этот вид архитектуры, находятся многие приложения искусственного интеллекта, в том числе экспертные системы. Основное различие

между традиционными системами и системами, ориентированными на образцы, заключается в механизме запуска программных модулей. Традиционная архитектура предполагает, что модули системы обращаются друг к другу в соответствии с фиксированной, заранее заданной и явным образом сформулированной схемой. Каждый программный модуль сам принимает решение о том, какой из других модулей следует запустить в данный момент, причем в нем содержится *явное* обращение к этим модулям. Соответствующая временная структура передач управления от одних модулей к другим оказывается последовательной и детерминированной.

В противоположность этому организация, ориентированная на образцы, не предполагает прямого обращения из одних модулей к другим. Модули запускаются *конфигурациями*, возникающими в их «информационной среде». Такие программные модули называют модулями, *управляемыми типовыми конфигурациями* (или *образцами*). Программа, управляемая образцами, представляет из себя набор модулей. Каждый модуль определяется

(1) образцом, соответствующим предварительному условию запуска, и

(2) тем действием, которое следует выполнить, если информационная среда согласуется с заданным образцом.

Запуск модулей на выполнение происходит при появлении тех или иных конфигураций в информационной среде системы. Такую информационную среду обычно называют *базой данных*. Наглядное представление о системе рассматриваемого типа дает рис. 16.1.

Следует сделать несколько важных замечаний относительно рис. 16.1. Совокупность модулей не имеет иерархической структуры. Отсутствуют явные указания на то, какие модули могут обращаться к каким-либо другим модулям. Модули связаны скорее с базой данных, чем непосредственно друг с другом. В принципе такая структура допускает параллельное выполнение сразу нескольких модулей, поскольку текущее состояние базы данных может прийти в соот-

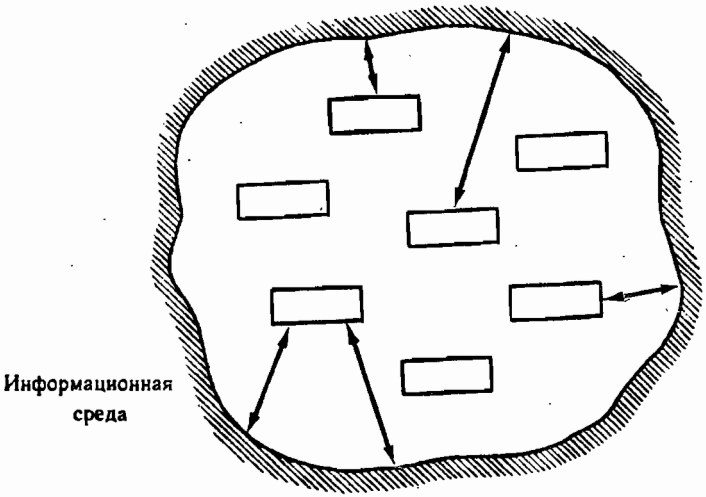


Рис. 16.1. Система, управляемая типовыми конфигурациями (образцами).

ветствие сразу с несколькими предварительными условиями, а следовательно, в принципе могут запуститься несколько модулей одновременно. В связи с этим, подобную организацию можно рассматривать как естественную модель параллельных вычислений, имея в виду, что каждый модуль физически реализован на отдельном процессоре.

Архитектура, ориентированная на образцы, обладает рядом достоинств. Одно из ее главных преимуществ состоит в том, что, разрабатывая подобную систему, мы не должны тщательно продумывать и заранее определять все связи между модулями. Следовательно, каждый модуль может быть разработан и реализован относительно автономно. Это придает системе высокую степень модульности, проявляющуюся, например, в том, что удаление из системы какого-либо модуля не обязательно приводит к фатальным последствиям. После удаления модуля система во многих случаях сохранит свою способность к решению задач, изменится может только *способ* их решения.

Аналогичное соображение верно и в случае добавления новых модулей или изменения уже существующих. Заметим, что при введении подобных модификаций в традиционные системы потребовалось бы, как минимум, пересмотреть связи между модулями.

Высокая степень модульности особенно желательна в системах со сложными базами знаний, поскольку очень трудно предсказать заранее все возможные взаимодействия между отдельными фрагментами знаний. Архитектура, ориентированная на образцы, обеспечивает простое решение этой проблемы: каждый фрагмент знаний, представленный в виде «если–то»-правила, можно считать отдельным модулем, запускаемым своим собственным образцом.

Перейдем теперь к более детальной проработке нашей базовой схемы для систем, ориентированных на образцы, и рассмотрим вопросы реализации. Как следует из рис. 16.1, параллельная реализация была бы для нашей системы наиболее естественным решением. Тем не менее предположим, что нам предстоит реализовать ее на традиционном последовательном процессоре. Тогда если в базе знаний окажется сразу несколько «пусковых» конфигураций, относящихся к нескольким модулям, то возникнет конфликтная ситуация: нам придется принять решение о том, какой из этих потенциально активных модулей будет запущен в действительности. Совокупность всех потенциально активных модулей назовем *конфликтным множеством.* Очевидно, что реализация схемы рис. 16.1 на последовательном процессоре потребует введения в систему дополнительного, *управляющего модуля.* Задача управляющего модуля – выбрать и активизировать один из модулей конфликтного множества и тем самым разрешить конфликт. Одно из возможных простых правил разрешения конфликта может основываться, например, на предварительном упорядочивании множества модулей системы.

Основной цикл работы системы, ориентированной на образцы, состоит, таким образом, из трех шагов:

(1) *Сопоставление с образцами*: найти в базе данных все конфигурации, сопоставимые с пусковыми образцами программных модулей.

Результат – конфликтное множество.

(2) *Разрешение конфликта*: выбрать один из модулей, входящих в конфликтное множество.

(3) *Выполнение*: запустить модуль, выбранный на предыдущем шаге.

Этот принцип реализации показан в виде схемы на рис. 16.2.

16.1.2. Прологовские программы как системы, управляемые образцами

Программы, написанные на Прологе, можно рассматривать как системы, управляемые образцами. Между пролог-программами и этими системами можно установить соответствие примерно следующим образом:

- Каждое предложение прологовской программы можно считать отдельным модулем со своим пусковым образцом. Голова предложения соответствует образцу, тело – тому действию, которое выполняет модуль.
- База данных системы – это текущий список целей, которые пролог-система пытается удовлетворить.
- Предложение пролог-системы «запускается», если его голова сопоставима с целью, расположенной первой в базе данных.
- Выполнить действие модуля (т.е. тело предложения) – это значит: поместить в базу данных вместо первой из целей весь список целей тела предложения (с соответствующей конкретизацией переменных).
- Процесс активизации модулей (предложений) недетерминирован в том смысле, что с первой целью базы данных могут удачно сопоставить свою голову сразу несколько предложений, и, вообще

говоря, любое из них может быть запущено. В Прологе этот недетерминизм реализован при помощи механизма возвратов.

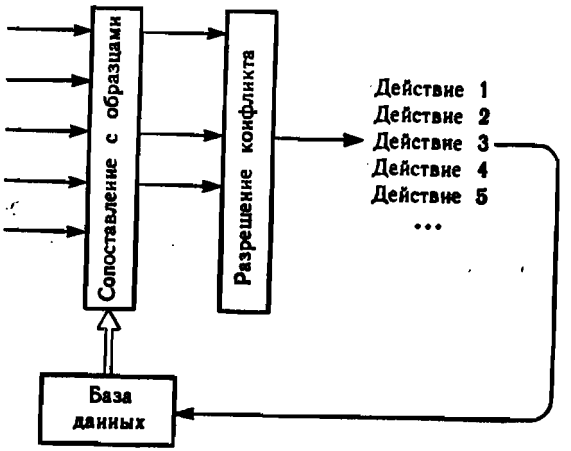


Рис. 16.2. Основной цикл работы системы, управляемой образцами. В этом примере база данных согласуется с пусковыми образцами модулей 1, 3 и 4; для выполнения выбран модуль 3.

16.1.3. Пример составления программы

С системами, управляемыми образцами, связан свой особый стиль программирования, требующий специфического программистского мышления. Мы говорим в этом случае о *программировании в терминах образцов.*

В качестве иллюстрации, рассмотрим элементарное упражнение по программированию – вычисление наибольшего общего делителя D двух целых чисел A и B. Рассмотрим классический алгоритм Евклида:

Для того, чтобы вычислить наибольший общий делитель D чисел A и B, необходимо:

Повторять циклически, пока A и B не совпадут:

если $A > B$, то заменить A на $A - B$,
иначе заменить B на $B - A$.

После выхода из цикла A и B совпадают; наибольший общий делитель D равен A (или B).

Тот же самый процесс можно описать при помощи двух модулей, управляемых образцами:

Модуль 1

Условие В базе данных существуют такие два числа X и Y, что $X > Y$.
Действие Заменить X на разность $X - Y$.

Модуль 2

Условие В базе данных имеется число X.
Действие Выдать результат X и остановиться.

Очевидно, что всегда, когда условие Модуля 1 удовлетворяется, удовлетворяется также и условие Модуля 2, и мы имеем конфликт. В нашем случае конфликт можно разрешить при помощи простого управляющего правила: всегда отдавать предпочтение Модулю 1. База данных первоначально содержит числа A и B.

Здесь нас ждет приятный сюрприз: оказывается, что наша программа способна решать более общую задачу, а именно, она может вычислять наибольший общий делитель для любого количества чисел. Если в базу данных загрузить несколько целых чисел, то программа выведет их наибольший общий делитель. На рис. 16.3 показана возможная последовательность изменений, которые претерпевает база данных прежде

де, чем будет получен результат. Обратите внимание на то, что предварительные условия модулей могут удовлетворяться одновременно в нескольких местах базы данных.

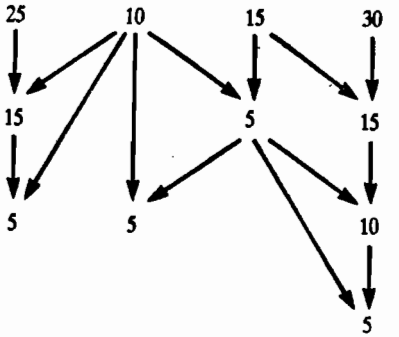


Рис. 16.3. Процесс вычисления наибольшего общего делителя множества чисел. Первоначально база данных содержит числа 25, 10, 15 и 30. Вертикальная стрелка соединяет число с его "заменителем". Конечное состояние базы данных: 5, 5, 5, 5.

В данной главе мы реализуем интерпретатор простого языка для описания систем, управляемых образцами, и проиллюстрируем на примерах дух программирования в терминах образцов.

16.2. Простой интерпретатор программ, управляемых образцами

Для описания модулей, управляемых образцами, мы применим следующую синтаксическую конструкцию:

ЧастьУсловия ---> ЧастьДействия

Часть условия представляет собой список условий:

[Условие1, Условие2, Условие3, ...]

где **Условие1**, **Условие2** и т.д. – обычные прологовские цели. Предварительное условие запуска модуля считается выполненным, если все цели, содержащиеся в списке, достигнуты. Часть действия – это список действий:

[Действие1, Действие2, ...]

Каждое отдельное действие – это, как и раньше, прологовская цель. Для того, чтобы выполнить список действий, нужно выполнить все действия из списка. Другими словами, все соответствующие цели должны быть удовлетворены. Среди допустимых действий будут действия, соответствующие манипулированию базой данных: **добавить**, **удалить** или **заменить** те или иные объекты базы данных.

На рис. 16.4 показано, как выглядит наша программа вычисления наибольшего общего делителя, записанная в соответствии с введенным нами синтаксисом.

```
% Продукционные правила для нахождения наибольшего общего
% делителя (алгоритм Евклида)

:- op( 300, fx, число).

[ число X, число Y, X > Y ] ---->
[ НовX is X - Y, заменить( число X, число НовX) ].

[ число X ] ----> [ write( X), стоп ].

% Начальное состояние базы данных

число 25.
число 10.
число 15.
число 30.
```

Рис. 16.4. Программа, управляемая образцами, для получения наибольшего общего делителя множества чисел.

Самый простой способ реализации этого языка – использовать механизмы управления базой данных, встроенные в Пролог. Добавить объект в базу данных

или удалить объект из базы данных можно, применяя встроенные процедуры

assert(Объект) **retract(Объект)**

Заменить один объект на другой также просто:

```
заменить( Объект1, Объект2) :-
    retract( Объект1), !,
    assert( Объект2).
```

Здесь задача оператора отсечения – не допустить, чтобы оператор **retract** удалил из базы данных более чем один объект (при возвратах).

```
% Простой интерпретатор для программ, управляемых образцами
% Работа с базой данных производится при помощи процедур
% assert и retract

:- op( 800, xfx, ---> ).

пуск :-
    Условие ---> Действие,          % Правило
    проверить( Условие),            % Условие выполнено?
    выполнить( Действие).

проверить( []).                     % Пустое условие

проверить([Усл|Остальные]) :-       % Проверить конъюнкцию
    call( Усл),                     % условий
    проверить( Остальные).

выполнить( [ стоп] ) :- !.          % Прекратить выполнение

выполнить( []) :-                   % Пустое действие (цикл завершен)
    пуск.                           % Перейти к следующему циклу

выполнить( [Д | Остальные] ) :-
    call( Д),
    выполнить( Остальные).

заменить( A, B) :-                  % Заменить в базе данных A на B
    retract( A), !,
    assert( B).
```

Рис. 16.5. Простой интерпретатор для программ, управляемых образцами.

Простой интерпретатор для программ, управляемых образцами, показан на рис. 16.5. Следует признать, что в интерпретаторе допущены значительные упрощения. Так, например, в него заложено чрезвычайно простое и жесткое правило разрешения конфликтов: всегда запускать *первый* из потенциально активных модулей (в соответствии с тем порядком, в котором модули записаны в программе). Таким образом, программисту предоставлено единственное средство управления процессом интерпретации – он может указать тот или иной порядок следования модулей. Начальное состояние базы данных задается в виде прологовских предложений, записанных в исходной программе. Запуск программы производится при помощи вопроса

```
?- пуск.
```

16.3. Простая программа для автоматического доказательства теорем

В настоящем разделе мы реализуем простую программу для автоматического доказательства теорем в виде системы, управляемой образцами. Эта программа будет основана на *принципе резолюции* – популярном методе, обычно используемом в машинном доказательстве теорем. Мы ограничимся случаем *пропозициональной логики*, поскольку нашей целью будет дать всего лишь простую иллюстрацию используемого принципа. На самом деле, принцип резолюции можно легко обобщить на случай исчисления высказываний первого порядка (с применением логических формул, содержащих переменные). Базовый Пролог можно рассматривать как частный случай системы доказательства теорем, основанной на принципе резолюции.

Задачу доказательства теорем можно сформулировать так: дана формула, необходимо показать, что

эта формула является теоремой, т. е. она верна всегда, независимо от интерпретации встречающихся в ней символов. Например, утверждение, записанное в виде формулы

$$p \vee \sim p$$

и означающее «p или не p», верно всегда, независимо от смысла утверждения p.

Мы будем использовать в качестве операторов следующие символы:

~	отрицание, читается как «не»
&	конъюнкцию, читается как «и»
v	дизъюнкцию, читается как «или»
=>	импликацию, читается как «следует»

Согласно правилам предпочтения операторов, оператор «не» связывает утверждения сильнее, чем «и», «или» и «следует».

Метод резолюции предполагает, что мы рассматриваем отрицание исходной формулы и пытаемся показать, что полученная формула противоречива. Если это действительно так, то исходная формула представляет собой тавтологию. Таким образом, основную идею можно сформулировать так: доказательство противоречивости формулы с отрицанием эквивалентно доказательству того, что исходная формула (без отрицания) есть теорема (т. е. верна всегда). Процесс, приводящий к искомому противоречию, состоит из отдельных шагов, на каждом из которых применяется резолюция.

Давайте проиллюстрируем этот принцип на примере. Предположим, что мы хотим доказать, что теоремой является следующая пропозициональная формула:

$$(a \Rightarrow b) \;\&\; (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$$

Смысл этой формулы таков: если из a следует b и из b следует c, то из a следует c.

Прежде чем начать применять процесс резолюции («резолюционный процесс»), необходимо представить отрицание нашей формулы в наиболее приспособленной для этого форме. Такой формой является *конъюнктивная нормальная форма*, имеющая вид

$$(p_1 \vee p_2 \vee \ldots) \& (q_1 \vee q_2 \vee \ldots) \& (r_1 \vee r_2 \vee \ldots) \& \ldots$$

Здесь p_i, q_i, r_i – элементарные утверждения или их отрицания. Конъюнктивная нормальная форма есть конъюнкция членов, называемых *дизъюнктами*, например $(p_1 \vee p_2 \vee \ldots)$ – это дизъюнкт.

Любую пропозициональную формулу нетрудно преобразовать в такую форму. В нашем случае это делается следующим образом. У нас есть исходная формула

$$(a \Rightarrow b) \& (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$$

Ее отрицание имеет вид

$$\sim ((a \Rightarrow b) \& (b \Rightarrow c) \Rightarrow (a \Rightarrow c))$$

Для преобразования этой формулы в конъюнктивную нормальную форму можно использовать следующие известные правила:

(1)	$x \Rightarrow y$	эквивалентно	$\sim x \vee y$
(2)	$\sim(x \vee y)$	эквивалентно	$\sim x \& \sim y$
(3)	$\sim(x \& y)$	эквивалентно	$\sim x \vee \sim y$
(4)	$\sim(\sim x)$	эквивалентно	x

Применяя правило 1, получаем

$$\sim (\sim ((a \Rightarrow b) \& (b \Rightarrow c)) \vee (a \Rightarrow c))$$

Далее, правила 2 и 4 дают

$$(a \Rightarrow b) \& (b \Rightarrow c) \& \sim(a \Rightarrow c)$$

Трижды применив правило 1, получаем

$$(\sim a \vee b) \& (\sim b \vee c) \& \sim(\sim a \vee c)$$

И наконец, после применения правила 2 получаем искомую конъюнктивную нормальную форму

$$(\sim a \vee b) \& (\sim b \vee c) \& a \& \sim c$$

состоящую из четырех дизъюнктов. Теперь можно приступить к резолюционному процессу.

Элементарный шаг резолюция выполняется всегда, когда имеется два дизъюнкта, в одном из которых встретилось элементарное утверждение p, а в другом

– $\sim p$. Пусть этими двумя дизъюнктами будут

$$p \vee Y \quad \text{и} \quad \sim p \vee Z$$

Шаг резолюции порождает третий дизъюикт:

$$Y \vee Z$$

Нетрудио показать, что этот дизъюнкт логически следует из тех двух дизъюнктов, из которых он получеи. Таким образом, добавив выражение $(Y \vee Z)$ к иашей исходной формуле, мы ие измеиим ее истиииости. Резолюциоииый процесс порождает иовые дизъюикты. Появление «пустого дизъюикта» (обычио записываемого как «nil») сигнализирует о противоречии. Действительно, пустой дизъюнкт *nil* порождается двумя дизъюиктами вида

$$x \quad \text{и} \quad \sim x$$

которые явно противоречат друг другу.

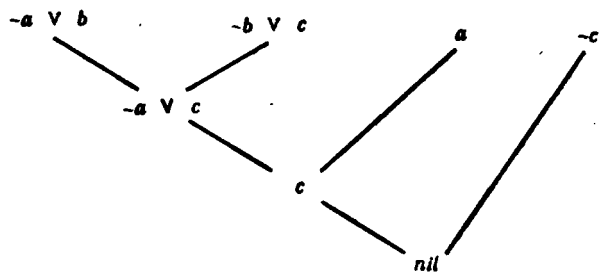


Рис. 16.6. Доказательство теоремы $(a \supset b) \& (b \supset c) \supset (a \supset c)$ методом резолюции. Верхняя строка – отрицание теоремы в конъюнктивной нормальной форме. Пустой днзъюикт внизу сигнализирует, что отрицание теоремы противоречиво.

На рис. 16.6 показаи процесс примеиеиия резолюций, иачинающийся с отрицаиия иашей предполагаемой теоремы и заканчивающийся пустым дизъюиктом.

На рис. 16.7 мы видим, как резолюционный процесс можио сформулировать в форме программы, упра-

вляемой образцами. Программа работает с дизъюнктами, записанными в базе данных. В терминах образцов принцип резолюции формулируется следующим образом:

если

существуют два таких дизъюнкта *C1* и *C2*, что *P* является (дизъюнктивным) подвыражением *C1*, а *~P* – подвыражением *C2*

то

удалить *P* из *C1* (результат – *CA*), удалить *~P* из *C2* (результат – *CB*) и добавить в базу данных новый дизъюнкт *CA* v *CB*.

На нашем формальном языке это можно записать так:

```
[ дизъюнкт( C1), удалить( P, C1, CA),
  дизъюнкт( C2), удалить( ~P, C2, CB) ] --->
[ assert( дизъюнкт( CA v CB) ) ].
```

Это правило нуждается в небольшой доработке. Дело в том, что мы не должны допускать повторных взаимодействий между дизъюнктами, так как они порождают новые копии уже существующих формул. Для этого в программе рис. 16.7 предусматривается запись в базу данных информации об уже произведенных взаимодействиях в форме утверждений вида

```
сделано( C1, C2, P)
```

В условных частях правил производится распознавание подобных утверждений и обход соответствующих повторных действий.

Правила, показанные на рис. 16.7, предусматривают также обработку специальных случаев, в которых требуется избежать явного представления пустого дизъюнкта. Кроме того, имеются два правила для упрощения дизъюнктов. Одно из них убирает избыточные подвыражения. Например, это правило превращает выражение

a v *b* v *a*

в более простое выражение *a* v *b*. Другое правило распознает те дизъюнкты, которые всегда истинны, например,

a v *b* v *~a*

и удаляет их из базы данных, поскольку они бесполезны при поиске противоречия.

```
% Продукционные правила для задачи автоматического
% доказательства теорем

% Протнворечие

[ дизъюнкт( X), дизъюнкт( ~X) ] --->
[ write( 'Обнаружено противоречие'), стоп].

% Удалить трнвиально истинный дизъюнкт

[ дизъюнкт(C), внутри(P,C), внутри(~P,C) ] --->
[ retract( C) ].

% Упростить днзъюнкт

[ дизъюнкт(C), удалить(P,C,C1), внутри(P,C1) ] --->
[ заменить( дизъюнкт( C), дизъюнкт( C1) ) ].

% Шаг резолюции, специальный случай

[ дизъюнкт(P), дизъюнкт(C), удалить(~P, C, C1) ,
            not сделано( P, C, P) ] --->
[ assert(дизъюнкт(C1)), assert(сделано(P,C,P))].

% Шаг резолюции, специальный случай

[ дизъюнкт(~P), дизъюнкт(C), удалить(P,C,C1) ,
            not сделано( ~P, C, P) ] --->
[ assert(дизъюикт(C1)), assert(сделано(~P,C,P))].

% Шаг резолюцнн, общнй случай

[ дизъюнкт( C1), удалить( P, C1, CA) ,
  дизъюнкт( C2), удалить( ~P, C2, CB) ,
  not сделано( C1, C2, P) ] --->
[ assert( дизъюнкт( CA v CB) ),
  assert( сделано( C1, C2, P) ) ].

% Последнее правнло: тупик

[] ---> [ write( 'Нет противоречия'), стоп ].
```

```
% удалить( P, E, E1) означает: получить из выражения E
% выражение E1, удалив из него подвыражение P

удалить( X, X v Y, Y).

удалить( X, Y v X, Y).

удалить( X, Y v Z, Y v Z1) :-
  удалить( X, Z, Z1).

удалить( X, Y v Z, Y1 v Z) :-
  удалить( X, Y, Y1).

% внутри( P, E) означает: P есть дизъюнктивное подвыражение
% выражения E

внутри( X, X).

внутри( X, Y) :-
  удалить( X, Y, _).
```

Рис. 16.7. Программа, управляемая образцами, для автоматического доказательства теорем.

Остается еще один вопрос: как преобразовать заданную пропозициональную формулу в конъюнктивную нормальную форму? Это несложное преобразование выполняется с помощью программы, показанной на рис. 16.8. Процедура

транс(Формула)

транслирует заданную формулу в множество дизъюнктов C1, C2 и т.д. и записывает их при помощи assert в базу данных в виде утверждений

```
дизъюнкт( C1).
дизъюнкт( C2).
...
```

Программа, управляемая образцами, для автоматического доказательства теорем запускается при помощи цели **пуск**. Таким образом, для того чтобы доказать при помощи этой программы некоторую теорему, мы транслируем ее отрицание в конъюнктивную нормальную форму, а затем запускаем резолюционный процесс. В нашем примере это можно сделать так:

```
% Преобразование пропозициональной формулы в множество
% дизъюнктов с записью их в базу данных при помощи assert

:- op( 100, fy, ~).                    % Отрицание
:- op( 110, xfy, &).                   % Конъюнкция
:- op( 120, xfy, v).                   % Дизъюнкция
:- op( 130, xfy, =>).                  % Импликация

транс( F & G) :- !,        % Транслировать конъюнктивную форм
  транс( F),
  транс( G).

транс( Формула) :-
  тр( Формула, НовФ), !,               % Шаг трансформации
  транс( НовФ).

транс( Формула) :-         % Дальнейшая трансформация невозмо
  assert( дизъюнкт( Формула) ).

% Правила трансформаций для пропозициональных формул

тр( ~( ~X), X) :- !.                  % Двойное отрицание

тр( X => Y, ~X v Y) :- !.              % Устранение импликации

тр( ~( X & Y), ~X v ~Y) :- !.          % Закон де Моргана

тр( ~( X v Y), ~X & ~Y) :- !.          % Закон де Моргана

тр( X & Y v Z, (X v Z) & (Y v Z) ) :- !.
                                       % Распределительный закон

тр( X v Y & Z, (X v Y) & (X v Z) ) :- !.
                                       % Распределительный закон

тр( X v Y, X1 v Y) :-                  % Трансформация подвыражения
  тр( X, X1), !.

тр( X v Y, X v Y1) :-                  % Трансформация подвыражения
  тр( Y, Y1), !.

тр( ~X, ~X1) :-                        % Трансформация подвыражения
  тр( X, X1).
```

Рис. 16.8. Преобразование пропозициональных формул в множество дизъюнктов с записью их в базу данных при помощи assert.

?- транс(~((a=>b) & (b=>c) => (a=>c))), пуск.

Ответ программы «Обнаружено противоречие» будет означать, что исходная формула является теоремой.

16.4. Заключительные замечания

Нашего простого интерпретатора было вполне достаточно для того, чтобы проиллюстрировать некоторые идеи, лежащие в основе программирования в терминах образцов. Применение этого интерпретатора для более сложных приложений потребовало бы его доработки в целом ряде направлений. Ниже приводится несколько критических замечаний, а также ряд конкретных предложений по усовершенствованию алгоритма интерпретации.

Задача разрешения конфликтов была сведена в нашем интерпретаторе к введению заранее заданного фиксированного порядка рассмотрения модулей. Часто возникает необходимость в более гибких механизмах. Для обеспечения более тонкого управления интерпретацией следует подавать все обнаруженные потенциально активные модули на вход специального управляющего модуля, запрограммированного пользователем.

Когда база данных велика, а программа содержит большое количество модулей, процесс сопоставления с образцами становится крайне неэффективным. Неэффективность можно уменьшить, усложнив организацию базы данных. В частности, можно ввести индексирование информации, записанной в базе данных, или разбить эту информацию на отдельные «подбазы данных», или же разбить все множество модулей на отдельные подмножества. Идея разбиения – в каждый момент дать доступ только к некоторому *подмножеству* базы данных или набора модулей, ограничив тем самым сопоставление образцов только этим подмножеством. Разумеется, в этом случае механизм

управления должен усложниться, поскольку он должен будет обеспечить переход от одних подмножеств другим с целью их активизации либо деактивизации. Для этого можно применить специальные метаправила.

К сожалению, наш интерпретатор запрограммирован таким образом, что он блокирует механизм автоматических возвратов, так как для манипулирования базой данных он использует процедуры **assert** и **retract**. Это положение можно исправить, применив другой способ реализации базы данных, не требующий обращения к этим встроенным процедурам. Например, все состояние базы данных можно представить одним прологовским термом, передаваемым в процедуру **пуск** в качестве аргумента. Простейший способ реализации этой идеи – организовать этот терм в виде списка объектов базы данных. Тогда верхний уровень базы данных примет вид:

```
пуск( Состояние) :-
    Условие ---> Действие,
    проверить( Условие, Состояние),
    выполнить( Действие, Состояние).
```

Задача процедуры **выполнить** – получить новое состояние базы данных и обратиться к процедуре **пуск**, подав на ее вход это новое состояние.

Проект

Запрограммируйте интерпретатор, который, в соответствии с приведенным выше замечанием, реализует базу данных как аргумент пусковой процедуры и не использует для этого внутренней базы данных пролог-системы (т. е. обходится без **assert** и **retract**). Эта новая версия интерпретатора будет допускать автоматические возвраты. Попытайтесь разработать такое представление базы данных, которое облегчало бы сопоставление с образцами.

Резюме

- Архитектура, ориентированная на типовые конфигурации (образцы), хорошо приспособлена для решения многих задач искусственного интеллекта.
- Программа, управляемая образцами, состоит из модулей, запускаемых при возникновении в базе данных тех или иных конфигураций.
- Прологовские программы можно рассматривать как частный случай систем, управляемых образцами.
- Параллельная реализация – наиболее естественный способ реализации систем, управляемых образцами. Реализация на последовательной машине требует *разрешения конфликтов* между модулями, содержащимися в *конфликтном множестве*.
- В этой главе был реализован простой интерпретатор для программ, управляемых образцами. Он был затем применен к задаче автоматического доказательства теорем пропозициональной логики.
- Были рассмотрены следующие понятия:

системы, управляемые образцами
архитектуры, ориентированные на образцы
программирование в терминах образцов
модули, управляемые образцами
конфликтное множество, разрешение конфликтов
принцип резолюции
автоматическое доказательство теорем на основе принципа резолюции

Литература

Waterman and Hayes-Roth (1978) – классическая книга по системам, управляемым образцами. В книге Nilsson (1980) можно найти фундаментальные понятия, относящиеся к задаче автоматического доказательства теорем, включая алгоритм преобразования логических формул в конъюнктивную нормальную форму. Прологовская программа для выполнения этого

преобразования приведена в Clocksin and Mell (1981).

Clocksin F. W. and Mellish C. S. (1981). *Progra ming in Prolog*. Springer-Verlag. [Имеется перев Клоксин У., Мелиш К. Программирование на язь Пролог. – М.: Мир, 1987.]

Nilsson N. J. (1980). *Principles of Artific Intelligence*. Tioga; Springer-Verlag.

Waterman D. A. and Hayes-Roth F. (1978, eds). *Pa tern-Directed Inference Systems*. Academic Press.

ОТВЕТЫ К НЕКОТОРЫМ УПРАЖНЕНИЯМ

Глава 1

1.1

(a) по
(b) X = пат
(c) X = боб
(d) X = боб, Y = пат

1.2

(a) ?- родитель(X, пат).
(b) ?- родитель(лиз, X).
(c) ?- родитель(Y, пат), родитель(X, Y).

1.3

(a)
```
счастлив( X) :-
  родитель( X, Y).
```

(b)
```
имеетдвухдетей( X) :-
  родитель( X, Y),
  сестра( Z, Y).
```

1.4

```
внук( X, Z) :-
  родитель( Y, X),
  родитель( Z, Y).
```

1.5

```
тетя( X, Y) :-
  родитель( Z, Y),
  сестра( X, Z).
```

1.6

Да. (Определение верно)

1.7

(a) возвратов не будет
(b) возвратов не будет
(c) возвратов не будет
(d) возвраты будут

Глава 2

2.1

(a) переменная
(b) атом
(c) атом
(d) переменная
(e) атом
(f) структура
(g) число
(h) синтаксически неправильное выражение
(i) структура
(j) структура

2.3

(a) успех

(b) неуспех

(c) неуспех

(d) D = 2, E = 2

(e)
```
P1 = точка(-1, 0)
P2 = точка( 1, 0)
P3 = точка( 0, Y)
```

Такая конкретизация определяет семейство треугольников, у которых две вершины располагаются на оси *x* в точках 1 и –1, а третья – в произвольной точке оси *y*.

2.4

```
отр( точка( 5, Y1), точка( 5, Y2) )
```

2.5

```
регулярный( прямоугольник( точка( X1, Y1),
                           точка( X2, Y1),точка( X2, Y3),
                           точка( X1, Y3) ) ).
```

Здесь предполагается, что первая точка соответствует нижней левой вершине прямоугольника.

2.6

(a) A = два

(b) по

(c) C = один

(d)
```
D = s(s(1));
D = s(s(s(s(s(1)))))
```

2.7
```
родственники( X, Y) :-
   предок( X, Y);
   предок( Y, X);
   предок( Z, X),
   предок( Z, Y);
   предок( X, Z),
   предок( Y, Z).
```

2.8
```
преобразовать( 1, один).
преобразовать( 2, два).
преобразовать( 3, три).
```

2.9
В случае, изображенном на рис. 2.10, пролог-система выполняет несколько больший объем работы.

2.10
В соответствии с определением сопоставления, приведенном в разд. 2.2, данное сопоставление будет успешным. X приобретает вид циклической структуры, в которой сам X присутствует в качестве одного из аргументов.

Глава 3

3.1

(a)
```
конк( L1, [_, _, _ ], L)
```

(b)
```
конк( [_, _, _ ], L1, L),
                % Удалить 3 первые элемента L
конк( L2, [_, _, _ ], L1)
                % Удалить 3 последние элемента L1
```

Вот более короткий вариант, предложенный I. Tvrdy:
```
конк( [_, _, _ | L2], [_, _, _ ], L)
```

3.2

(a)
```
последний( Элемент, Список) :-
конк( _, [Элемент], Список).
```

(b) , последний(Элемент, [Элемент]).

последний(Элемент,[Первый|Остальные]):-
последний(Элемент, Остальные).

3.3

```
четнаядлина( [] ).

четнаядлина( [Первый | Остальные] ) :-
  нечетнаядлина( Остальные).

нечетнаядлина( [ _ ] ).

нечетнаядлина( [Первый | Остальные] ) :-
  четнаядлина( Остальные).
```

3.4

```
обращение( [], []).

обращение([Первый|Остальные],ОбращСпис):-
  обращение( Остальные, ОбращСписОстальных),
конк(ОбращСписОстальных,[Первый],ОбращСпис).
```

3.5

```
% Такой предикат легко определить при помощи отношения обрат

палиндром( Список) :-
  обратить( Список, Список).

% Вот другое решение, не использующее обратить

палиндром1( [] ).

палиндром1( [_ ] ).

палиндром1( [Первый | Остальные] ) :-
  конк( Середина, [Первый], Остальные),
  палиндром1( Середина).
```

3.6

```
сдвиг( [Первый | Остальные], Сдвинут) :-
  конк( Остальные, [Первый], Сдвинут).
```

3.7

```
перевод( [], []).

перевод([Голова|Хвост],[Голова1|Хвост1]) :-
  означает( Голова, Голова1),
  перевод( Хвост, Хвост1).
```

3.8

```
подмножество( [], [] ).

подмножество([Первый|Остальные],[Первый|Подмн]):-
                    % Оставить первый элемент в подмножестве
  подмножество( Остальные, Подмн).

подмножество([Первый|Остальные], Подмн) :-
                    % Убрать первый элемент из подмножества
  подмножество( Остальные, Подмн).
```

3.9

```
разбениесписка( [], [], []).          % Разбивать нечего

разбениесписка( [X], [X], []).
                    % Разбиение одноэлементного списка

разбениесписка( [ X, Y|Список], [X|Список1],
                  [Y|Список2]) :-
  разбениесписка( Список, Список1, Список2).
```

3.10

```
можетзавладеть( состояние( _, _, _, имеет), [] ).
                    % Ничего не надо делать

можетзавладеть( Состояние, [Действие|Действия]):-
  ход( Состояние, Действие, НовоеСостояние),
                    % Первое действие
  можетзавладеть( НовоеСостояние, Действия).
                    % Оставшиеся действия
```

3.11

```
линеаризация( [Голова | Хвост], ЛинейныйСписок ) :-
                    % Линеаризация непустого списка
  линеаризация( Голова, ЛинейнаяГолова ),
  линеаризация( Хвост, ЛинейныйХвост ),
  конк( ЛинейнаяГолова, ЛинейныйХвост,
        ЛинейныйСписок ).

линеаризация( [], [] ).       % Линеаризация пустого списка

линеаризация( X, [X] ).
                % Линеаризация объекта, не являющегося списком

% Замечание: при попытке получить от этой программы более
% одного варианта решения выдается бессмыслица
```

3.12

```
Терм1 = играет_в( джимми, и( футбол, сквош) )
```

```
Терм2 = играет_в( сьюзаи, и( теннис,
                             и( баскетбол,  волейбол) ) )
```

3.13

```
:- op( 300, xfx, работает)
:- op( 200, xfx, в)
:- op( 100, xfx, нашем)
```

3.14

```
(a)     A = 1 + 0
(b)     B = 1 + 1 + 0
(c)     C = 1 + 1 + 1 +1 + 0
(d)     D = 1 + 1 + 0 + 1
```

3.15

```
:- op( 100, xfx, входит_в)
:- op( 300, fx, конкатенация_списков)
:- op( 200, xfx, дает)
:- op( 100, xfx, и)
:- op( 300, fx, удаление_элемента)
:- op( 100, xfx, из_списка)        % Принадлежность к сп

Элемент входит_в [Элемеит | Список].
Элемент аходит_в [Первый | СписокОстальных] :-
Элемент входит_в СписокОстальиых.

% Конкатенацня списков

конкатенация_списков [] и Список дает Список.

конкатенация_списков [X | L1] и L2 дает [X | L3] :-
    конкатенация_списков L1 и L2 дает L3.

% Удаление элемента из списка

удаление_элемента Элемеит из_списка
        [Элемент | ОстальныеЭлементы]
        дает ОстальныеЭлемеиты.

удаление_элемента Элемеит из_списка
      [Первый | ОстальныеЭлемеиты]
      дает [Первый | НовСписОстЭлементов] :-
    удаление_элемента Элемеит из_списка
        ОстальныеЭлемеиты дает НовСписОстЭлементов
```

3.16

```
max( X, Y, X) :-
    X >= Y.
```

```
max( X, Y, Y) :-
   X < Y.
```

3.17

```
максспис( [X], X).
                % Максимум в одноэлементном списке
максспис( [X, Y | Остальные], Max) :-
            % В списке есть по крайней мере два элемента?
   максспис( [Y | Остальные], МаксОстальные),
   max( X, МаксОстальные, Max).
            % Max - наибольшее из чисел X и МаксОстальные
```

3.18

```
сумспис( [], 0).
сумспис( [Первый | Остальные], Сумма) :-
   сумспис( Остальные, СуммаОстальных),
   Сумма is Первый + СуммаОстальных.
```

3.19

```
упорядоченный( []).
            % Одноэлементный список является упорядоченным
упорядоченный( [X, Y | Остальные] :-
   X =< Y,
   упорядоченный( [Y | Остальные] ).
```

3.20

```
подсумма( [], 0, []).
подсумма( [N | Список], Сумма, [N | Подмн]) :-
            % N принадлежит подмножеству
   Сумма1 is Сумма - N,
   подсумма( Список, Сумма1, Подмн).
нодсумма( [N | Список], Сумма, Подмн) :-
            % N не принадлежит подмножеству
   подсумма( Список, Сумма, Подмн).
```

3.21

```
между( N1, N2, N1) :-
   N1 =< N2.
между( N1, N2, X) :-
   N1 < N2,
   НовоеN1 is N1 + 1,
   между( НовоеN1, N2, X).
```

3.22

```
:- op( 900, fx, если).
:- op( 800, xfx, то).
:- op( 700, xfx, иначе).
:- op( 600, xfx, :=).
если Вел1 > Вел2 то Перем := Вел3
                    иначе ЧтоУгодно :-
    Вел1 > Вел2,
    Перем = Вел3.

если Вел1 > Вел2 то ЧтоУгодно
                    иначе Перем := Вел4 :-
    Вел1 =< Вел2,
    Перем = Вел4.
```

Глава 4

4.1

(a)
```
?- семья(членсемьи(_,Фамилия,_,_),_, []).
```

(b)
```
?- ребенок( членсемьи( Имя, Фамилия, _,
                работает( _, _ ) )).
```

(c)
```
семья(членсемьи(_,Фамилия,_,неработает),
      членсемьи(_,_,_,работает(_,_ ) ),_ ).
```

(d)
```
?- семья( Муж, Жена, Дети),
   датарождения( Муж, дата( _, _, Год1) ),
   датарождения( Жена, дата( _, _, Год2) ),
   ( Год1 - Год2 >= 15;
     Год2 - Год1 >= 15 ),
   принадлежит( Ребенок, Дети).
```

4.2

```
близнецы( Ребенок1, Ребенок2) :-
  семья( _, _, Дети),
  удалить( Ребенок1, Дети, ДругиеДети),
                % Выделить первого ребенка
  принадлежит( Ребенок2, ДругиеДети),
  принадлежит( Ребенок1, Дата),
  принадлежит( Ребенок2, Дата).
```

4.3

```
n_элемент( 1, [X | L], X).
                % X - первый элемент списка [X|L]

n_элемент( N, [Y | L], X) :-
                % X - n-й элемент [Y|L]
```

```
  N1 is N - 1,
  n_элемент( N1, L, X).
```

4.4

Входная цепочка укорачнвается на каждом неспонтанном цикле, а укорачнваться бесконечно она не может.

4.5

```
допускается( S, [], _ ) :-
  конечное( S).

допускается(S,[X|Остальные],Макс_переходов) :-
  Макс_переходов > 0,
  переход( S, X, S1),
  НовыйМакс is Макс_переходов - 1,
  допускается( S1, Остальные, НовыйМакс).

допускается( S, Цепочка, Макс_переходов) :-
  Макс_переходов > 0,
  спонтанный( S, S1),
  НовыйМакс is Макс_переходов - 1,
  допускается( S1, Цепочка, НовыйМакс).
```

4.7

(a)

```
ходконя( X/Y, X1/Y1) :-
        % Ход коня с поля X/Y на поле X1/Y1
  ( dxy( DX, DY);
        % Расстояния по направлениям X и Y
    dxy( DY, DX) ),
        % Или расстояния по направлениям Y и X
  X1 is X + DX,
        % X1 расположен в пределах шахматной доски
  надоске( X1),
  Y1 is Y + DY,
        % Y1 расположен в пределах шахматной доски
  надоске( Y1).

dxy( 2, 1).      % 2 поля вправо, 1 поле вперед
dxy( 2, -1).     % 2 поля вправо, 1 поле назад
dxy( -2, 1).     % 2 поля влево, 1 поле вперед
dxy( -2, -1).    % 2 поля влево, 1 поле назад

надоске( Коорд) :-
        % Координаты в пределах доски
  0 < Коорд,
  Коорд < 9.
```

```
(b)   путьконя( [ Поле]).      % Конь стоит на поле Поле
      путьконя( [S1, S2 | Остальные] ) :-
      ходконя( S1, S2),
      путьконя( [S2 | Остальные]).

(c)   ?- путьконя( [2/1, R, 5/4, S, X/8] ).
```

Глава 5

5.1

```
(a)   X = 1;
      X = 2;

(b)   X = 1
      Y = 1;

      X = 1
      Y = 2;

      X = 2
      Y = 1;

      X = 2
      Y = 2;

(c)   X = 1
      Y = 1;

      X = 1
      Y = 2;
```

5.2

```
класс( Число, положительное) :-
  Число > 0, !.

класс( 0, нуль) :- !.

класс( Число, отрицательное).
```

5.3

```
разбить( [], [], []).

разбить( [X | L], [X | L1], L2) :-
  X >= 0, !,
  разбить( L, L1, L2).

разбить( [X | L], L1, [X | L2]) .
  разбить( L, L1, L2).
```

5.4

```
принадлежит(Некто, Каидидаты),
      not принадлежит(Некто, Исключенные)
```

5.5

```
разность( [], _, []).

разность( [X | L1], L2, L) :-
  принадлежит( X, L2), !,
  разность( L1, L2, L).

разность( [X | L1], L2, [X | L]) :-
  разность( L1, L2, L).
```

5.6

```
унифицируемые( [], _, []).

унифицируемые([Первый|Остальные], Терм, Список) :-
  not( Первый = Терм), !,
  унифицируемые( Остальные, Терм, Список).

унифицируемые([Первый|Остальные], Терм,
              [Пераый|Список] ) :-
  унифицируемые( Остальные, Терм, Список).
```

Глава 6

6.1

```
найтитерм( Терм) :-
                % Пусть текущнй входной поток - это файл f
  read( Терм), !,
                % Текущий терм нз f сопоставим с Терм'ом?
  write( Терм);         % Если да - вывести его на терминал
  найтитерм( Терм).     % В противном случае - обработать
```

6.2

```
найтитермы( Терм) :-
  read( ТекущнйТерм),
  обработать( ТекущнйТерм, Терм).

обработать( end_of_file, _) :- !.

обработать( ТекущнйТерм, Терм) :-
  ( not( ТекущнйТерм = Терм), !;
                % Термы несопоставимы
    write( ТекущнйТерм), nl),
                % В противном случае вывестн текущий терм
  найтивсетермы( Терм).
                % Обработать оставшуюся часть файла
```

6.4
```
начинается( Атом, Символ) :-
   name( Символ, [ Код]),
   name( Атом, [Код | _ ]).
```

6.5
```
plural( Существительное, Существительные) :-
   name( Существительное, СписокКодов),
   name( s, КодS),
   конк( СписокКодов, КодS, НовыйСписокКодов),
   name( Существительные, НовыйСписокКодов).
```

Глава 7

7.2
```
добавить( Элемент, Список) :-
   var( Список), !,
                % Переменная Список представляет пустой список
   Список = [Элемент | Хвост].

добавить( Элемент, [_ | Хвост]) :-
   добавить( Элемент, Хвост).

принадлежит( X, Список) :-
   var( Список), !,
                % Переменная Список представляет пустой список,
                % поэтому X не может ему принадлежать
   fail.

принадлежит( X, [X | Хвост]).

принадлежит( X, [_ | Хвост] ) :-
   принадлежит( X, Хвост).
```

Глава 8

8.2
```
добавить_в_конец(L1-[Элемент|Z2],Элемент,L1 - Z2).
```

8.3
```
обратить( A - Z, L - L) :-
                % Результатом является пустой список,
                % если A-Z представляет пустой список
   A == Z, !.
```

```
обратить( [X | L] - Z, RL - RZ ) :-
                % Непустой список
  обратить( L - Z, RL  - [X | RZ].
```

Глава 9

9.1

```
список([]).

список([_ | Хвост]) :-
  список( Хвост).
```

9.2

```
принадлежит( X, X затем ЧтоУгодно).

принадлежит( X, Y затем Спис) :-
  принадлежит( X, Спис).
```

9.3

```
преобр( [], ничего_не_делать).

преобр([Первый | Хвост],Первый затем Остальные):-
  преобр( Хвост, Остальные).
```

9.4

```
преобр( [], ПустСпис, _, ПустСпис).
                % Случай пустого списка

преобр([Первый|Хвост],НовСпис,Функтор,Пустой) :-
  НовСпис =.. [Фуиктор, Первый, НовХвост],
  преобр( Хвост, НовХвост, Функтор, Пустой).
```

9.8

```
сорт1( [], []).

сорт1( [X], [X]).

сорт1( Спис, УпорСпис) :-
  разбить( Спис, Спис1, Спис2),
                % Разбить на 2 прибл. равных списка
  сорт1( Спис1, Упор1),
  сорт1( Спис2, Упор2),
  слить( Упор1, Упор2, УпорСпис).
                % Слить отсортированные списки

разбить( [], [], []).

разбить( [X], [X], []).
```

```
разбить( [X, Y | L], [X | L1], [Y | L2]) :-
                    % X и Y помещаются в разные списки
  разбить( L, L1, L2).
```

9.9

(a)
```
двдерево( nil).

двдерево( д( Лев, Кор, Прав) ) :-
  двдерево( Лев),
  двдерево( Прав).
```

9.10

```
глубина( пусто, 0).

глубина( д( Лев, Кор, Прав), Г) :-
  глубина( Лев, ГЛ),
  глубина( Прав, ГП),
  макс( ГЛ, ГП, МГ),
  Г is МГ + 1.

макс( A, B, A) :-
  A >= B, !.

макс( A, B, B).
```

9.11

```
линевризация( nil, []).

линеаризация( д( Лев, Кор, Прав), Спис) :-
  линеаризация( Лев, Спис1),
  линеаризация( Прав, Спис2),
  коик( Спис1, [Кор | Спис2], Спис).
```

9.12

```
мвксэлемент( д( _, Кор, nil), Кор) :- !.
                    % Корень - самый правый элемент

максэлемент( д( _, _, Прав,), Макс) :-
                    % Правое поддерево непустое
  мвксэлемент( Прав, Макс).
```

9.13

```
внутри( Элем, д( _, Элем, _), [ Элем]).

внутри( Элем, д( Лев, Кор, _), [Кор | Путь]) :-
  больше( Кор, Элем),
  внутри( Элем, Лев, Путь).

внутри( Элем, д( _, Кор, Прав), [Кор | Путь]) :-
```

```
  больше( Элем, Кор),
  внутри( Элем, Прав, Путь).
```

9.14

```
% Отображение двоичного дерева, растущего сверху вниз
% Предполагается, что каждая вершина занимает при печати
% один символ

отобр( Дер) :-
  уровни( Дер, 0, да).
                % Обработать все уровни

уровни( Дер, Уров, нет) :- !.
                % Ниже уровня Уров больше нет вершин

уровни( Дер, Уров, да) :-
                % Обработать все уровни, начиная с Уров
  вывод( Дер, Уров, 0, Дальше), nl,
                % Вывести вершины уровня Уров
  Уров1 is Уров + 1,
  уровни( Дер, Уров1, Дальше).
                % Обработать следующие уровни

вывод( nil, _, _, _, _).

вывод( д( Лев, Х, Прав), Уров, ГлубХ, Дальше) :-
  Глуб1 is ГлубХ + 1,
  вывод( Лев, Уров, Глуб1, Дальше),
                % Вывод левого поддерева
  ( Уров = ГлубХ, !,
                % Х на нашем уровне?
    write( Х), Дальше = да;
                % Вывести вершину, продолжить
    write(' ') ),
                % Иначе - оставить место
    вывод( Прав, Уров, Глуб1, Дальше).
                % Вывод левого поддерева
```

Глава 10

10.1

```
внутри(Элем,л(Элем)).        % Элемент найден в листе

внутри( Элем, в2( Д1, М, Д2) ):-
                % Вершина имеет два поддерева
  больше( М, Элем), !,        % Вершина не во втором поддереве
```

```
  внутри( Элем, Д1);        % Поиск в первом поддереве
  внутри( Элем, Д2).        % Иначе - во втором поддереве

внутри( Элем, в3( Д1, М2, Д2, М3, Д3) ):-
                            % Вершина имеет три поддерева
  больше( М2, Элем), !,
          Элемент не во втором и не в третьем поддереве
  внутри( Элем, Д1);        % Поиск в первом поддереве
  больше( М3, Элем), !,  % Элемент не в третьем поддерев
  внутри( Элем, Д2);        % Поиск во втором поддереве
  внутри( Элем, Д3).        % Поиск в третьем поддереве
```

10.3

```
avl( Дер) :-
  avl(Дер,Глуб).   % Дер является AVL-деревом глубины Глуб

avl( nil, 0).      % Пустое дерево - AVL -дерево глубины 0

avl( д( Лев, Кор, Прав), Г) :-
  avl( Лев, ГЛ),
  avl( Прав, ГП),
  ( ГЛ is ГП; ГЛ is ГП + 1; ГЛ is ГП - 1),
                   % Глубины поддеревьев примерно совпадают
    макс( ГЛ, ГП, Г).

макс1( U, V, M) :-                  % M = 1 + макс( U, V)
  U > V, !, M is U + 1;
  M is V + 1.
```

Глава 11

11.1

```
вглубину1( [Верш | Путь], [Верш | Путь]) :-
  цель( Верш).

вглубину1( [Верш | Путь], Решение) :-
  после( Верш, Верш1),
  not принадлежит( Верш1, Путь),
  вглубину1( [ Верш1, Верш | Путь], Решение).
```

11.6

```
решить( СтартМнож, Решение) :-
                  % СтартМнож - множество стартовых вершин
  bagof( [Верш], принадлежит( Верш, СтартМнож),
                                  Пути),
  вширину( Пути, Решение).
```

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

ОГЛАВЛЕНИЕ

Учебное издание

Иван Братко
Программирование на языке Пролог
для искусственного интеллекта

Заведующий редакцией чл.-корр. АН СССР проф. В.И.Арнольд
Зам.зав.редакцией А.С. Попов
Ст. научный редактор М. В. Хатунцева
Художественный редактор В.И. Шаповалов
Художник В.А. Мединков
График Л.Г. Туранова
Корректор А.Ф. Рыбальченко

ИБ № 7199

Оригинал-макет подготовлен на персональном компьютере и отпечатан на лазерном принтере в издательстве "Мир"

Подписано к печати 13.08.90. Формат 84×108¹/₃₂. Бумага кн. журн. имп. Печать высокая. Гарнитура литературная. Объем 8,75 бум. л. Усл. печ. л. 29,40. Усл. кр.-отт. 29,40. Уч.-изд. л. 25,60.

Изд. № 1/6603. Тираж 100 000 экз. Зак. № 633. Цена 3 р. 50 к.

Издательство "Мир" В/О "Совэкспорткнига" Госкомпечати СССР
129820, ГСП, Москва, И-110, 1-й Рижский пер., 2.

Отпечатано в Ленинградской типографии №2 головном предприятии ордена Трудового Красного Знамени Ленинградского объединения "Техническая книга" им. Евгении Соколовой Государственного комитета СССР по печати.
198052, г.Ленинград, Л-52, Измайловский проспект, 29.