ПЕДАГОГИЧЕСКОЕ ОБРАЗОВАНИЕ

Е. В. Боровская
Н. А. Давыдова

ОСНОВЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Лаборатория ЗНАНИЙ

ПЕДАГОГИЧЕСКОЕ ОБРАЗОВАНИЕ

Е. В. Боровская
Н. А. Давыдова

ОСНОВЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Учебное пособие

4-е издание, электронное

Москва
Лаборатория знаний
2020

УДК 004.8
ББК 32.813
Б83

Серия основана в 2007 г.

Боровская Е. В.

Б83 Основы искусственного интеллекта : учебное пособие / Е. В. Боровская, Н. А. Давыдова. — 4-е изд., электрон. — М. : Лаборатория знаний, 2020. — 130 с. — (Педагогическое образование). — Систем. требования: Adobe Reader XI ; экран 10". — Загл. с титул. экрана. — Текст : электронный.

ISBN 978-5-00101-908-4

Учебное пособие знакомит читателей с историей искусственного интеллекта, моделями представления знаний, экспертными системами и нейронными сетями. Описаны основные направления и методы, применяемые при анализе, разработке и реализации интеллектуальных систем. Рассмотрены модели представления знаний и методы работы с ними, методы разработки и создания экспертных систем. Книга поможет читателю овладеть навыками логического проектирования баз данных предметной области и программирования на языке ProLog.

Для студентов и преподавателей педагогических вузов, учителей общеобразовательных школ, гимназий, лицеев.

УДК 004.8
ББК 32.813

Деривативное издание на основе печатного аналога: Основы искусственного интеллекта : учебное пособие / Е. В. Боровская, Н. А. Давыдова. — М. : БИНОМ. Лаборатория знаний, 2010. — 127 с. : ил. — (Педагогическое образование). — ISBN 978-5-94774-480-4.

ISBN 978-5-00101-908-4

ОГЛАВЛЕНИЕ

ГЛАВА 1
ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

1.1. Введение в системы искусственного интеллекта

1.1.1. Понятие об искусственном интеллекте

Система **искусственного интеллекта (ИИ)** — это программная система, имитирующая на компьютере процесс мышления человека. Для создания такой системы необходимо изучить сам процесс мышления человека, решающего определенные задачи или принимающего решения в конкретной области, выделить основные шаги этого процесса и разработать программные средства, воспроизводящие их на компьютере. Следовательно, методы ИИ предполагают простой структурный подход к разработке сложных программных систем принятия решений [5].

Искусственный интеллект — это направление информатики, целью которого является разработка аппаратно-программных средств, позволяющих пользователю-непрограммисту ставить и решать свои традиционно считающиеся интеллектуальными задачи, общаясь с ЭВМ на ограниченном подмножестве естественного языка.

Идея создания искусственного подобия человека для решения сложных задач и моделирования человеческого разума, что называется, «витала в воздухе» еще в древнейшие времена. Родоначальником искусственного интеллекта считается средневековый испанский философ, математик и поэт Раймонд Луллий, который еще в XIII в. пытался создать механическое устройство для решения различных задач на основе разработанной им всеобщей классификации понятий.

Позже Лейбниц и Декарт независимо друг от друга продолжили эту идею, предложив универсальные языки классификации для всех наук. Эти работы можно считать первыми теоретическими работами в области искусственного интеллекта.

Однако окончательное рождение искусственного интеллекта как научного направления произошло только после создания ЭВМ в 1940-х гг., когда Норберт Винер создал свои основополагающие работы по новой науке — *кибернетике*.

Термин «искусственный интеллект» (ИИ; англ. AI — «Artificial Intelligence») был предложен в 1956 г. на семинаре с аналогичным названием в Дартмутском колледже (США). Этот семинар был посвящен разработке методов решения логических (а не вычислительных) задач. Заметим, что в английском языке данное словосочетание не имеет той слегка фантастической антропоморфной окраски, которую оно приобрело в довольно неудачном русском переводе. Слово «intelligence» означает всего лишь «умение рассуждать разумно», а вовсе не «интеллект» (для которого есть отдельный английский аналог «intellect»).

Вскоре после признания искусственного интеллекта особой областью науки произошло его разделение на два направления: *нейрокибернетику* и *кибернетику «черного ящика»*. Эти направления развивались практически независимо, существенно различаясь как в методологии, так и технологически. И только в настоящее время стали заметны тенденции к объединению этих частей вновь в единое целое.

Нейрокибернетика

Основную идею этого направления можно сформулировать следующим образом: «Единственный объект, способный мыслить, — это человеческий мозг, поэтому любое мыслящее устройство должно так или иначе воспроизводить его структуру». Таким образом, нейрокибернетика ориентирована на *программно-аппаратное моделирование структур, подобных структуре мозга*. Усилия нейрокибернетики были сосредоточены на создании элементов, аналогичных нейронам, и на их объединении в функционирующие системы — *нейронные сети* [2].

Первые нейросети были созданы в 1956–1965-х гг. Это были не очень удачные попытки создать системы, моделирующие человеческий глаз и его взаимодействие с мозгом. Постепенно в 1970–1980-х гг. количество работ по этому направлению искусственного интеллекта стало снижаться, — слишком уж неутешительными были первые результаты.

Обычно авторы разработок объясняли свои неудачи малой памятью и низким быстродействием существующих в то время компьютеров.

Первый нейрокомпьютер был создан в Японии в рамках проекта «ЭВМ пятого поколения». К этому времени ограничения по памяти и быстродействию ЭВМ были практически сняты. Появились *транспьютеры* — компьютеры с большим количеством процессоров, реализующих параллельные вычисления. Транспьютерная технология — это один из десятка новых подходов к аппаратной реализации нейросетей, которые моделируют иерархическую структуру мозга человека.

В целом сегодня можно выделить три основных разновидности подходов к созданию нейросетей: аппаратный (создание специальных компьютеров, нейрочипов, плат расширения, наборов микросхем), программный (создание программ и программных инструментариев, рассчитанных на высокопроизводительные компьютеры; такие сети создаются «виртуально», в памяти компьютера, тогда как всю работу выполняют его собственные процессоры) и гибридный (комбинация первых двух способов).

Кибернетика «черного ящика» и искусственный интеллект

В основу этого подхода заложен принцип, противоположный нейрокибернетике. Здесь уже не имеет значения, как именно устроено «мыслящее» устройство, — главное, чтобы на заданные входные воздействия оно реагировало так же, как и человеческий мозг.

Сторонники этого направления мотивировали свой подход тем, что человек не должен слепо следовать природе в своих научных и технологических поисках. К тому же пограничные науки о человеке не смогли внести существенно-

го теоретического вклада, объясняющего (хотя бы приблизительно), как протекают интеллектуальные процессы у человека, как устроена его память и как человек познает окружающий мир [2].

Это направление искусственного интеллекта было ориентировано на поиск алгоритмов решения интеллектуальных задач на существующих моделях компьютеров. Существенный вклад в становление новой науки внесли такие ее пионеры, как Маккарти, Минский, Ньюэлл, Саймон, Шоу, Хант и др.

В 1956–1963-х гг. велись интенсивные поиски моделей и алгоритмов человеческого мышления и разработка первых программ на их основе. Представители гуманитарных наук — философы, психологи, лингвисты — ни тогда, ни сейчас не смогли предложить такие алгоритмы, тогда кибернетики начали создавать собственные модели.

Так последовательно были созданы и опробованы различные подходы.

В конце 1950-х гг. появилась *модель лабиринтного поиска*. Этот подход представляет задачу как некоторое пространство состояний в форме графа, после чего в этом графе производится поиск оптимального пути от входных данных к результирующим. Была проделана большая работа по созданию такой модели, но для решения практических задач эта идея все же не нашла широкого применения.

Начало 1960-х гг. стало эпохой эвристического программирования. *Эвристика* — это некоторое правило, теоретически не обоснованное, но позволяющее сократить количество переборов в пространстве поиска. *Эвристическое программирование* — это разработка стратегии действий на основе известных, заранее заданных эвристик.

В 1963–1970-х гг. к решению задач стали подключать *методы математической логики*. Робинсон разработал *метод резолюций*, который позволяет автоматически доказывать теоремы при наличии набора исходных аксиом. Примерно в это же время выдающийся отечественный математик Ю. С. Маслов предложил так называемый *обратный вывод* (впоследствии названный его именем), решающий аналогичную задачу другим способом. На основе метода резолюций

француз Альбер Кольмероэ в 1973 г. создал *язык логического программирования Prolog*. Большой резонанс в научном сообществе вызвала *программа «Логик-теоретик»*, созданная Ньюэллом, Саймоном и Шоу, которая доказывала школьные теоремы. Однако большинство реальных задач не сводится к набору аксиом, а человек, решая производственные задачи, далеко не всегда использует классическую логику, поэтому логические модели при всех своих преимуществах имеют существенные ограничения по классам решаемых задач.

История искусственного интеллекта полна драматических событий, одним из которых стал в 1973 г. так называемый «*доклад Лайтхилла*», который был подготовлен в Великобритании по заказу Британского совета научных исследований. Известный математик Лайтхилл, никак с искусственным интеллектом профессионально не связанный, подготовил обзор состояния дел в этой области. В докладе были признаны определенные достижения, однако их уровень определялся как «разочаровывающий», а общая оценка была отрицательной с позиций практической значимости. Этот отчет отбросил европейских исследователей примерно на пять лет назад, так как финансирование работ существенно сократилось.

Примерно в это же время существенный прорыв в развитии практических приложений искусственного интеллекта произошел в США, когда в середине 1970-х гг. на смену поиску универсального алгоритма мышления пришла идея моделировать конкретные знания специалистов-экспертов. В США появились первые коммерческие системы, основанные на знаниях, или *экспертные системы* (*ЭС*). Стал применяться и новый подход к решению задач искусственного интеллекта — *представление знаний*. Были созданы две первые экспертные системы для медицины и химии — Mycin и Dendral, ставшие классическими. Существенный финансовый вклад внес и Пентагон, предлагая базировать новую программу министерства обороны США на принципах искусственного интеллекта. Уже вдогонку упущенных возможностей Европейский союз в начале 1980-х гг. объявил о глобальной программе развития новых технологий ESPRIT,

в которую была включена проблематика искусственного интеллекта.

В конце 1970-х гг. в гонку включается Япония, объявив о начале проекта машин пятого поколения, основанных на знаниях. Проект был рассчитан на десять лет и объединял лучших молодых специалистов крупнейших японских компьютерных корпораций. Для этих специалистов был специально создан новый институт ICOT и предоставлена полная свобода действий (правда, без права публикации предварительных результатов). В результате они создали достаточно громоздкий и дорогой символьный процессор, программно реализующий пролого-подобный язык, но не получивший широкого признания. Однако положительный эффект от этого проекта был очевиден. В Японии появилась значительная группа высококвалифицированных специалистов в области искусственного интеллекта, которая добилась существенных результатов в различных прикладных задачах. К середине 1990-х гг. японская ассоциация искусственного интеллекта насчитывала уже 40 тыс. человек.

Начиная с середины 1980-х гг. повсеместно происходила коммерциализация искусственного интеллекта. Росли ежегодные капиталовложения, создавались промышленные экспертные системы. Рос интерес к *самообучающимся системам*. Издавались десятки научных журналов, ежегодно собирались международные и национальные конференции по различным направлениям искусственного интеллекта, который становился одной из наиболее перспективных и престижных областей информатики.

В настоящее время различают два основных подхода к моделированию искусственного интеллекта: *машинный интеллект*, заключающийся в строгом задании результата функционирования, и *искусственный разум*, направленный на моделирование внутренней структуры системы. Моделирование систем первой группы достигается за счет использования законов формальной логики, теории множеств, графов, семантических сетей и других достижений науки в области дискретных вычислений, а основные результаты заключаются в создании экспертных систем, систем разбора

естественного языка и простейших систем управления вида «стимул — реакция». Системы же второй группы базируются на математической интерпретации деятельности нервной системы (прежде всего мозга человека) и реализуются в виде *нейроподобных сетей* на базе нейроподобного элемента — аналога нейрона [1].

1.1.2. Искусственный интеллект в России

В 1954 г. в МГУ начал работу семинар «Автоматы и мышление» под руководством академика А. А. Ляпунова (1911–1973), одного из основателей российской кибернетики. В этом семинаре принимали участие физиологи, лингвисты, психологи, математики. Принято считать, что именно в это время родился искусственный интеллект в России. Как и за рубежом, в нем выделились два основных направления — нейрокибернетика и кибернетика «черного ящика».

В 1954–1964-х гг. создавались отдельные программы и проводились исследования в области поиска решения логических задач. В ЛОМИ (Ленинградском отделении Математического института им. Стеклова) была создана *программа АЛПЕВ ЛОМИ*, автоматически доказывающая теоремы, которая основана на оригинальном обратном выводе Маслова, аналогичном методу резолюций Робинсона. Среди наиболее значимых результатов, полученных отечественными учеными в 1960-е гг., следует отметить *алгоритм «Кора»* М. М. Бонгарда, моделирующий деятельность человеческого мозга при распознавании образов. Большой вклад в становление российской школы искусственного интеллекта внесли и такие выдающиеся ученые, как М. Л. Цетлин, В. Н. Пушкин, М. А. Гаврилов, чьи ученики стали пионерами этой науки в России.

В 1965–1980-х гг. родилось новое направление ИИ — *ситуационное* (соответствующее *представлению знаний* в западной терминологии). Основателем этой научной школы стал профессор Д. А. Поспелов. Были разработаны и специальные *модели представления ситуаций* (представления знаний).

При том, что отношение к новым наукам в советской России всегда было настороженным, наука с таким «вызы-

вающим» названием тоже не избежала этой участи и была встречена в Академии наук «в штыки». К счастью, среди членов Академии наук СССР нашлись люди, не испугавшиеся столь необычного словосочетания в качестве названия нового научного направления. Однако только в 1974 г. при Комитете по системному анализу при президиуме АН СССР был создан научный совет по проблеме «Искусственный интеллект», его возглавил Д. А. Поспелов. По инициативе этого совета было организовано пять комплексных научных проектов, возглавляемых ведущими специалистами в данной области: «*Диалог*» (работы по пониманию естественного языка), «*Ситуация*» (ситуационное управление), «*Банк*» (банки данных), «*Конструктор*» (поисковое конструирование) и «*Интеллект робота*».

В 1980–1990-х гг. в нашей стране проводились активные исследования в области представления знаний, разрабатывались языки представления знаний и экспертные системы; в МГУ был создан *язык Рефал*.

В 1988 г. была сформирована Ассоциация искусственного интеллекта (АИИ), президентом которой единогласно был избран Д. А. Поспелов. В рамках этой ассоциации проводилось большое количество исследований, были организованы школы для молодых специалистов, семинары, симпозиумы, раз в два года проводились объединенные конференции, издавался научный журнал.

Следует отметить, что уровень теоретических исследований по искусственному интеллекту в России всегда был ничуть не ниже общемирового. Однако, к сожалению, начиная с 1980-х гг. на прикладных работах начинало сказываться постепенное отставание в технологии. На данный момент отставание в области разработки промышленных интеллектуальных систем составляет примерно 3–5 лет.

Основные области применения систем ИИ: доказательство теорем, игры, распознавание образов, принятие решений, адаптивное программирование, сочинение машинной музыки, обработка данных на естественном языке, обучающиеся сети (нейросети), вербальное концептуальное обучение.

1.1.3. Функциональная структура системы искусственного интеллекта

Функциональная структура системы ИИ (рис. 1.1) состоит из трех комплексов вычислительных средств. Первый из них представляет собой *исполнительную систему* (*ИС*) — совокупность средств, выполняющих программы и спроектированных с позиций эффективного решения задач; этот комплекс имеет в ряде случаев проблемную ориентацию. Второй комплекс — это совокупность *средств интеллектуального интерфейса*, имеющих гибкую структуру, которая обеспечивает возможность адаптации в широком спектре интересов конечных пользователей. Третьим комплексом средств, с помощью которых организуется взаимодействие первых двух комплексов, является *база знаний*, обеспечивающая использование вычислительными средствами первых

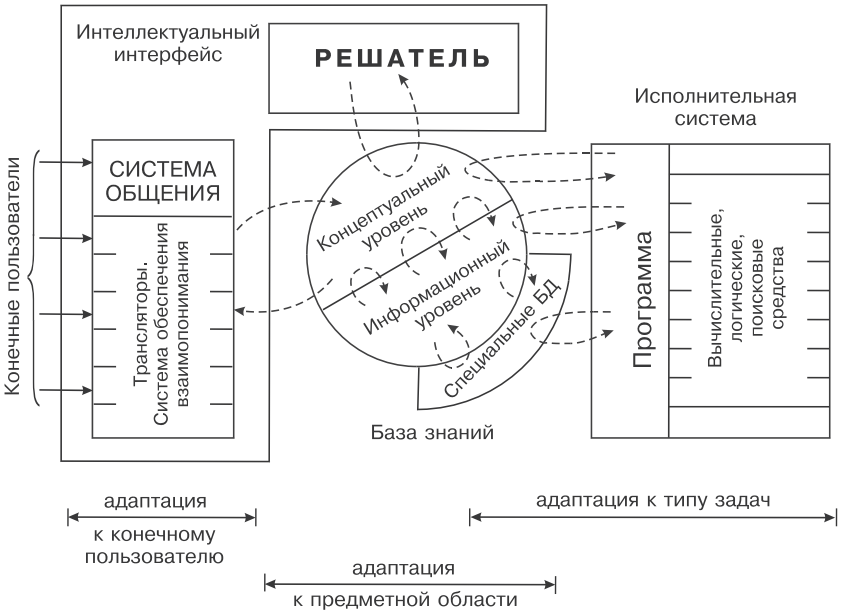


Рис. 1.1. Функциональная структура системы ИИ

двух комплексов целостной и независимой от обрабатывающих программ системы знаний о проблемной среде. Исполнительная система при этом объединяет всю совокупность средств, обеспечивающих выполнение сформированной программы; интеллектуальный интерфейс представляет собой систему программных и аппаратных средств, обеспечивающих для конечного пользователя возможность работы с компьютером для решения задач, возникающих в его профессиональной деятельности, без посредников или с незначительной их помощью; база знаний (БЗ) же занимает центральное положение по отношению к остальным компонентам вычислительной системы в целом, через нее осуществляется интеграция средств вычислительной системы, участвующих в решении задач [4].

1.2. Направления развития искусственного интеллекта

Основное направление развития ИИ — это **представление знаний и разработка систем, основанных на знаниях.** Оно связано с разработкой моделей представления знаний, созданием баз знаний, образующих ядро *экспертных систем* (*ЭС*). В последнее время это направление также включает в себя модели и методы извлечения и структурирования знаний и сливается с *инженерией знаний.*

Традиционно искусственный интеллект включает в себя **игровые интеллектуальные задачи** — шахматы, шашки и т. п. В их основе лежит один из ранних подходов — лабиринтная модель (плюс эвристики). Сейчас это скорее коммерческое направление, так как в научном плане указанные идеи считаются тупиковыми.

Следующее направление — **разработка естественно-языковых интерфейсов и машинный перевод.** В 1950-х гг. одной из популярных тем исследований искусственного интеллекта являлась область машинного перевода. Первой компьютерной программой в этой области стал переводчик с английского языка на русский. Однако использованная в нем идея пословного перевода оказалась неплодотворной.

В настоящее время для решения подобных задач используется более сложная модель, включающая в себя анализ и синтез естественно-языковых сообщений и состоящая из нескольких блоков.

Традиционное направление искусственного интеллекта, берущее начало у самых его истоков, — это **распознавание образов**. Здесь каждому объекту ставится в соответствие матрица признаков, по которой происходит распознавание этого объекта. Данное направление близко к машинному обучению и тесно связано с нейрокибернетикой.

Такое направление как **новые архитектуры компьютеров** занимается разработкой новых аппаратных решений и архитектур, ориентированных на обработку символьных и логических данных. Последние разработки в этой области посвящены компьютерам баз данных, параллельным компьютерам и нейрокомпьютерам.

Еще одно направление — **интеллектуальные роботы**. Робот — это электромеханическое устройство, предназначенное для автоматизации человеческого труда. Сама идея создания роботов — исключительно древняя (к ней можно, например, отнести еще средневековые легенды о «големах»). Само же это слово появилось в 1920-х гг. и было придумано чешским писателем Карелом Чапеком в его повести «RUR». В настоящее время в мире изготавливается более 60 тыс. роботов в год.

Роботы с жесткой схемой управления. Практически все современные промышленные роботы принадлежат к этой группе и фактически представляют собой программируемые манипуляторы.

Адаптивные роботы с сенсорными устройствами. Существуют отдельные образцы таких роботов, но в промышленности они пока не используются.

Самоорганизующиеся, или интеллектуальные роботы. Это — идеал, конечная цель развития робототехники. Основная проблема при создании интеллектуальных роботов — проблема машинного зрения.

В рамках такого направления как **специальное программное обеспечение** разрабатываются специальные языки для

решения задач невычислительного плана. Эти языки ориентированы на символьную обработку информации — Lisp, Prolog, SmallTalk, Рефал и др. Кроме них создаются пакеты прикладных программ, ориентированные на промышленную разработку интеллектуальных систем, — *программные инструментарии искусственного интеллекта*. Достаточно популярно и создание так называемых «*пустых экспертных систем*» — «оболочек», которые можно наполнять базами знаний, создавая различные экспертные системы.

Еще одна активно развивающаяся область искусственного интеллекта — **обучение и самообучение**. Это направление включает в себя модели, методы и алгоритмы, ориентированные на автоматическое накопление знаний на основе анализа и обобщения данных, обучение на примерах (или индуктивное), а также традиционные подходы распознавания образов.

Появление сети Интернет оказало существенное влияние на развитие научного направления «искусственный интеллект» и наоборот. Можно отметить следующие *существующие и перспективные применения технологий ИИ в Интернете*:

- управление порталами, крупными интернет-магазинами и другими сложными web-системами;
- маршрутизация пакетов информации при их передаче по сети;
- прогнозирование и оптимизация загрузки каналов передачи информации;
- управление сетевыми роботами[1] и др.

---

[1] *Сетевые роботы* (*спайдеры*) — интеллектуальные агенты, которые, начиная с некоторого заданного множества ссылок (URL) на web-страницы, рекурсивно обходят ресурсы сети Интернет, извлекая ссылки на новые ресурсы из уже полученных документов до тех пор, пока не будет выполнено некоторое условие остановки. Кроме того, сетевыми роботами называют специальные программные модули, которые посещают заданные web-ресурсы и автоматически скачивают размещенную на них информацию.

1.3. Данные и знания. Представление знаний в интеллектуальных системах

В рамках направления «Представление знаний» решаются задачи, связанные с формализацией и представлением знаний в памяти *интеллектуальной системы* (*ИС*). Для этого разрабатываются специальные модели представления знаний и языки для описания знаний, выделяются различные типы знаний. Изучаются также источники, из которых ИС может черпать знания, и создаются процедуры и приемы, с помощью которых возможно приобретение знаний для ИС [2]. Проблема представления знаний для ИС чрезвычайно актуальна, так как ИС — это система, функционирование которой опирается на знания о проблемной области, которые хранятся в ее памяти.

Однако чтобы моделировать знания, нужно вначале ответить на целый ряд вопросов: Что такое знания? Чем они отличаются от данных? Чем отличаются базы знаний от баз данных?

1.3.1. Данные и знания. Основные определения

Информация, с которой имеют дело компьютеры, разделяется на *процедурную* и *декларативную*. При этом процедурная информация реализуется в форме программ, которые выполняются в процессе решения задач, а декларативная информация — в форме данных, с которыми работают эти программы.

Данные — это отдельные факты, характеризующие объекты, процессы и явления в предметной области, а также их свойства [4].

Одновременно с развитием структуры ЭВМ происходило развитие информационных структур для представления данных. Появились способы их описания в виде векторов и матриц, возникли списочные и иерархические структуры. В настоящее время в языках программирования высокого уровня используются абстрактные типы данных, структура которых задается программистом. Появление *баз данных*

(*БД*) ознаменовало собой еще один шаг на пути организации работы с декларативной информацией. В базах данных могут одновременно храниться большие объемы информации, а специальные средства, образующие *систему управления базами данных* (*СУБД*), позволяют эффективно манипулировать данными, при необходимости извлекать их из БД или записывать в БД в нужном порядке, вести поиск данных в БД и пр.

По мере развития исследований в области ИС возникла *концепция знаний*, которые объединили в себе многие черты процедурной и декларативной информации.

В компьютере знания, так же как и данные, отображаются в знаковой форме — в виде формул, текста, файлов, информационных массивов и т. д. Поэтому можно сказать, что *знания — это особым образом организованные данные*. Но это слишком узкое понимание. Знания связаны с данными, основываются на них, но, в отличие от данных, представляют результат мыслительной деятельности человека, обобщают его опыт, полученный в ходе выполнения какой-либо практической деятельности. Знания добываются эмпирическим путем.

Знания — это выявленные закономерности предметной области (принципы, связи, законы), позволяющие решать задачи в этой области [4].

В системах ИИ знания являются основным объектом формирования, обработки и исследования. *База знаний* (*БЗ*) — это необходимая составляющая программного комплекса ИИ. Машины, реализующие алгоритмы ИИ, также называют машинами, основанными на знаниях.

Любая база знаний содержит в себе базу данных в качестве составляющей, но вовсе не сводится к ней. Главное отличие базы знаний от базы данных состоит в следующем: из базы данных можно извлечь лишь ту фактическую информацию, которая в нее заложена, тогда как благодаря закономерностям и связям из базы знаний можно выводить новые *факты*, которые непосредственно в нее заложены не были.

При построении баз знаний традиционные средства, основанные на числовом представлении данных, являются неэф-

фективными. Для этого используются специальные *языки представления знаний*, основанные на символьном представлении данных.

1.3.2. Модели представления знаний

Существуют десятки моделей (языков) представления знаний для различных предметных областей. Большинство из них может быть сведено к следующим классам [2]:

- семантические сети;
- фреймы;
- формальные логические модели;
- продукционные.

Семантические сети

Термин «семантическая» означает «смысловая», а сама семантика — это наука, устанавливающая отношения между символами и объектами, которые они обозначают, т. е. наука, определяющая смысл знаков.

Семантическая сеть — это ориентированный граф, вершины которого представляют собой понятия, а дуги — отношения между ними.

В качестве *понятий* обычно выступают абстрактные или конкретные объекты, а *отношения* — это связи типа: «это», «имеет частью», «принадлежит», «любит» и т. д. Характерной особенностью семантических сетей является обязательное наличие трех типов отношений:

- класс — элемент класса;
- свойство — значение;
- пример элемента класса.

Проблема поиска решения в базе знаний типа семантической сети сводится к поиску путей на графе, позволяющих от вершин с исходными данными добраться до вершин с искомыми величинами. Такой поиск можно автоматизировать.

Вывод (продуцирование) новых знаний в семантической сети может проводиться «в двух направлениях»: от известных данных к цели (результатам) или, наоборот, от цели

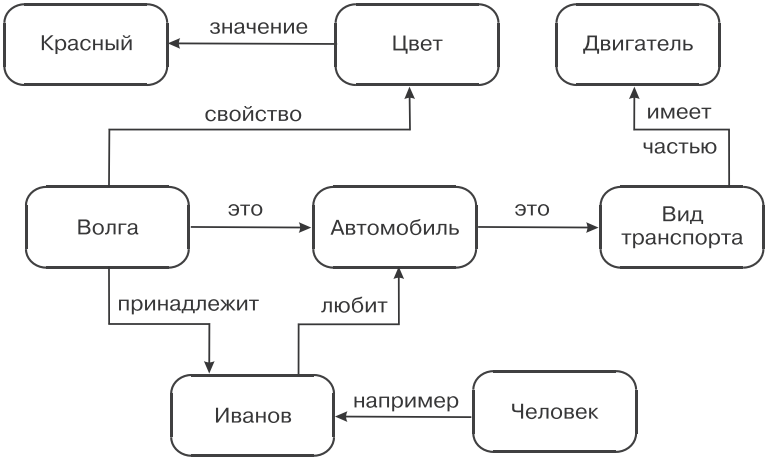


Рис. 1.2. Семантическая сеть

к известным данным. Первый способ называется *прямой волной*, *прямым поиском*, *прямой стратегией вывода*, а второй — *обратной волной*, *обратным поиском*, *обратной стратегией вывода*.

Пример: на рис. 1.2 изображена семантическая сеть, где вершинами являются понятия: «Человек», «Иванов», «Волга», «Автомобиль», «Вид транспорта», «Двигатель».

Основной недостаток подобной модели — сложность поиска вывода на семантической сети.

Фреймы

Автором теории фреймов является М. Минский. В основе этой теории лежат психологические представления о том, как мы видим, слышим и концентрируем внимание на воспринимаемом. Сам Минский считал теорию фреймов скорее «теорией постановки задач», чем продуктивной теорией, и суть ее излагал следующим образом. Каждый раз, попадая в некую ситуацию, человек вызывает в своей памяти соответствующую этой ситуации структуру, именуемую *фреймом* («frame» — «рамка»). Таким образом, **фрейм** — это единица представления знания, заполненная в прошлом, детали которой по необходимости изменяются и уточняются

применительно к ситуации. Каждый такой фрейм может быть дополнен различной информацией, касающейся способов применения этого фрейма, последствий такого применения и т. п. Например, образ жизни каждого человека — это большей частью череда типовых ситуаций, различающихся каждый раз в деталях, но в целом повторяющихся.

В психологии и философии известно понятие *абстрактного образа*. Например, слово «комната» вызывает у слышащих его примерно следующий образ комнаты: «жилое помещение с четырьмя стенами, полом, потолком, окнами и дверью, средней площади». В таком описании ничего нельзя пропустить по существу (например, убрав из него окна, мы получим уже чулан, а не комнату), но в нем есть «лакуны», или «*слоты*», — незаполненные *значения* некоторых атрибутов — конкретное количество окон, цвет стен, высота потолка, покрытие пола и др.

В теории фреймов такой образ и называется фреймом, равно как и формализованная модель для отображения такого образа.

С точки зрения пользователя различают три уровня общности фреймов:

1) *скелетный*, *пустой фрейм* (*шаблон*), превращаемый после его заполнения в общее или конкретное понятие;
2) *фрейм общего понятия* (*прототип*) — шаблон, заполненный не конкретными значениями (константами), а переменными;
3) *фрейм конкретного понятия* (*экземпляр*) — прототип, заполненный конкретными значениями (константами).

Каждому фрейму присваивается *имя*, которое должно быть уникальным во всей фреймовой системе. Описание фрейма состоит из ряда описаний, именуемых *слотами*, которым также присвоены имена (они должны быть уникальны в пределах фрейма). Каждый слот предназначен для заполнения определенной структурой данных. Значением слота может быть практически все, что угодно (числа, математические соотношения, тексты на естественном языке, программы, правила вывода, ссылки на другие слоты данного фрейма или других фреймов). При конкретиза-

ции фрейма ему и его слотам присваиваются конкретные имена и происходит заполнение слотов их значениями. Переход от исходного фрейма-прототипа к фрейму-экземпляру может быть многошаговым (за счет постепенного уточнения значений слотов).

Внутреннее (машинное) представление фрейма имеет более сложную организацию и содержит средства для создания иерархии фреймов, их взаимодействия, обмена информацией, порождения конкретных фреймов из общих и общих — из скелетных.

Важнейшим свойством теории фреймов является заимствованное из теории семантических сетей *наследование свойств*. И во фреймах, и в семантических сетях наследование происходит по *АКО-связям* («A-Kind-Of» — «это»). *Слот АКО* указывает на фрейм более высокого уровня иерархии, откуда неявно наследуются (переносятся) значения аналогичных слотов.

Например, в сети фреймов, показанной на рис. 1.3, понятие «ученик» наследует свойства фреймов «ребенок» и «человек», которые находятся на более высоком уровне иерархии. Соответственно, на вопрос: «Любят ли ученики сладкое?» следует ответ: «Да», так как этим свойством обладают все дети, что указано во фрейме «ребенок». Наследование свойств может быть частичным, например, возраст для учеников не наследуется из фрейма «ребенок», поскольку он указан явно в своем собственном фрейме.

Некоторые специалисты по искусственному интеллекту полагают, что нет необходимости специально выделять фреймовые модели в представлении знаний, так как в них объединены все основные особенности моделей остальных типов. Поэтому фреймовые модели часто рассматривают в общем контексте с сетевыми моделями. В частности, сеть фреймов можно рассматривать как семантическую сеть с блочной структурой, позволяющую реализовать альтернативные интерпретации предметных областей. Фрейм в такой сети содержит информационный и процедурный элементы, которые обеспечивают преобразование информации внутри фрейма и его связь с другими фреймами, а слоты фрейма заполняются конкретной информацией в процессе

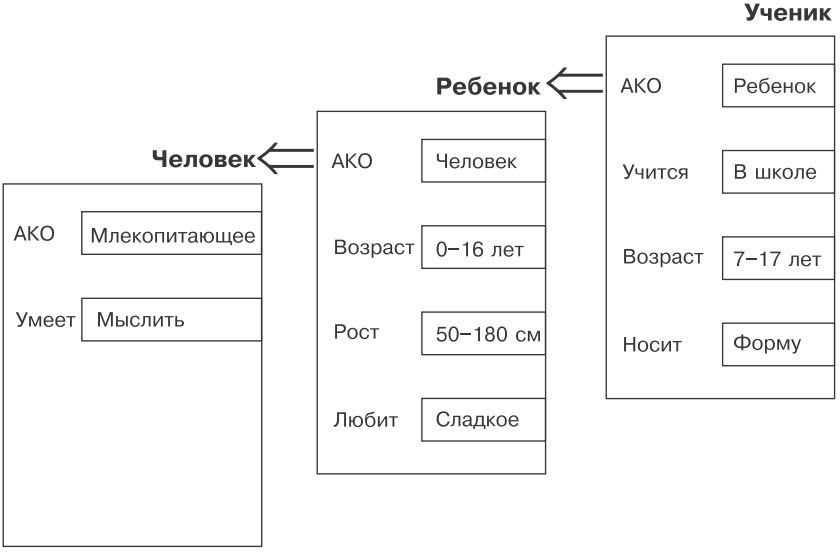


Рис. 1.3. Сеть фреймов

его функционирования. В сети фреймов могут быть также реализованы логические связки, кванторы общности и существования.

Общий вывод из сказанного выше заключается в том, что на некотором глубинном уровне все формы представления знания равносильны (в том смысле, что они универсальны, а знания, представленные в одной форме, могут быть преобразованы в другую), но не равноценны (в том смысле, что для различных предметных областей и различных задач более удобными и эффективными в вычислительном отношении оказываются различные формы представления знания). Основным же преимуществом фреймов как модели представления знаний является способность отражать концептуальную основу организации памяти человека, а также ее гибкость и наглядность.

Формальные логические модели

В основе моделей такого типа лежит *формальная система*, задаваемая четверкой множеств вида: $M = <T, P, A, B>$.

Здесь множество T — это *множество базовых элементов* различной природы (например, слов из некоторого ограниченного словаря, деталей детского конструктора, входящих в состав некоторого набора, и т. д.). Важно, что для множества T существует некоторый способ определения принадлежности или непринадлежности произвольного элемента к этому множеству. Процедура такой проверки может быть любой, но за конечное число шагов она должна давать положительный или отрицательный ответ на вопрос: является ли x элементом множества T? Обозначим эту процедуру как $П(T)$.

Множество P есть множество *синтаксических правил*. С их помощью из элементов T образуют *синтаксически правильные совокупности*. Например, из слов ограниченного словаря строятся синтаксически правильные фразы, из деталей детского конструктора с помощью гаек и болтов собираются новые конструкции и т. д. Декларируется также существование процедуры $П(P)$, с помощью которой за конечное число шагов можно получить ответ на вопрос: является ли совокупность X синтаксически правильной?

В множестве синтаксически правильных совокупностей выделяется некоторое подмножество A, элементы которого называются *аксиомами*. Как и для других составляющих формальной системы, должна также существовать процедура $П(A)$, с помощью которой для любой синтаксически правильной совокупности можно получить ответ на вопрос о принадлежности ее к множеству A.

Наконец, множество B — это множество *правил вывода*. Применяя их к элементам A, можно получать новые синтаксически правильные совокупности, к которым снова можно применять правила из B, и т. д. Так формируется *множество выводимых в данной формальной системе совокупностей*. Если имеется процедура $П(B)$, с помощью которой можно определить для любой синтаксически правильной совокупности, является ли она выводимой, то соответствующая формальная система называется *разрешимой*. Это показывает, что именно правило вывода является наиболее сложной составляющей формальной системы.

Для знаний, входящих в базу знаний, можно считать, что множество *A* образуют все информационные единицы, которые введены в базу знаний извне, а с помощью правил вывода из них выводятся новые *производные знания*. Другими словами, формальная система представляет собой генератор порождения новых знаний, образующих *множество выводимых в данной системе знаний*. Это свойство логических моделей делает их привлекательными для использования в базах знаний, оно позволяет хранить в базе лишь знания, которые образуют множество *A*, а все остальные знания получать из них по правилам вывода.

Продукционные модели

Продукционная модель, или **модель, основанная на правилах**, позволяет представить знания в виде предложений типа: «*Если* (*условие*), *то* (*действие*)». Здесь под *условием* понимается некоторое предложение — образец, по которому осуществляется поиск в базе знаний, а под *действием* — действия, выполняемые при успешном исходе поиска (они могут быть промежуточными, выступающими далее тоже как условия, либо *терминальными*, или *целевыми*, завершающими работу системы).

При использовании продукционной модели база знаний состоит из набора *правил*. Программа же, управляющая перебором этих правил, называется *машиной вывода*. Чаще всего вывод бывает *прямым* (от данных — к поиску цели) или *обратным* (от цели, для ее подтверждения, — к данным).

Машина вывода выполняет две функции:

1) просмотр существующих фактов (из рабочей памяти) и правил (из базы знаний) и добавление в рабочую память новых фактов;
2) определение порядка просмотра и применения правил.

Действие машины вывода основано на применении следующего правила: *если известно, что истинно утверждение A, и существует правило вида «если A, то B», тогда утверждение B также истинно*.

Правила срабатывают, если найдены факты, удовлетворяющие их левой части: если истинна посылка, то должно быть истинно и заключение.

Управляющий компонент машины вывода определяет порядок применения правил и выполняет четыре следующих функции:

1) сопоставление — образец правила сопоставляется с имеющимися фактами;
2) выбор — если может быть применено несколько правил, то выбирается только одно из них по какому-либо критерию;
3) срабатывание — если образец правила совпал с каким-либо фактом, то данное правило срабатывает;
4) действия — в рабочую память добавляются заключения сработавшего правила.

Эти действия циклически повторяются, пока мы не дойдем до терминального правила или у нас больше не окажется сопоставлений. Цикл работы машины вывода изображен на рис. 1.4.

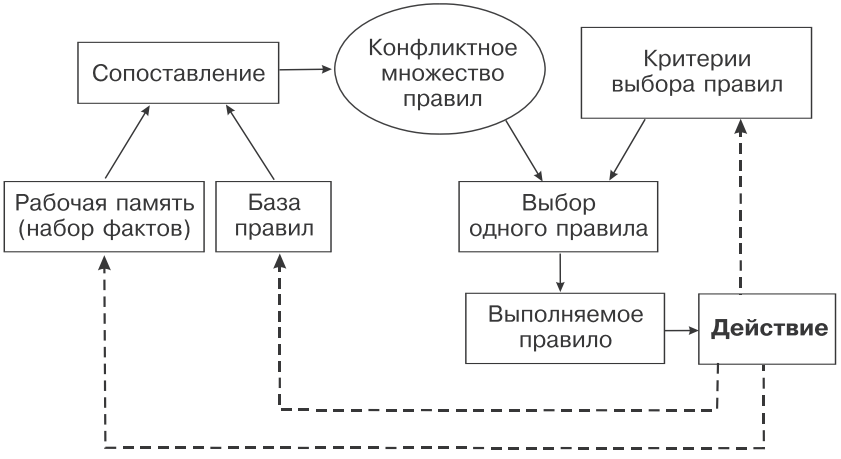


Рис. 1.4. Цикл работы машины вывода

Пример. Пусть имеется фрагмент базы знаний, состоящий из двух правил.

П1: Если «отдых — летом» и «человек — активный», то «ехать в горы».
П2: Если «любит солнце», то «отдых летом».

Предположим, что в систему поступили данные: «человек активный» и «любит солнце».

Прямой вывод — требуется, исходя из этих данных, получить ответ.

1-й проход.

Шаг 1. Пробуем **П1** — не работает (не хватает данных «отдых — летом»).
Шаг 2. Пробуем **П2** — работает; в базу поступает факт «отдых — летом».

2-й проход.

Шаг 3. Пробуем **П1** — работает, активируется цель «ехать в горы», которая и выступает как совет, который будет предложен.

Обратный вывод — требуется подтвердить выбранную цель при помощи имеющихся правил и данных.

1-й проход.

Шаг 1. Цель — «ехать в горы». Пробуем **П1** — данных «отдых — летом» нет, они становятся новой целью, и осуществляется поиск правила, где они располагаются в правой части.

Шаг 2. Цель «отдых — летом». Правило **П2** подтверждает цель и активирует ее.

2-й проход.

Шаг 3. Пробуем **П1** — подтверждается искомая цель.

Продукционная модель чаще всего применяется в промышленных экспертных системах. Она привлекает разработчиков своей наглядностью, высокой модульностью, легкостью внесения дополнений и изменений и простотой механизма логического вывода.

1.4. Экспертные системы

В начале 1980-х гг. в исследованиях по искусственному интеллекту сформировалось самостоятельное направление, получившее название «экспертные системы».

Экспертные системы (ЭС) — это сложные программные комплексы, аккумулирующие знания специалистов в конкретных предметных областях и тиражирующие этот эмпирический опыт для оказания консультаций менее квалифицированным пользователям [4]. Исследователи в области ЭС для названия своей дисциплины часто используют также термин «*инженерия знаний*», введенный Е. Фейгенбаумом и означающий «привнесение принципов и инструментария исследований из области искусственного интеллекта в решение трудных прикладных проблем, требующих знаний экспертов». Программные средства, базирующиеся на технологии экспертных систем, или инженерии знаний, получили значительное распространение в современном мире.

Экспертные системы отличаются от систем обработки данных тем, что в них в основном используются символьный (а не числовой) способ представления, символьный вывод и эвристический поиск решения (а не исполнение известного алгоритма).

Экспертные системы способны пополнять свои знания в ходе взаимодействия с экспертом. В настоящее время технология экспертных систем используется для решения различных типов задач (интерпретация, предсказание, диагностика, планирование, конструирование, контроль, отладка, инструктаж, управление) в самых разнообразных прикладных областях, таких как финансы, нефтяная и газовая промышленность, энергетика, транспорт, фармацевтическое производство, космос, металлургия, горное дело, химия, образование, целлюлозно-бумажная промышленность, телекоммуникации и связь и др.

1.4.1. Структура экспертной системы

Типичная *статическая ЭС* состоит из следующих основных компонентов (рис. 1.5):

- решатель (интерпретатор);
- рабочая память (РП), называемая также базой данных (БД);

- база знаний (БЗ);
- компоненты приобретения знаний;
- объяснительный компонент;
- диалоговый компонент.

База данных (*рабочая память*) предназначена для хранения исходных и промежуточных данных решаемой в текущий момент задачи.

База знаний (*БЗ*) в ЭС предназначена для хранения долгосрочных данных (а не текущих), описывающих рассматриваемую область, и правил, описывающих целесообразные преобразования данных этой области.

Решатель, используя исходные данные из рабочей памяти и знания из БЗ, формирует такую последовательность правил, которые, будучи примененными к исходным данным, приводят к решению задачи.

Компонент приобретения знаний автоматизирует процесс наполнения ЭС знаниями, осуществляемый пользователем-экспертом.

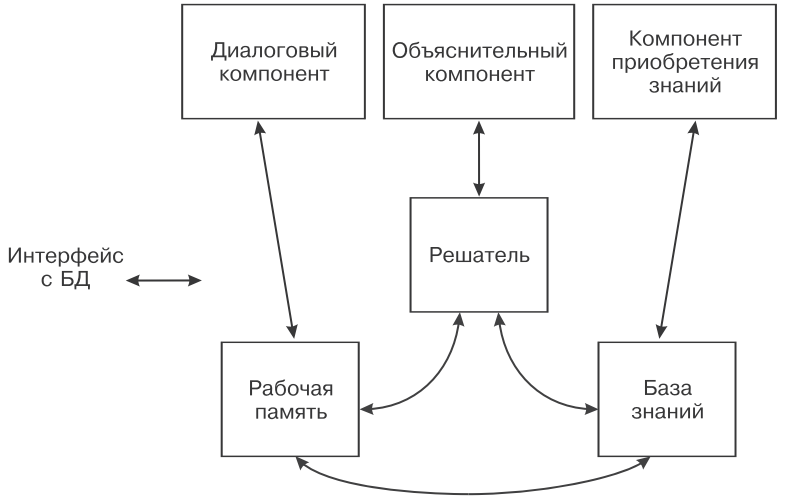


Рис. 1.5. Структура статической ЭС

Объяснительный компонент расшифровывает, как система получила решение задачи (или почему она не получила решения) и какие знания она при этом использовала, что облегчает эксперту тестирование системы и повышает доверие пользователя к полученному результату.

Диалоговый компонент ориентирован на организацию дружественного общения с пользователем как в ходе решения задач, так и в процессе приобретения знаний и объяснения результатов работы.

1.4.2. Разработка и использование экспертных систем

В разработке ЭС обычно участвуют представители следующих специальностей:

- эксперт в проблемной области, задачи которой будет решать ЭС;
- инженер по знаниям — специалист по разработке ЭС (используемую им технологию (методы) называют технологией (методами) инженерии знаний);
- программист по разработке инструментальных средств (ИС), предназначенных для ускорения разработки ЭС.

Эксперт определяет знания (данные и правила), характеризующие проблемную область, обеспечивает полноту и правильность введенных в ЭС знаний.

Инженер по знаниям помогает эксперту выявить и структурировать знания, необходимые для работы ЭС, осуществляет выбор инструментальных средств, которые наиболее подходят для данной проблемной области, и определяет способ представления знаний в этом инструментарии, выделяет и программирует (традиционными средствами) стандартные функции, типичные для данной проблемной области, которые будут использоваться в правилах, вводимых экспертом.

Программист разрабатывает ИС (если инструментарий разрабатывается заново), содержащие все основные компоненты ЭС, и осуществляет их сопряжение с той средой, в которой они будут использованы.

Необходимо отметить, что отсутствие среди участников разработки инженеров по знаниям (т. е. их замена програм-

мистами) чаще всего приводит к неудаче процесса создания ЭС либо значительно удлиняет его.

Экспертная система может работать в двух режимах: в режиме приобретения знаний и в режиме решения задачи (называемом также «режимом консультации» или «режимом использования ЭС») [2].

В режиме приобретения знаний общение с ЭС осуществляет (через посредничество инженера по знаниям) эксперт. Он, используя *компонент приобретения знаний*, наполняет систему знаниями, которые позже позволят ЭС в режиме решения самостоятельно (уже без эксперта) решать задачи из определенной проблемной области. Эксперт описывает проблемную область в виде совокупности данных и правил, где данные определяют объекты, их характеристики и значения, существующие в области экспертизы, а правила определяют способы манипулирования с данными, характерные для рассматриваемой области.

В режиме консультации общение с ЭС осуществляет конечный пользователь, которого интересует результат и (или) способ его получения. При этом данные о задаче пользователя после их обработки диалоговым компонентом поступают в рабочую память. Решатель на основе входных данных из рабочей памяти, общих данных о проблемной области и правил из БЗ формирует решение задачи. Заметим, что ЭС при решении задачи не только исполняет предписанную последовательность операции, но и предварительно формирует ее.

1.4.3. Классификация экспертных систем

Класс «экспертные системы» сегодня объединяет несколько тысяч различных программных комплексов, которые можно классифицировать по различным критериям (рис. 1.6).

Классификация по решаемой задаче

Интерпретация данных — это одна из традиционных задач для экспертных систем. Под интерпретацией понимается определение смысла данных, результаты которого должны быть согласованными и корректными. Обычно предусматри-

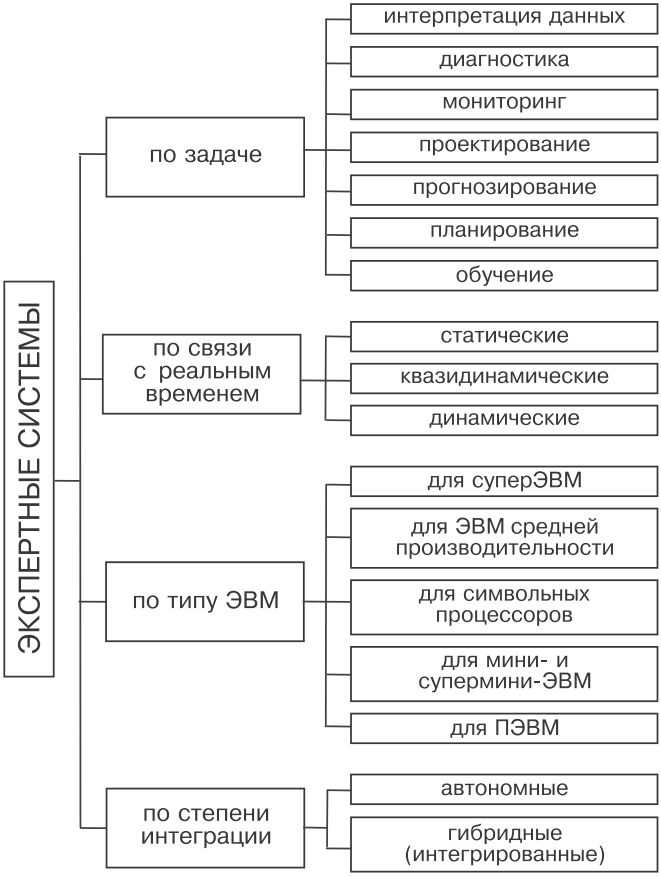


Рис. 1.6. Классификация ЭС

вается многовариантный анализ данных, например, обнаружение и идентификация различных типов океанских обитателей, определение основных свойств личности по результатам психодиагностического тестирования и др.

Диагностика. Под диагностикой понимается обнаружение неисправности в некоторой системе, где неисправность — это некоторое отклонение от нормы. Такая трактовка позволяет с единых теоретических позиций рассматривать и неисправность оборудования в технических системах, и заболевания живых организмов, и всевозможные природ-

ные аномалии. Например, это может быть диагностика ошибок в аппаратуре или математическом обеспечении ЭВМ и др.

Мониторинг. Основная задача мониторинга — непрерывная интерпретация данных в реальном масштабе времени и сигнализация о выходе тех или иных параметров за допустимые пределы. Главные проблемы при этом — пропуск тревожной ситуации и «инверсная» проблема ложного срабатывания. Примерами являются контроль за работой электростанции, помощь диспетчерам атомного реактора, контроль аварийных датчиков на химическом заводе и др.

Проектирование — состоит в подготовке спецификаций на создание объектов с заранее определенными свойствами. Под спецификацией здесь понимается весь набор необходимых документов, включая чертежи, пояснительные записки и т. д. Для организации эффективного проектирования (и в еще большей степени — перепроектирования) необходимо формировать не только сами проектные решения, но и мотивы их принятия. Таким образом, в задачах проектирования тесно связаны два основных процесса, выполняемых в рамках соответствующей ЭС: процесс вывода решения и процесс объяснения. Примером может являться проектирование конфигураций ЭВМ, синтез электрических схем и др.

Прогнозирование. Прогнозирующие системы логически выводят вероятные следствия из заданных ситуаций. В прогнозирующей системе обычно используется параметрическая динамическая модель, в которой значения параметров «подгоняются» под заданную ситуацию. Выводимые из этой модели следствия составляют основу для прогнозов с вероятностными оценками. Примеры: предсказание погоды, оценка будущего урожая, прогнозы в экономике и др.

Планирование. Под планированием понимается нахождение планов действий, относящихся к объектам, способным выполнять некоторые функции. В таких ЭС используются модели поведения реальных объектов, чтобы логически вывести последствия их планируемой деятельности. Примеры: планирование поведения робота, планирование промышленных заказов, планирование эксперимента и др.

Обучение. Системы обучения диагностируют ошибки при изучении какой-либо дисциплины с помощью ЭВМ и подсказывают правильные решения. Они аккумулируют знания о гипотетическом «ученике» и его характерных ошибках, а затем при работе способны диагностировать слабости в знаниях обучаемых и находить соответствующие средства для ликвидации пробелов в знаниях. Кроме того, подобные системы планируют акт общения с учеником, в зависимости от успехов ученика, с целью передачи знаний.

В общем случае все системы, основанные на знаниях, можно подразделить на системы, решающие *задачи анализа* и *задачи синтеза*. Основное отличие задач анализа от задач синтеза заключается в следующем: если в первых из них множество решений может быть перечислено и включено в систему, то во вторых множество решений потенциально строится из решений компонентов или подпроблем. Таким образом, задача анализа — это интерпретация данных и диагностика, а к задачам синтеза относятся проектирование и планирование. Возможны также комбинированные задачи: обучение, мониторинг, прогнозирование.

Классификация по связи с реальным временем

Статические ЭС разрабатываются в предметных областях, в которых база знаний и интерпретируемые данные не меняются во времени, стабильны. Пример — диагностика неисправностей в автомобиле.

Квазидинамические ЭС интерпретируют ситуацию, которая меняется с некоторым фиксированным интервалом времени. Примером являются микробиологические ЭС (скажем, в производстве лизина), в которых один раз в 4–5 часов снимаются лабораторные измерения с технологического процесса и анализируется динамика полученных показателей по отношению к предыдущему измерению.

Динамические ЭС работают в сопряжении с датчиками объектов в режиме реального времени с непрерывной интерпретацией поступаемых данных. Примеры — управление гибкими производственными комплексами, мониторинг в реанимационных палатах и т. д.

Классификация по типу ЭВМ

На сегодня существуют:

- ЭС для уникальных стратегически важных задач, функционирующие на суперЭВМ;
- ЭС для ЭВМ средней производительности;
- ЭС для символьных процессоров и рабочих станций;
- ЭС для мини- и супермини-ЭВМ;
- ЭС для персональных компьютеров.

Классификация по степени интеграции с другими программами

Автономные ЭС работают непосредственно в режиме консультаций с пользователем для «специфически экспертных» задач, при решении которых не требуется привлекать традиционные методы обработки данных (расчеты, моделирование и т. д.).

Гибридные ЭС представляют собой программные комплексы, объединяющие в себе стандартные пакеты прикладных программ (например, математической статистики, линейного программирования или системы управления базами данных) и средства манипулирования знаниями.

Несмотря на внешнюю привлекательность гибридного подхода, следует отметить, что разработка таких систем представляет собой задачу, на порядок более сложную, чем разработка автономной ЭС. Стыковка не просто разных пакетов, а разных *методологий* (что часто и происходит в гибридных системах) порождает целый комплекс теоретических и практических трудностей.

1.4.4. Представление знаний в экспертных системах

Первый (и основной) вопрос, который надо решить при представлении знаний, — это вопрос определения *состава знаний*, т. е. определение того, *что представлять* в экспертной системе. Второй вопрос касается того, *как представлять* знания. Необходимо отметить, что эти две проблемы являются взаимозависимыми, поскольку выбранный

способ представления может оказаться непригодным в принципе либо неэффективным для выражения некоторых знаний.

В соответствии с общей схемой статической экспертной системы (см. рис. 1.5) для ее функционирования требуются следующие знания [2]:

- о процессе решения задачи, используемые интерпретатором (решателем);
- о языке общения и способах организации диалога, используемые лингвистическим процессором (диалоговым компонентом);
- о способах представления и модификации знаний, используемые компонентом приобретения знаний;
- поддерживающие структурные и управляющие знания, используемые объяснительным компонентом.

1.4.5. Инструментальные средства построения экспертных систем

Традиционные языки программирования

В эту группу инструментальных средств входят традиционные языки программирования (C, C++, Basic, SmallTalk, Fortran и т. д.), ориентированные в основном на вычислительные алгоритмы и потому малопригодные для работы с символьными и логическими данными. Поэтому создание систем искусственного интеллекта на основе этих языков требует значительных усилий программистов. Однако большим достоинством этих языков является высокая эффективность, связанная с их близостью к базовой машинной архитектуре. Кроме того, использование традиционных языков программирования позволяет включать интеллектуальные подсистемы (например, интегрированные экспертные системы) в крупные программные комплексы общего назначения. Среди традиционных языков наиболее удобными считаются объектно-ориентированные (SmallTalk, C++). Это связано с тем, что парадигма объектно-ориентированного программирования тесно связана с фреймовой моделью представления знаний.

Языки искусственного интеллекта

Это прежде всего Lisp и Prolog — наиболее распространенные языки, предназначенные для решения задач искусственного интеллекта. Универсальность этих языков меньше, нежели у традиционных, но ее потерю языки искусственного интеллекта компенсируют богатыми возможностями для работы с символьными и логическими данными, что очень важно для задач искусственного интеллекта. На основе таких языков создаются специализированные компьютеры (например, «Лисп-машины»), предназначенные для решения задач искусственного интеллекта. Недостаток же этих языков заключается в их неприменимости для создания гибридных экспертных систем.

Специальный программный инструментарий

В эту группу программных средств искусственного интеллекта входят специальные инструментарии общего назначения. Как правило, это различные библиотеки и надстройки над языком искусственного интеллекта Lisp: KEE (Knowledge Engineering Environment), FRL (Frame Representation Language), KRL (Knowledge Representation Language), ARTS и др., позволяющие пользователям работать с заготовками экспертных систем на более высоком уровне, чем это возможно в обычных языках искусственного интеллекта.

«Оболочки»

Под «*оболочками*» (*shells*) понимают «пустые» версии существующих экспертных систем, т. е. готовые экспертные системы без базы знаний. Примером может служить система EMYCIN («Empty MYCIN» — «пустой MYCIN»), которая представляет собой незаполненную экспертную систему MYCIN. Достоинство «оболочек» в том, что они вообще не требуют работы программистов для создания готовой экспертной системы, — требуются только специалисты в предметной области для заполнения базы знаний. Однако если некоторая предметная область плохо укладывается в модель, используемую в данной конкретной «оболочке», то заполнить базу знаний в этом случае очень непросто.

1.4.6. Технология разработки экспертной системы

В настоящее время сложилась определенная технология разработки ЭС, которая включает в себя следующие шесть этапов (рис. 1.7): идентификацию, концептуализацию, формализацию, выполнение, тестирование и опытную эксплуатацию.

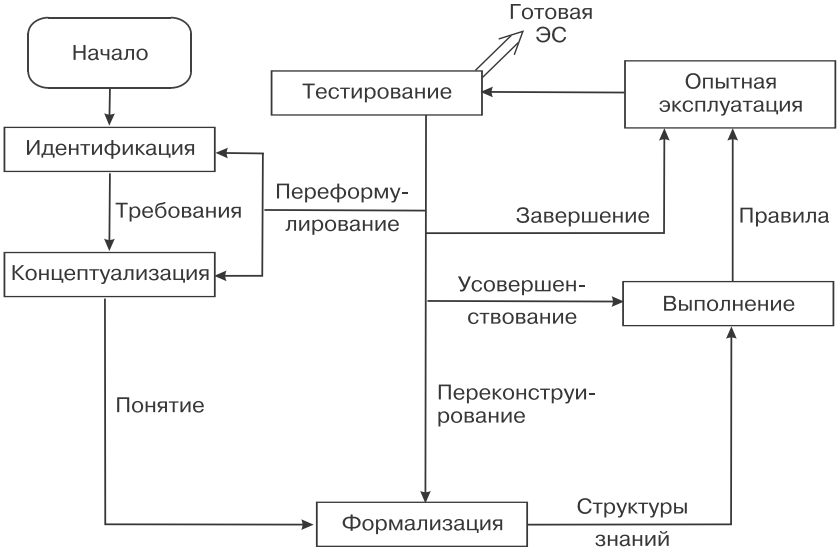


Рис. 1.7. Этапы разработки ЭС

Этап идентификации

Данный этап связан прежде всего с осмыслением задач, которые предстоит решать будущей ЭС, и с формированием требований к ней. Результатом его является ответ на вопрос, что надо сделать и какие ресурсы необходимо задействовать (идентификация задачи, определение участников процесса проектирования и их ролей, выявление ресурсов и целей).

Идентификация задачи заключается в составлении неформального (вербального) описания, в котором указываются общие характеристики задачи, подзадачи, выделяемые внутри данной задачи, ключевые понятия (объекты), их

входные (выходные) данные, предположительный вид решения, а также знания, относящиеся к решаемой задаче.

При *идентификации целей* важно отличать цели, ради которых создается ЭС, от задач, которые она должна решать. Примерами возможных целей могут быть формализация неформальных знаний экспертов, улучшение качества решений, принимаемых экспертом, автоматизация рутинных аспектов работы эксперта (пользователя), тиражирование знаний эксперта.

Этап концептуализации

На данном этапе проводится содержательный анализ проблемной области, выявляются используемые понятия и их взаимосвязи, определяются методы решения задач. Он завершается созданием *модели предметной области* (ПО), включающей в себя основные концепты и отношения. На этапе концептуализации определяются следующие особенности задачи: типы доступных данных, исходные и выводимые данные, подзадачи общей задачи, используемые стратегии и гипотезы, виды взаимосвязей между объектами ПО, типы используемых отношений (иерархия, причина — следствие, часть — целое и т. п.), процессы, используемые в ходе решения, состав знаний, используемых при решении задачи, типы ограничений, накладываемых на процессы, используемые в ходе решения, состав знаний, используемых для обоснования решений.

Существуют два подхода к процессу построения модели предметной области, которая и является целью разработчиков ЭС на этапе концептуализации. *Признаковый*, или *атрибутивный подход* предполагает наличие полученной от экспертов информации в виде троек «объект — атрибут — значение атрибута», а также наличие обучающей информации. Этот подход развивается в рамках направления, получившего название «*формирование знаний*», или «*машинное обучение*» (*machine learning*).

Второй подход, называемый *структурным*, или *когнитивным*, осуществляется путем выделения элементов предметной области, их взаимосвязей и семантических отношений.

Этап формализации

Далее все ключевые понятия и отношения выражаются на некотором формальном языке, который либо выбирается из числа уже существующих, либо создается заново. Другими словами, на данном этапе определяются состав средств и способы представления декларативных и процедурных знаний, осуществляется это представление и, в итоге, формируется описание решения задачи ЭС на предложенном (инженером по знаниям) формальном языке.

Результатом этапа формализации является описание того, как рассматриваемая задача может быть представлена в выбранной или разработанной формальной системе. Сюда относится указание способов представления знаний (фреймы, сценарии, семантические сети и т. д.), а также определение способов манипулирования этими знаниями (логический вывод, аналитическая модель, статистическая модель и др.) и интерпретации знаний.

Этап выполнения

Цель этого этапа — создание одного или нескольких *прототипов* ЭС, решающих требуемые задачи. Затем по результатам их тестирования и опытной эксплуатации создается *конечный продукт*, пригодный для промышленного использования. Разработка прототипа состоит в программировании его компонентов (или их выборе из известных инструментальных средств) и в наполнении базы знаний.

Главное в создании прототипа заключается в том, чтобы он обеспечивал проверку адекватности идей, методов и способов представления знаний решаемым задачам. Создание первого прототипа должно подтвердить, что выбранные методы решений и способы представления пригодны для успешного решения по крайней мере части задач из актуальной предметной области, а также должно продемонстрировать тенденцию к получению высококачественных и эффективных решений для *всех* задач предметной области по мере увеличения объема знаний.

После разработки первого прототипа («ЭС-1») круг предлагаемых для решения задач расширяется, собираются пожелания и замечания пользователей, которые должны быть учтены в очередной версии системы («ЭС-2» и т. д.).

Этап тестирования

В ходе данного этапа производится оценка выбранного способа представления знаний в ЭС в целом. Для этого инженер по знаниям подыскивает примеры, обеспечивающие проверку всех возможностей разработанной ЭС.

Различают следующие источники неудач в работе системы: тестовые примеры, ввод-вывод, правила вывода и управляющие стратегии.

Показательные тестовые примеры являются наиболее очевидной причиной неудачной работы ЭС. В худшем случае тестовые примеры могут оказаться вообще не из той предметной области, на которую рассчитана данная ЭС, однако гораздо чаще множество тестовых примеров оказывается слишком однородным и не охватывает всю требуемую предметную область. Поэтому при подготовке тестовых примеров необходимо классифицировать их по подпроблемам предметной области, выделяя стандартные случаи, определяя границы трудных ситуаций и т. п.

Ввод-вывод характеризуется данными, приобретенными в ходе диалога с экспертом, и заключениями, предъявленными ЭС в ходе объяснений. Методы приобретения данных могут не давать требуемых результатов из-за того, что, например, экспертной системой были заданы неправильные вопросы или собрана не вся необходимая информация. Кроме того, вопросы системы могут быть трудными для понимания, многозначными и не соответствующими знаниям пользователя. Ошибки при вводе могут возникать также и из-за неудобного для пользователя входного языка. В ряде приложений для пользователя удобен ввод не только в печатной, но и в графической или звуковой форме.

Выходные сообщения (*заключения*) *системы* могут оказаться непонятными пользователю (или эксперту) по разным причинам. Например, их может быть слишком много или, наоборот, слишком мало. Также причиной подобных ошибок может являться неудачная организация, упорядоченность заключений или не подходящий для пользователя уровень абстракций с непонятной ему лексикой.

Наиболее распространенный источник ошибок в рассуждениях касается *правил вывода*. Одна важная причина здесь

часто кроется в отсутствии учета взаимозависимости сформированных правил. Другая причина заключается в ошибочности, противоречивости и/или неполноте используемых правил. Если неверна посылка правила, то это может привести к употреблению правила в неподходящем контексте. Если ошибочно действие правила, то трудно предсказать конечный результат. Правило также может быть ошибочно, если при корректности его условия и действия нарушено соответствие между ними.

Нередко к ошибкам в работе ЭС приводят и применяемые *управляющие стратегии*. Недостатки в них могут привести к чрезмерно сложным заключениям и объяснениям ЭС.

Этап опытной эксплуатации

На этом этапе проверяется пригодность ЭС для конечного пользователя, которая определяется в основном удобством работы с ней и ее полезностью. Под *полезностью ЭС* при этом понимается ее способность в ходе диалога определять потребности пользователя, выявлять и устранять причины неудач в работе ЭС, а также удовлетворять потребности пользователя (т. е. решать поставленные задачи). В свою очередь, *удобство работы с ЭС* подразумевает естественность взаимодействия с ней (общение в привычном, не утомляющем пользователя виде), гибкость ЭС (способность системы настраиваться на различных пользователей, а также учитывать изменения в квалификации одного и того же пользователя) и устойчивость системы к ошибкам (способность не выходить из строя при ошибочных действиях неопытного пользователя).

В ходе разработки ЭС почти всегда осуществляется ее *модификация*. Выделяют следующие виды модификации системы: переформулирование понятий и требований, переконструирование представления знаний в системе и усовершенствование прототипа.

Контрольные вопросы и задания к главе 1

1. Сформулируйте цель проведения научных и технических разработок в области искусственного интеллекта.
2. Назовите два основных направления искусственного интеллекта. Какова основная идея каждого из этих направлений?
3. Сформулируйте суть модели лабиринтного поиска.
4. Что такое эвристическое программирование?
5. Назовите два основных подхода к моделированию искусственного интеллекта.
6. Назовите основные области применения систем искусственного интеллекта.
7. Назовите три известных вам комплекса вычислительных средств систем искусственного интеллекта. Каково их назначение?
8. Перечислите направления развития искусственного интеллекта.
9. Что такое данные?
10. Что такое знания?
11. В чем состоит основное отличие базы знаний от базы данных?
12. Что такое семантическая сеть? Приведите пример семантической сети.
13. Как осуществляется вывод новых знаний в семантической сети?
14. Что такое фрейм? Приведите пример фрейма.
15. Назовите три уровня общности фреймов.
16. Как представить знания в продукционной модели? Приведите пример продукционной модели.
17. Что называют машиной вывода? Каковы функции машины вывода?
18. Опишите цикл работы машины вывода.
19. Что такое экспертная система?
20. В чем состоит отличие экспертных систем от систем обработки данных?
21. Перечислите основные компоненты статической экспертной системы. Для чего предназначен каждый из этих компонентов?

22. Назовите два возможных режима работы экспертной системы. Как экспертная система работает в каждом из этих режимов?
23. Классифицируйте экспертные системы по решаемой задаче.
24. Классифицируйте экспертные системы по связи с реальным временем.
25. Классифицируйте экспертные системы по типу ЭВМ.
26. Классифицируйте экспертные системы по степени интеграции.
27. Назовите известные вам инструментальные средства для построения экспертных систем.
28. Перечислите этапы технологии разработки экспертных систем. Какова цель каждого из этих этапов?

Литература к главе 1

1. *Гаврилов Т. А., Хорошевский В. Ф.* Базы знаний и интеллектуальные системы: Учебник. — СПб.: Питер, 2000.
2. *Кузнецов В. Е.* Представление в ЭВМ неформальных процедур: продукционные системы / С послесловием Д. А. Поспелова. — М.: Наука, 1989.
3. *Любарский Ю. Я.* Интеллектуальные информационные системы. — М.: Наука, 1990.
4. Справочник по искусственному интеллекту. В 3-х тт. / Под ред. Э. В. Попова и Д. А. Поспелова. — М.: Радио и связь, 1990.
5. *Хант Э.* Искусственный интеллект. — М.: Мир, 1978.

ГЛАВА 2

ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

2.1. Методологии программирования

В этом разделе рассказывается об основных *методологиях программирования* — о совокупности методов, применяемых в жизненном цикле программного обеспечения и объединенных общим философским подходом.

На сегодня существует не так много методологий, особенно *полных*, т. е. учитывающих *все* стадии жизненного цикла программного обеспечения. Именно методология определяет, какие языки и системы будут применяться для разработки программного обеспечения и какой технологический подход будет при этом использован.

Наш подход к методологиям заключается в том, что существует некоторое *ядро методологии* со своими методами, которое уточняется некоторыми дополнительными особенностями. Ядра же методологий определяются способом описания алгоритмов. Перечислим основные ядра методологий и посвятим следующие разделы книги их подробному анализу:

- методология императивного программирования;
- методология объектно-ориентированного программирования;
- методология функционального программирования;
- методология логического программирования.

Некоторые методологии имеют алгоритмическое происхождение. Четыре главных модели алгоритма математически эквивалентны, но на практике они породили разные направления в программировании, в том числе некоторые основные методологии [5]:

- абстрактные вычислительные машины Тьюринга и Поста — определяют методологию императивного программирования;

- рекурсивные функции Гильберта и Аккермана — от них унаследовала свои идеи и конструкции методология структурного программирования;
- лямбда-исчисление Черча, Шейнфинкеля и Карри — эти идеи активно развиваются в методологии функционального программирования;
- нормальные алгоритмы Маркова — эта модель послужила основой логического программирования и обработки символьной информации.

2.1.1. Методология императивного программирования

Данный подход характеризуется принципом последовательного изменения состояния вычислителя пошаговым образом. При этом управление изменениями полностью определено и полностью контролируемо.

Императивное программирование — это исторически первая аппаратно поддерживаемая методология программирования. Она ориентирована на классическую фон Неймановскую модель, остававшуюся долгое время единственной аппаратной архитектурой, получившей широкое практическое применение.

Императивное программирование основано на описании последовательного изменения состояний вычислителя. Если под вычислителем понимать современный компьютер, то его состоянием будут значения всех его ячеек памяти, состояние процессора (в том числе значение указателя текущей команды) и всех сопряженных устройств. Единственная структура данных здесь — это последовательность ячеек (пар «адрес — значение») с линейно упорядоченными адресами.

Языки, поддерживающие данную вычислительную модель, являются компактным средством описания функции переходов между состояниями вычислителя.

В качестве математической модели императивное программирование использует *машину Тьюринга—Поста* — абстрактное вычислительное устройство, предложенное на заре компьютерной эры для описания алгоритмов.

Основным синтаксическим понятием здесь является *оператор*, причем возможно несколько разновидностей (групп)

таких операторов. Первая группа — это *атомарные операторы*, у которых никакая часть не является самостоятельным оператором (например, операторы присваивания, безусловного перехода, вызова процедуры и т. п.). Вторая группа — *структурные операторы*, объединяющие другие операторы в новый, более крупный оператор (например, составной оператор, оператор выбора, цикла и т. д.).

Операторы исполняются в порядке, предписанном объемлющим их структурным оператором. Если это составной оператор, то входящие в него операторы исполняются в том порядке, в котором они записаны.

Традиционное средство структурирования — *подпрограмма* (процедура или функция). Подпрограммы имеют параметры и локальные определения и могут быть вызваны рекурсивно. Функции же возвращают значения как результат своей работы.

Императивные языки программирования манипулируют данными в пошаговом режиме, используя последовательные инструкции и применяя их к тем или иным данным. Наиболее известные и распространенные императивные языки программирования — Fortran, Algol, Pascal, C.

Императивное программирование наиболее пригодно для решения задач, в которых последовательное исполнение каких-либо команд является естественным. Примером может служить управление современными аппаратными средствами: поскольку практически все современные компьютеры императивны, эта методология позволяет получать достаточно эффективный исполняемый код. Однако с ростом сложности задачи императивные программы становятся все менее читабельными.

Важнейшим развитием императивной методологии является *методология структурного императивного программирования* — подход, заключающийся в отказе от использования глобальных данных и оператора безусловного перехода, в разработке модулей с сильной связностью и обеспечении их независимости от других модулей. Этот подход базируется на двух основных принципах:

- последовательная декомпозиция алгоритма решения задачи сверху вниз;
- использование структурного кодирования.

Создателем структурного подхода считается Эдсгер Дейкстра. Ему также принадлежит попытка (к сожалению, совершенно неприменимая для массового программирования) соединить структурное программирование с методами доказательства правильности программ [1].

2.1.2. Методология объектно-ориентированного программирования

Этот подход использует *объектную декомпозицию*, при которой статическая структура системы описывается в терминах объектов и связей между ними, а поведение системы описывается в терминах обмена сообщениями между объектами.

На возникновение объектного мышления оказали влияние моделирование и представление данных, графические пользовательские интерфейсы и системное программирование (с понятием «процесс»). Исследования в области моделирования реальных систем привели к необходимости создания средств естественного описания сущностей, которые в них встречаются, — *объектов* и *событий*. Позже оказалось, что такие концепции, как *инкапсуляция* (абстрактные типы данных), *наследование* и *полиморфизм*, являются весьма полезным дополнением к традиционному структурному программированию. Возможность их достаточно эффективной реализации привела к созданию широко распространенных в наши дни *объектно-ориентированных языков программирования*.

Вычислительная модель «чистого» объектно-ориентированного программирования (ООП) явно поддерживает только одну операцию, которой является посылка объекту сообщения. Такие сообщения могут иметь параметры, являющиеся объектами; само сообщение также является объектом.

Объект имеет набор *обработчиков сообщений* (*методов*). У объекта также есть *поля* — «персональные» переменные для данного объекта, значениями которых являются ссылки на другие объекты. В одном из полей объекта хранится ссылка на *объект-предок*, которому переадресуются все сообщения, не обрабатываемые данным объектом. Структуры, описывающие обработку и переадресацию сообщений, обыч-

но выделяют в отдельный объект, называемый *классом* данного объекта, тогда как сам объект называют *экземпляром* указанного класса.

В синтаксисе «чистых» объектно-ориентированных языков все может быть записано в форме посылки сообщений объектам. Класс в объектно-ориентированных языках описывает структуру и функционирование множества объектов с аналогичными характеристиками, атрибутами и поведением. Объект здесь естественным образом принадлежит к некоторому классу и обладает своим собственным внутренним состоянием. Методы же представляют собой функциональные свойства, которые можно активизировать.

В объектно-ориентированном программировании определяют три основных свойства:

- *инкапсуляция* — скрытие информации и комбинирование данных и функций, которые аналогичны абстрактным типам данных;
- *наследование* — построение иерархии порожденных объектов с возможностью для каждого такого объекта, относящегося к иерархии, доступа к коду и данным всех порождающих объектов;
- *полиморфизм* (*полиморфизм включения*) — присваивание действию одного имени, которое затем разделяется вверх и вниз по иерархии объектов, причем каждый объект иерархии выполняет это действие способом, подходящим именно ему.

Объектно-ориентированные языки программирования содержат конструкции, позволяющие определять объекты, принадлежащие классам и обладающие свойствами инкапсуляции, наследования и полиморфизма.

Наиболее распространенные объектно-ориентированные языки — C++, Object Pascal, Java, Visual Basic и др.

Данная методология является мощным средством для моделирования отношений между объектами практически в любой предметной области. Особенно удобно и легко в форме объектов выражать взаимодействие между различными элементами графического интерфейса пользователя.

2.1.3. Методология функционального программирования

Данная методология представляет собой способ составления программ, в которых единственным действием является вызов функции, единственным способом расчленения программы на части является введение имени для функции и задание для этого имени выражения, вычисляющего значения функции, а единственным правилом композиции является оператор *суперпозиции* функции.

Функциональная методология — одна из старейших. По происхождению она тесно связана с *лямбда-исчислением*, изобретенным еще в начале 1930-х гг. логиком Алонзо Черчем (Alonzo Church). Для многих людей функциональная методология стала ассоциироваться с языком Lisp, созданным Джоном Маккарти (John McCarthy) в конце 1950-х гг. В то же время эта методология в основном используется теоретиками программирования и является средством лабораторных исследований искусственного интеллекта.

Метод аппликативности заключается в том, что программа есть выражение, составленное из применений функций к их аргументам. Программа при этом состоит из совокупности определений функций, представляющих собой вызовы других функций и предложений, управляющих последовательностью вызовов. Данный метод поддерживается *концепцией функции*.

Метод рекурсивного поведения заключается в самоповторяющемся поведении, возвращающемся к самому себе. Данный метод поддерживается *концепцией рекурсии*.

Метод настраиваемости заключается в том, что можно легко порождать новые программные объекты по образцу как значения соответствующих выражений (применяя порождающую функцию к параметрам образца). Этому способствует то, что в идеале не только программа, но и любой программный объект является *выражением*.

Функциональное программирование представляет собой одну из альтернатив императивному подходу. В функциональном программировании отсутствует понятие времени, программы являются выражениями, а исполнение программ заключается в вычислении этих выражений. В качестве

математической модели функциональное программирование использует *лямбда-исчисление Черча*.

Функциональные языки программирования — это языки, в которых, как уже говорилось, единственным действием является вызов функции, единственным способом расчленения программы на части является введение имени для функции и задание для этого имени выражения, вычисляющего значения функции, а единственным правилом композиции — оператор суперпозиции функции. Основная специфика функциональных языков программирования заключается в том, что функции обмениваются между собой данными *непосредственно*, т. е. без использования промежуточных переменных и присваиваний, а переменные, однажды получив значение, никогда его не изменяют. (Присваивание же, например, в императивной методологии, — это конструкция, «вносящая время» в процесс вычисления: правая часть присваивания должна быть вычислена, и только после этого ее значение будет связано с левой частью.) В результате в функциональных языках циклы заменяются аппаратом рекурсивных функций.

Функциональное программирование обычно применяется для решения задач, которые трудно сформулировать в терминах последовательных операций. В эту категорию попадают и практически все задачи, связанные с искусственным интеллектом, — такие как обработка естественного языка, экспертные (консультирующие) системы, проблемы зрительного восприятия и многие другие.

2.1.4. Методология логического программирования

Согласно данному подходу, программа содержит описание проблемы в терминах фактов и логических формул, а решение проблемы система находит с помощью механизмов логического вывода.

Логическое программирование появилось в конце 1960-х гг., когда Корделл Грин (Cordell Green) предложил использовать *резолюцию* как основу логического программирования. Позже, в 1970-х гг., Алан Колмероэ (Alain Colmerauer) создал *язык логического программирования Prolog*. В основе логи-

ческих языков лежит *теория хорновских дизъюнктов*. Логическое программирование пережило пик популярности в середине 1980-х гг., когда оно было положено в основу проекта разработки программного и аппаратного обеспечения вычислительных систем пятого поколения [7].

В логике *теории* задаются при помощи *аксиом* и *правил вывода*. То же самое мы имеем и в базисном языке логического программирования Prolog, — только аксиомы здесь принято называть *фактами*, а правила вывода ограничивать по форме до так называемых «*дизъюнктов Хорна*» — утверждений вида `A <= B1& ... &Bn`. В языке Prolog такие утверждения принято записывать следующим образом:

```
a :- b1, ..., bn
```

Факты (они же аксиомы) представляются в языке Prolog как правила с пустой «посылкой», например, `a`.

Обычно Prolog-система работает в форме диалога с пользователем. Утверждение, которое требуется доказать, вводится с клавиатуры. Однако компилирующие версии трансляторов Prolog могут располагать и специальными синтаксическими средствами для задания утверждений, которые требуется доказать. Такие утверждения в Prolog принято называть *целями*.

Prolog-система использует для доказательства утверждений *метод унификации* и *метод резолюций*. **Унификация** — это сопоставление двух произвольных термов, содержащих переменные, чтобы определить, можно ли присвоить этим переменным такие значения, чтобы получились два одинаковых терма. **Метод резолюций** же заключается в последовательном доказательстве отдельных утверждений, входящих в посылку дизъюнкта Хорна, для доказательства его следствия. Например, применение метода резолюций к правилу `a :- b, c.` и утверждению `a` приведет к последовательному доказательству утверждений `b` и `c`. Метод резолюций имеет прямой аналог в обычной логике высказываний, — это *правило modus ponens*, согласно которому $(A \,\&\, A \Rightarrow B) \Rightarrow B$.

Логические языки программирования обычно содержат в себе конструкции, позволяющие выполнить описание проблемы в терминах фактов и логических формул, тогда

как собственно решение проблемы выполняет программная система с помощью *механизмов логического вывода*. Неотъемлемой частью такого языка программирования является и *механизм конструктивного вывода целевого утверждения*, основанный на строгих математических моделях.

Язык Prolog является родоначальником целого семейства подобных ему языков, в котором можно выделить три ветви:

- модификации языка (использование более мощных логических средств, внесение модульности и т. п.);
- функциональное направление (комбинация с функциональными языками);
- направление, связанное с использованием параллельных вычислений (так как логическое программирование по сути своей параллелизуемо).

Отметим, что класс задач логического программирования практически совпадает с классом задач функционального программирования.

2.1.5. Методология программирования в ограничениях

Рассмотрим теперь другие, менее распространенные методологии, применяемые при решении задач искусственного интеллекта.

Методология программирования в ограничениях — это подход, в котором в программе для искомого решения определяется тип данных, предметная область и ограничения на его значение. Решение же отыскивается системой. Данная методология предполагает двухуровневую архитектуру, интегрирующую компонент ограничений и программный компонент. *Компонент ограничений* при этом обеспечивает основные операции и состоит из *системы выводов на фундаментальных свойствах системы ограничений*. Операции, окружающие компонент ограничений, реализуются программно-языковым компонентом.

Методология программирования в ограничениях возникла в начале 1980-х гг. как перспективная область исследо-

ваний на пересечении символьных вычислений, искусственного интеллекта, исследования операций и интервальной арифметики.

Программирование в ограничениях — это программирование в терминах постановок задач. **Постановка задачи** представляет собой конечный набор переменных $V = \{v[1], \ldots, v[n]\}$, соответствующих им конечных (перечислимых) множеств значений $D = \{D[1], \ldots, D[n]\}$ и набор ограничений $C = \{C[1], \ldots, C[m]\}$. При этом ограничения представлены как утверждения, в которые в качестве «параметров» входят переменные из некоторого подмножества $v[j]$, $j = 1 \ldots m$ заданного набора V. Решение такой задачи есть набор значений переменных, удовлетворяющий всем ограничениям $C[j]$.

Семантически исполнение программы здесь рассматривается как нахождение значений переменных. Языки программирования в ограничениях также получили наибольшую известность в 1980-х гг. Язык программирования «УТОПИСТ» («Универсальные Текстовые ОПИСания Терминов») предназначен для описания понятий и задач. Этот язык является базовым для инструментальной системы программирования «ПРИЗ». Он имеет процедурную часть (поскольку при описании задачи иногда приходится описывать действия), но основная его выразительность достигается за счет описаний.

К классу задач программирования в ограничениях относятся задачи исследования операций и искусственного интеллекта. В них часто используется некоторое *пространство решений*, сужением которого и достигается необходимый результат. Такое сужение исходного пространства решений можно естественным образом представить как ограничения.

2.1.6. Методология нейросетевого программирования

Методология нейросетевого программирования — это подход, заключающийся в том, что на основе знаний, полученных от экспертов, создается программа на *нейронном языке программирования*, которая затем компилируется в эквивалентную нейронную сеть из аналоговых нейронов. Более подробно методологию нейросетевого программирования мы рассмотрим в главе 3.

2.2. Краткое введение в исчисление предикатов и доказательство теорем

Высказывание — это логическое утверждение, которое может быть истинным или ложным. Оно состоит из объектов и отношений между ними [5].

Символьную логику (symbolic logic) можно использовать для решения трех основных задач формальной логики: для выражения высказываний, выражения отношений между высказываниями и описания способов вывода новых высказываний из других высказываний, считающихся истинными.

Объекты в высказываниях логического программирования представляются простыми термами, являющимися либо константами, либо переменными. **Константа** — это символ, представляющий некий объект. **Переменная** — это символ, который может представлять разные объекты в разное время, хотя в некотором смысле переменная в этом контексте намного ближе к математическому пониманию переменной, чем к переменным в императивных языках программирования.

Простейшие высказывания, называемые **атомарными высказываниями**, состоят из составных термов. **Составной терм** — это элемент математического отношения, формально записанного в виде математической функции.

Составной терм состоит из двух частей: *функтора* (*functor*), представляющего собой функциональный символ, называющий отношение, и *упорядоченного списка параметров*. Примеры:

```
человек(Джек)
любит(Боб, стейк)
```

Высказывания можно формулировать двумя способами: в первом случае высказывание считается истинным, а во втором истинность высказывания требуется установить. Иными словами, высказывания можно формулировать либо как *факты*, либо как *запросы*. Высказывания в примерах, приведенных выше, могут представлять собой как факты, так и запросы.

Исчисление предикатов дает нам метод для выражения *совокупностей высказываний*. Использовать совокупности высказываний — это означает определять, можно ли вывести из них какие-нибудь интересные или полезные факты. Последнее аналогично работе математиков, старающихся открыть новые теоремы, которые можно вывести из уже известных аксиом и теорем.

В 1950-х — начале 1960-х гг. процессу автоматического доказательства теорем уделялось большое внимание. Одним из крупных научных достижений в этой области было открытие *принципа резолюции* (*resolution*) Аланом Робинсоном (Alan Robinson) из Сиракузского университета.

Резолюция (resolution) — это правило логического вывода, позволяющее вычислять выводимые высказывания по заданным высказываниям, обеспечивая таким образом метод, имеющий потенциальные приложения в области автоматического доказательства теорем [5]. Резолюция была изобретена для применения к высказываниям в дизъюнктивной форме.

Концепция резолюции заключается в следующем. Предположим, что нам даны два высказывания в следующих формах:

```
старше(Джоанна,Джек)  => мать(Джоанна,Джек)
мудрее(Джоанна,Джек)  => старше(Джоанна,Джек)
```

По этим высказываниям, используя резолюцию, можно построить новое высказывание:

```
мудрее(Джоанна,Джек)  => мать(Джоанна,Джек)
```

Механизм этой резолюции прост: термы в левых частях этих двух высказываний объединяются с помощью логической операции «И», образуя левую часть нового высказывания. Затем точно так же формируется правая часть нового высказывания:

```
старше(Джоанна,Джек)  И мудрее(Джоанна,Джек)=>
=>мать(Джоанна,Джек)  И старше(Джоанна,Джек)
```

А далее терм, появляющийся в обеих частях нового высказывания, удаляется из них («сокращается»):

```
мудрее(Джоанна,Джек) => мать(Джоанна,Джек)
```

Этот процесс применим и тогда, когда высказывания содержат составные термы в одной или в обеих частях. Левая часть нового выведенного высказывания вначале содержит все термы левых частей двух заданных высказываний; новая правая часть также конструируется аналогично. Затем термы, появляющиеся в обеих частях нового высказывания, удаляются.

В действительности резолюция представляет собой более сложный процесс, чем показано в этом простом примере. В частности, наличие переменных в высказываниях требует выполнять в процессе резолюции поиск значений этих переменных, что приводит к процессу *поиска соответствий*. Этот процесс определения полезных значений переменных называется **унификацией**, а временное присваивание значений переменным с целью унификации называется **конкретизацией**.

Обычно во время резолюции переменная конкретизируется неким значением, не полностью удовлетворяющим требуемому соответствию, а затем следует отменить последнее действие и конкретизировать эту переменную новым значением. (Мы будем изучать унификацию более подробно в контексте языка Prolog.)

Крайне важное свойство резолюции — ее способность обнаруживать любое противоречие в заданной совокупности высказываний. Это свойство позволяет использовать резолюцию для доказательства теорем следующим образом. В терминах исчисления высказываний мы можем представить себе доказательство теоремы как заданную совокупность соответствующих высказываний, в которых отрицание теоремы само по себе формулируется в виде нового высказывания. Таким образом, теорема отрицаема, так что можно использовать резолюцию для доказательства этой теоремы, обнаружив противоречие. Это — доказательство от противного. Обычно исходные высказывания называются *гипотезами*, а отрицание теоремы — *целью*.

2.3. Процесс логического вывода в языке Prolog

В этом разделе изучается резолюция в языке Prolog. Для эффективного использования языка Prolog требуется, чтобы программист точно знал, что именно делает система языка Prolog с его программой.

Запросы в Prolog называются *целями* (*goal*). Когда цель представляет собой составной оператор, каждый из входящих в нее фактов (структур) называется *подцелью* (*subgoal*). Для доказательства того, что цель истинна, процесс логического вывода должен найти цепочку правил логического вывода и/или факты в базе данных, которые связывают цель с одним или несколькими другими фактами в базе данных [5].

Так как доказательство подцели осуществляется с помощью поиска соответствия между высказываниями, иногда его называют *сопоставлением*.

Рассмотрим следующий запрос:

```
man(bob).
```

Эта цель имеет простейший вид. Она относительно проста для того, чтобы резолюция определила, истинно ли это высказывание или ложно: образец этой цели сравнивается с фактами и правилами в базе данных. Если факт

```
man(bob).
```

содержится в базе данных, то доказательство является тривиальным. Допустим, однако, что в базе данных содержатся приведенные ниже факт и правило логического вывода:

```
father (bob).
man(X) :- father(X).
```

Тогда можно потребовать, чтобы система языка Prolog нашла два эти утверждения и использовала их для логического вывода истинности цели. Это может привести к необходимости выполнить унификацию, чтобы временно конкретизировать переменную X значением `bob`.

Рассмотрим теперь цель

```
man(X).
```

В этом случае система языка Prolog должна сравнить заданную цель с высказываниями, хранящимися в базе данных. Первое обнаруженное высказывание, имеющее форму указанной цели с любым объектом в качестве параметра, приведет к конкретизации переменной *X* значением этого объекта. Если же в базе данных нет высказываний, имеющих форму указанной цели, то система отмечает, что цель не может быть достигнута, отвечая «No».

Существуют два противоположных подхода к сравнению заданной цели и факта в базе данных. Система может начать поиск с фактов и правил, хранящихся в базе данных, и попытаться найти последовательность совпадений, ведущую к цели. Этот подход называется *резолюцией снизу вверх*, или *прямым выводом* (*forward chaining*). Альтернативный подход заключается в том, что система, наоборот, начинает поиск с цели и пытается найти последовательность соответствующих высказываний, ведущую к некоторому множеству исходных фактов, хранящихся в базе данных. Этот подход называется *резолюцией сверху вниз*, или *обратным выводом* (*backward chaining*). Обратный вывод хорошо работает, когда существует небольшой набор возможных ответов. Прямой же вывод работает лучше, когда количество возможных правильных ответов велико; в этой ситуации при обратном выводе для получения ответа может потребоваться очень большое количество сопоставлений. Существующие реализации языка Prolog используют для резолюции обратный вывод, предположительно потому, что их разработчики думали, будто обратный вывод подходит для более широкого класса задач, чем прямой вывод.

Снова рассмотрим тот же пример запроса:

```
man(bob).
```

Предположим, что база данных содержит следующий факт и правило вывода:

```
father(bob).
man(X) :- father(X).
```

При прямом выводе система должна отыскать первое высказывание. Тогда логический вывод цели осуществляется следующим образом: переменная *X* конкретизируется значением `bob`, первое высказывание сопоставляется с правой частью второго правила `father(X)`, а затем левая часть второго высказывания сопоставляется с целью. При обратном же выводе следует сначала сопоставить цель с левой частью второго высказывания `man(X)`, конкретизировав переменную *X* значением `bob`. Тогда на последнем этапе система должна сопоставить правую часть второго высказывания `father(bob)` с первым высказыванием.

Еще одна сложность, относящаяся к разработке языка, возникает всякий раз, когда цель имеет несколько структур (как в примере, приведенном выше). В этом случае вопрос заключается в том, как именно надо выполнять поиск, — сначала в глубину или в ширину? При *поиске сначала вглубь* (*depth-first search*) система находит полную цепочку высказываний (доказательство) для первой подцели раньше, чем приступить к работе над остальными подцелями. При *поиске сначала вширь* (*breadth-first search*) система работает над всеми подцелями параллельно. Разработчики языка Prolog выбрали в качестве основного способа поиск сначала вглубь, поскольку он требует меньше компьютерных ресурсов, тогда как поиск сначала вширь требует реализации параллельных вычислений и может потребовать наличия большого объема памяти.

Последнее свойство механизма резолюции в языке Prolog, которое нам следует обсудить, — *бэктрекинг* (*backtracking*). При обработке цели с несколькими подцелями, если система не способна доказать истинность одной из подцелей, то она отказывается от обработки этой подцели, которую не способна доказать. Вместо этого система заново рассматривает предыдущую подцель (если она является единственной) и пытается найти ее альтернативное решение. Это восстановление предшествующего состояния цели для пересмотра ранее доказанной подцели и называют «бэктрекингом» («откатом»). Новое решение отыскивается в результате поиска, предпринятого с того места, где остановился предыдущий поиск для этой подцели.

Множественность решений для подцели является результатом наличия различных конкретизаций ее переменных. Следует заметить, что бэктрекинг требует больших затрат времени и объема памяти, поскольку он может найти все возможные решения для каждой подцели. К тому же эти доказательства подцелей могут оказаться недостаточно организованными, чтобы минимизировать объем времени, которое требуется для поиска окончательного решения, что еще больше обостряет проблему.

Чтобы укрепить наше понимание бэктрекинга, рассмотрим следующий пример. Пусть в базе данных есть совокупность фактов и правил, а в системе языка Prolog представлена следующая составная цель:

```
male(X), parent(X, "Shelley").
```

В этой цели спрашивается, существует ли какая-либо конкретизация переменной *X*, определяющая, что *X* является мужчиной (`male`) и родителем (`parent`) некоего Шелли (`Shelley`). Система языка Prolog сначала должна найти в базе данных первый факт с функтором `male`. Затем она конкретизирует переменную *X* параметром найденного факта (скажем, параметром `Mike`) и далее пытается доказать, что высказывание `parent("Mike","Shelley")` является истинным. Если же это не удается сделать, то она возвращается к первой подцели `male(X)` и пробует снова удовлетворить ее с помощью некоторой альтернативной конкретизации переменной *X*. Может оказаться, что, выполняя процесс резолюции, система должна будет просмотреть каждого мужчину в базе данных, прежде чем она найдет одного из них, являющегося родителем Шелли. Более того, система *обязательно должна* просмотреть в базе *всех* мужчин, чтобы доказать, что цель не может быть удовлетворена. Заметим, однако, что наш пример цели можно решить более эффективно, если порядок следования двух подцелей поменять местами: только после того, как система с помощью резолюции найдет родителя Шелли, пытаться найти лицо с подцелью `male`. Этот способ эффективен, если у Шелли меньше родителей, чем мужчин в базе данных, — что, конечно же, вполне правдоподобно.

Заметим также, что система языка Prolog всегда выполняет поиск в базе данных в направлении от первого элемента к последнему.

В следующих разделах описываются примеры на языке Prolog, иллюстрирующие процесс резолюции.

2.4. Структура программы на языке Prolog

Программа, написанная на языке Prolog, состоит из следующих разделов: описание доменов, база данных, описание предикатов, описание цели и описание утверждений [4]. Начала этих разделов отмечают ключевые слова `domains`, `database`, `predicates`, `goal` и `clauses`.

Назначение указанных разделов:

- раздел `domains` содержит определения доменов, которые описывают различные классы объектов, используемых в программе;
- раздел `database` содержит утверждения базы данных, которые являются предикатами динамической базы данных; если программа не требует наличия такой базы данных, то этот раздел может быть опущен (возможности использования динамической базы данных будут рассмотрены позже);
- раздел `predicates` служит для описания используемых программой предикатов;
- в разделе `goal` формулируется назначение создаваемой программы; составными частями при этом могут являться некие подцели, из которых формируется единая цель программы;
- в раздел `clauses` заносятся факты и правила, известные априорно; о содержимом этого раздела можно говорить как о данных, необходимых для работы программы.

Заметим, что большинство программ не обязательно содержит все пять названных разделов.

Язык Prolog также обеспечивает возможность включения в программу *комментариев*, которые обрамляются символами `/*` и `*/`.

Приведем пример описания доменов и предикатов. Рассмотрим отношение «любит» между двумя объектами (пусть это будут имя человека и название вещи, которую любит этот человек).

Ранее уже приводилось несколько примеров использования предиката `likes`, например, `likes("Mary",apples)`. Здесь `likes` является предикатом (термом предиката), а `Mary` и `apples` — объектами этого предиката.

Prolog требует указания типов объектов для каждого предиката программы. Некоторые из этих объектов могут быть, к примеру, числовыми данными, а другие — символьными строками. Поэтому в разделе `predicates` необходимо задать тип объектов каждого из предикатов:

```
predicates
  likes (symbol, symbol)
```

Это описание означает, что оба объекта предиката `likes` относятся к типу `symbol`, который является одним из базисных типов в Prolog (другие базисные типы будут рассмотрены ниже).

В некоторых случаях, однако, представляется необходимым иметь возможность несколько большей конкретизации типа используемого предикатом объекта. Например, в предикате `likes` объекты имеют смысл «тот, кто любит» и «вещь, которую любят». Для этого язык Prolog позволяет конструировать свои собственные типы объектов из базисных типов доменов. Например, в разделе программы `domains` могут появиться такие описания:

```
domains
  person, thing = symbol
predicates
  likes (person, thing)
```

Имена person и thing при этом будут обозначать некие совокупности (домены) значений. Рассмотрим, например, следующие три утверждения:

```
likes("John",camera).
likes("Tom",computer).
likes("Kathy",computer).
```

Термы John, Tom и Kathy принадлежат здесь к домену person, а термы camera и computer — к домену thing. Все три эти утверждения восходят к одному и тому же предикату — likes, отличие же состоит лишь в значениях, которые принимают объекты. Другими словами, все три утверждения являются вариациями одного и того же предиката.

Вернемся к описанию доменов. Prolog имеет несколько встроенных типов доменов: символы (все возможные отдельные символы), целые числа, действительные числа, строки (последовательности символов), символические имена (последовательности букв, цифр и знаков подчеркивания, где первый символ — строчная буква, либо последовательности любых символов, заключенных в кавычки) и файлы. Тип каждого из доменов должен быть объявлен в разделе программы domains.

Ранее было отмечено, что в раздел clauses заносятся факты и правила, причем каждый факт завершается символом «точка».

Примеры:

```
clauses
  likes("John",camera).
  likes("Tom",computer).
  likes("Kathy",computer).
```

Рассмотрим особенности использования целей. Далеко не каждая из программ на языке Prolog содержит внутри себя описание своей цели (*внутреннюю цель*), — часто цель задается в процессе работы программы, т. е. является *внешней* [4].

Внутренние цели — это цели поиска, которые задаются в самой программе. Цель при этом может состоять из нескольких подцелей, разделенных запятой (логический союз «И») или точкой с запятой (логический союз «ИЛИ»). Завершается цель точкой. Для вывода на экран во внутренних целях используется встроенный предикат `write`, аргументами которого могут быть строковые константы (заключенные в кавычки) и имена переменных; такие аргументы можно произвольно комбинировать.

Примеры:

```
write ("Любимые вещи:"),
write (X), nl,
write (Y, "любит", Z)
```

Встроенная операция `nl` при этом позволяет осуществлять перевод курсора на новую строку.

Внешние цели. В формулировке запроса к программе можно использовать несколько переменных. Тогда программа выдаст все возможные комбинации значений переменных (предикат `write` для этого использовать не надо).

Пример:

```
/* Демонстрация структуры программы */
domains
  person, thing = symbol
predicates
  likes (person, thing)
clauses
  likes ("John",camera).
  likes ("Tom",computer).
  likes ("Kathy",computer).
goal
  write ("Что же любит Tom?"), nl,
  likes ("Tom", X),
  write ("Tom любит", X).
```

Запросы строятся из предикатов, содержащих условия, которые ограничивают пути поиска желаемых результатов, причем в случае, когда какой-либо запрос нужно повторить несколько раз, разумно предусмотреть возможность не задавать всякий раз одни и те же условия. Полезно также для получения ответов из базы данных не использовать факты из базы данных. В языке Prolog эта задача решается конструированием правил, не содержащих в себе данных, т. е. *правил нулевой арности*.

Пример. Представим некую гипотетическую семью, где Фрэнк и Мэри являются мужем и женой, их сына зовут Сэмом, а дочку — Дебби.

Ниже приведен диалог, касающийся этой семьи.

Вопрос: Кем приходятся друг другу Дебби и Сэм?
Ответ: Дебби — сестра Сэма.
Вопрос: Из чего вы это заключили?
Ответ: У Дебби и Сэма одни и те же родители, Дебби — девочка. Таким образом, Дебби — сестра Сэма.

Второй из этих вопросов является разговорной формулировкой правила, которое будет использоваться для ответа на запрос. Это правило можно перефразировать таким образом:

```
Дебби — сестра Сэма, если
  Дебби — существо женского пола
  И родители Дебби есть родители Сэма.
```

Эта фраза включает в себя *условие «если»* (`if`, может быть обозначено комбинацией символов `:-`), которое логически связывает оба утверждения. Утверждение, предшествующее «если», называется *заключением*, или *логическим следствием*, а утверждение, следующее за «если», — *допущением*, или *предпосылкой*.

Правило, задающее отношение «брат–сестра», тогда выглядит следующим образом:

```
sister (Sister,Brother) if
female (Sister),
  parents (Sister,Father,Mother),
  parents (Brother,Father,Mother).
```

Пример программы:

```
/* Демонстрация конструкции правила */
domains
  person = symbol
predicates
  male(person)
  female(person)
  parents(person,person,person)
  sister(person,person)
  who_is_the_sister
goal
  who_is_the_sister.
clauses
  male("Frank").
  male("Sam").
  female("Mary").
  female("Debbie").
  parents("Sam","Frank","Mary").
  parents("Debbie","Frank","Mary").

who_is_the_sister if
  sister(Sister,Brother),
  write(Sister,"is the sister of",Brother,"."),nl.
sister(Sister,Brother) :-
  female(Sister),
  parents(Sister,Father,Mother),
  parents(Brother,Father,Mother).
```

2.4.1. Использование составных объектов

Рассмотрим факт `student ("Иванов", 14, 05, 1980)`. При таком задании факта оказывается непонятным назначение последних трех объектов. На самом деле они представляют собой дату рождения студента Иванова. Более определенно описать этот объект можно следующим образом: `student ("Иванов", birthday(14, 05, 1980))`. Объект, представляющий собой другой объект или совокупность объектов, называется *составным объектом* [4].

Пример:

```
/* Описание составного объекта */
domains
  date=birthday(integer, integer, integer)
predicates
  student (symbol, date)
clauses
  student ("Иванов",birthday (14, 05, 1980)).
  student ("Петров",birthday (30, 12, 1981)).
  student ("Сидоров",birthday (29, 05, 1981)).
goal
  student ("Иванов",X),
  write("Дата рождения Иванова: ",X).
```

Если необходимо определить только год рождения студента, то цель будет выглядеть так:

```
student ("Иванов", birthday (_,_,Y)),
write("Год рождения Иванова: ",Y).
```

Обратите внимание, что переменная, в определении значения которой нет необходимости, называется *анонимной переменной* и обозначается знаком подчеркивания.

2.4.2. Использование альтернативных доменов

Представление данных часто требует наличия большого числа структур. В Prolog эти структуры должны быть описаны с помощью альтернативного описания доменов.

Пример:

```
/* Использование альтернативных доменов. Для
   разделения альтернативных доменов применена
   точка с запятой (;) */
domains
  thing = misc_thing(whatever);
          book(author,title);
          record(artist,album,type)
  person, whatever, author, title, artist, album,
  type = symbol
```

```
predicates
  owns (person, thing)
clauses
  owns ("Иванов", misc_thing("Piano")).
  owns ("Петров", book("J.R.R. Tolkien",
    "Return of the Ring")).
  owns ("Сидоров", record("Elton John",
    "Ice Fair", "popular")).
```

2.5. Организация повторений в языке Prolog

Очень часто в программах необходимо выполнить одну и ту же задачу несколько раз. Существуют два способа реализации правил, выполняющих одну и ту же задачу многократно. Первый их них мы будем называть *повторением*, а второй — *рекурсией*. Правила Turbo-Prolog, выполняющие повторения, используют откат, а правила, выполняющие рекурсию, используют *самовызов*.

Вид правила, выполняющего повторение, следующий:

```
repetitive_rule :- /* правило повторения */
   <предикаты и правила>,
   fail.           /* неудача */
```

Конструкция *<предикаты и правила>* в теле этого правила обозначает предикаты, содержащие несколько утверждений, а также правила, определенные в программе. Встроенный предикат `fail` («неудача») вызывает откат, так что предикаты и правила выполняются еще раз.

Вид правила, выполняющего рекурсию, следующий:

```
recursive_rule :-   /* правило рекурсии */
   <предикаты и правила>,
   recursive_rule.
```

Заметим, что последним в теле данного правила является само же правило `recursive_rule`. Это и есть рекурсия: тело правила содержит вызов самого себя.

Правила повтора и рекурсии могут обеспечивать одинаковый результат, хотя алгоритмы их выполнения не одинаковы. Каждый из них в конкретной ситуации имеет свои преимущества. Рекурсия, например, может требовать больше системных ресурсов, поскольку всякий раз при рекурсивном вызове новые копии используемых значений помещаются в *стек* — особую область памяти, используемую в основном для передачи значений между правилами. Эти значения сохраняются, пока правило не завершится успешно или неуспешно. В некоторых ситуациях такое использование стека может быть оправдано, если промежуточные значения должны храниться в определенном порядке для дальнейшего использования.

2.5.1. Метод отката после неудачи

Рассмотрим, как *метод отката после неудачи* (*ОПН*) может быть использован для управления вычислением внутренней цели при поиске всех возможных ее решений. Метод ОПН использует предикат `fail`, который вызывает неуспешное завершение правила. При его срабатывании внутренние унификационные подпрограммы выполняют откат в точку возврата, и процесс повторяется до тех пор, пока последнее утверждение не будет обработано.

Использование метода ОПН позволяет извлекать данные из каждого утверждения базы данных. Добавив же дополнительные условия на значения объектов для одной или более переменных предиката, можно извлекать данные только из определенных утверждений.

Пример 1. Пусть в базе данных в виде набора фактов хранятся сведения о нескольких городах (название города и название реки, протекающей через город). Цель — получить на экране полный список городов.

```
/* Демонстрация метода отката после неудачи */
predicates
  city (symbol, symbol)
  all_cities
```

```
clauses
  city ("Самара","Волга").
  city ("Саратов","Волга").
  city ("Ростов","Дон").
  city ("Москва","Москва").
  city ("Волгоград","Волга").
/*правило для получения полного списка городов*/
  all-cities:- city (C,_),
               write (C), nl,
               fail.
goal
  write ("Полный список городов:"),nl,
all_cities.
```

Пример 2. Пусть в базе данных в виде набора фактов хранятся сведения о нескольких городах (название города и название реки, протекающей через город). Цель — получить на экране список городов, стоящих на указанной реке.

```
predicates
  city (symbol, symbol)
  search (symbol)
clauses
  city ("Самара","Волга").
  city ("Саратов","Волга").
  city ("Ростов","Дон").
   city ("Москва","Москва").
  city ("Волгоград","Волга").
  /*правило для поиска*/
  search (R):- city (C,R),
               write (C), nl,
               fail.
goal
  write ("Укажите название реки: "),
  readln (River), nl,
 write ("Города, стоящие на реке", River,":"), nl,
  search (River).
```

2.5.2. Метод отсечения и отката

В некоторых ситуациях необходимо иметь доступ только к определенной части данных. *Метод отсечения и отката* (*OO*) может быть использован для фильтрации данных, выбираемых из утверждений базы данных. Задавая условие на окончание просмотра базы данных, можно получить только требуемую часть информации.

Для этих целей Prolog имеет встроенный предикат `cut` («отсечение»), который обозначается символом восклицательного знака (`!`). Этот предикат, вычисление которого всегда завершается успешно, заставляет внутренние унификационные подпрограммы «забыть» все указатели отката, установленные во время попыток вычислить текущую подцель. Другими словами, предикат `cut` «устанавливает барьер», запрещающий выполнить откат ко всем альтернативным решениям текущей подцели. Однако последующие подцели могут создать новые указатели отката и тем самым создать условия для поиска новых решений, на которые область действия предиката `cut` уже не распространяется. Но если все более поздние цели окажутся неуспешными, то барьер, установленный предикатом `cut`, заставит механизм отката отсечь все решения в области действия `cut` путем немедленного отката к другим возможным решениям вне области действия предиката `cut`. Тем самым метод отсечения и отката имитирует неуспешное вычисление и выполняет последующий откат до тех пор, пока не будет обнаружено определенное условие, а предикат `cut` служит для устранения всех последующих откатов.

Пример. Пусть имеется база данных, которая содержит несколько имен детей. Цель — выдать список этих имен до имени `Diana` включительно.

```
/* Демонстрация метода отката и отсечения */
predicates
  name (symbol)
  choice
clauses
  name ("Mary").
  name ("Bob").
```

```
  name ("Diana").
  name ("John").
  name ("Peter").
  /*правило для поиска*/
  choice:- name (N),
           write (N), nl,
           N="Diana",!.
goal
  write ("Список имен до Diana: "), nl,
  choice.
```

2.5.3. Простая рекурсия

Правило, содержащее само себя в качестве компонента, называется *правилом рекурсии*.

Пример. Программа «Эхо» — ввод с клавиатуры строк и вывод их на экран; признак окончания ввода данных — пустая строка.

```
/* Демонстрация рекурсии на примере
   программы "Эхо" */
predicates
  echo
clauses
  echo :- readln (String),
          String<>"",
          write (String), nl,
          echo.
goal
  echo.
```

2.5.4. Метод обобщенного правила рекурсии (ОПР)

Обобщенное правило рекурсии содержит в теле правила само себя. Рекурсия будет конечной, если в правило включено условие выхода, гарантирующее окончание его работы. Тело правила состоит из утверждений и правил, определяющих

задачи, которые должны быть выполнены. Общий вид правила рекурсии в символическом виде [4]:

```
<имя правила рекурсии> :-
  <список предикатов>,          (1)
  <предикат условия выхода>,    (2)
  <список предикатов>,          (3)
  <имя правила рекурсии>,       (4)
  <список предикатов>.          (5)
```

Данное правило рекурсии имеет пять компонентов. Первый — это группа предикатов, которые всегда истинны и на рекурсию не влияют. Второй компонент — это предикат условия выхода. Успех этого предиката позволяет продолжить рекурсию, а неудача вызывает ее остановку (рекурсивное правило возвращает «Ложь»). Третий компонент — список других предикатов, которые также всегда истинны и на рекурсию тоже не оказывает влияния. Четвертая группа — само рекурсивное правило, успех которого и вызывает рекурсию. Наконец, пятая группа — это список предикатов, которые тоже всегда успешны и опять-таки не влияют на рекурсию. Пятая группа также получает значения (если они имеются), помещенные в стек во время выполнения рекурсии.

Напомним, что правило рекурсии обязательно должно содержать условие выхода. В противном случае рекурсия будет бесконечной, а такое правило — бесполезным. Ответственность за обеспечение завершаемости правила рекурсии возлагается на программиста. Правила, построенные указанным образом, являются *обобщенными правилами рекурсии* (*ОПР*), а сам метод называется *ОПР-методом*.

Пример. Правило генерации всех целых чисел, начиная с 1 и заканчивая 7.

Пусть имя правила будет `write_number(Number)`.

Для этого примера первый компонент структуры общего правила рекурсии не используется. Вторым компонентом, т. е. предикатом выхода, является условие `Number < 8`: когда значение `Number` равно 8, правило будет успешным, а программа завершится. Третий компонент правила оперирует с числами: число выдается на экран, а затем увеличивается на 1. Для

увеличенного числа будет использоваться новая переменная Next_Number. Четвертый компонент — вызов самого правила рекурсии write_number(Next_number). Пятый компонент, представленный в общем случае, здесь не используется.

Таким образом, программа генерации ряда чисел будет использовать следующее правило рекурсии:

```
write_number(8).
write_number(Number) :-
  Number < 8,
  write(Number), nl,
  Next_Number = Number + 1,
  write_number(Next_number).
```

Рассмотрим ход выполнения этой программы. Она начинается с попытки вычислить подцель write_number(1). Сначала программа сопоставляет подцель с первым правилом write_number(8). Так как 1 не равно 8, то это сопоставление неуспешно. Тогда программа вновь пытается сопоставить подцель, но уже с головой правила write_number (Number). На этот раз сопоставление успешно, так как переменной Number присвоено значение 1. Теперь программа сравнивает это значение с 8 (условие выхода). Так как 1 меньше 8, то данное подправило успешно. Следующий предикат выдает значение, присвоенное Number. Переменная Next_Number получает значение 2, а значение Number увеличивается на 1.

2.6. Списки в языке Prolog

В разделе 2.4 мы познакомились с основами представления данных в языке Prolog. Это прежде всего утверждения, объектами которых являются конкретные значения (данные). Однако Prolog также поддерживает связанные объекты, называемые *списками*. **Список** — это упорядоченный набор объектов, следующих друг за другом. Составляющие списка внутренне связаны между собой, поэтому с ними можно работать и как с группой (списком в целом), и как с индивидуальными объектами (элементами списка).

Рассмотрим структуру, организацию и представление списков.

Список является набором объектов одного и того же доменного типа. Объектами списка могут быть целые числа, действительные числа, символы, символьные строки и структуры. Порядок расположения элементов является отличительной чертой списка; те же самые элементы, упорядоченные иным способом, представляют уже совсем другой список, поскольку порядок играет важную роль в процессе сопоставления.

Prolog допускает списки, элементами которых являются структуры. Если структуры принадлежат к альтернативному домену, то элементы списка могут иметь разный тип. Такие списки используются для специальных целей.

Совокупность элементов списка заключается в квадратные скобки — `[]`, а друг от друга элементы списка отделяются запятыми.

Примеры:

```
[1,2,3,6,9,3,4]
[3.2,4.6,1.1,2.64,100.2]
["YESTERDAY","TODAY","TOMORROW"]
```

Элементами первого списка являются целые числа, элементами второго — действительные числа, а третьего — символьные строки.

Количество элементов в списке называется его *длиной*. Например, длина списка `["MADONNA","AND","CHILD"]` равна 3, а длина списка `[4.50,3.50,6.25,2.9,100.15]` равна 5. Список может также содержать всего один элемент или даже не содержать элементов вовсе, например `["summer"]` или `[]`. Список, не содержащий элементов, называется *пустым*, или *нулевым списком*.

Непустой список можно рассматривать как состоящий из двух частей: первый элемент списка — его *голова* (1), а остальная часть списка — *хвост* (2). Голова является одним из элементов списка, а хвост — это список сам по себе. Голова списка представляет собой отдельное неделимое значение; хвост же — это список, составленный из того, что осталось от исходного списка в результате «усекновения головы». Если, например, список состоит из одного-единственного элемента, то его можно разделить на голову, которой будет этот самый единственный элемент, и хвост, являющийся пустым списком.

Примеры:

Список	Голова	Хвост
[1,2,3,4,5]	1	[2,3,4,5]
['s','k','y']	's'	['k','y']
[cat]	cat	[]
[]	*не определена*	*не определен*

Чтобы использовать в программе список, необходимо описать *предикат списка*.

Примеры:

```
num ([1,2,3,6,9,3,4])
realnum ([3.2,4.6,1.1,2.64,100.2])
time (["YESTERDAY","TODAY","TOMORROW"])
```

Введение списков в программу необходимо отразить в трех ее разделах. Домен списка должен быть описан в разделе `domains`, работающий со списком предикат — в разделе `predicates` и, наконец, надо задать сам список в разделе `clauses` или `goal`.

Пример:

```
/* Демонстрация работы со списками */
domains
   bird_list = bird_name *
   bird_name = symbol
   number_list = number *
   number = integer
predicates
   birds(bird_list)
   score(number_list)
clauses
   birds(["sparrow", "robin", "mockingbird",
          "thunderbird"]).
   score([56,87,63,89,91,62,85]).
```

Отличительной особенностью описания списка является наличие звездочки (*) после имени домена элементов. Так, запись `bird_name *` указывает, что это домен списка, элементами которого являются `bird_name`, т. е. запись `bird_name *` следует понимать как список, состоящий из элементов домена `bird_name`.

Примеры внешних целей:

```
birds(All).
birds([_,_,_,B,_]).
birds([B1,B2,_,_,_]).
score(All).
score([F,S,T,_,_,_,_]).
```

При задании целей во втором, третьем и пятом примерах требовалось точное знание количества элементов списка, являющегося объектом предиката `birds`, что не всегда возможно (с точки зрения пользователя). Но вспомним, что Prolog позволяет отделять от списка первый элемент и обрабатывать его отдельно. Данный метод работает вне зависимости от длины списка до тех пор, пока список не будет исчерпан. Этот метод доступа к голове списка называется *методом разделения списка на голову и хвост*.

Операция деления списка на голову и хвост обозначается при помощи вертикальной черты (|):

```
[Head|Tail].
```

`Head` здесь является переменной для обозначения головы списка, переменная `Tail` обозначает хвост списка.

Пример. Правило печати элементов списка `[4,-9,5,3]`:

```
print_list ([]).
  print_list ([Head|Tail]) :-
    write (Head),nl,
    print_list (Tail).
```

Когда это правило пытается удовлетворить цель `print_list([4,-9,5,3])`, то первый вариант правила — `print_list[]` — дает неуспех, так как его объект является пустым списком. Напротив, введенный список соответствует объекту второго варианта предиката — `print_list([Head|Tail])`.

Переменной Head, следовательно, присваивается значение первого элемента в списке: 4, в то время как переменной Tail ставится в соответствие оставшаяся часть списка: [-9,5,3]. Теперь, когда выделен первый элемент списка, с ним можно обращаться так же, как и с любым простым объектом: write(Head),nl. Далее, так как хвост списка есть список сам по себе, то значение переменной Tail может быть использовано в качестве объекта рекурсивного вызова print_list: print_list(Tail). Когда испытывается данное подправило, Tail имеет значение [-9,5,3]. Снова первый вариант не проходит и соответствие устанавливается при помощи второго. Переменной Head присваивается значение −9, которое затем печатается на экране, а процесс повторяется со списком [5,3]. В итоге, когда переменная Head принимает значение 3, переменной Tail присваивается пустой список. Теперь при рекурсивном вызове print_list(Tail) значение Tail соответствует объекту правила print_list([]). Поскольку этот вариант не имеет рекурсивных вызовов, цель считается удовлетворенной, и вырабатывается условие окончания рекурсии print_list. Тем самым первый вариант позволяет print_list завершиться успехом, когда рекурсивные вызовы опустошат весь список.

Пример:

```
/* Демонстрация использования метода деления списка
на голову и хвост при выводе списка на экран */
domains
   animal_list = symbol *
predicates
   print_list(animal_list)
clauses
   animal (["тигр","заяц","лев","волк"]).
   print_list([]).
   print_list([Head|Tail]) :-
      write(Head),nl,
      print_list(Tail).
goal
   animal (Animals_list),
   print_list (Animals_list).
```

2.6.1. Операции над списками

Поиск элемента в списке представляет собой просмотр списка для выявления соответствия между элементом данных и элементом просматриваемого списка. Если такое соответствие найдено, то поиск заканчивается успехом; в противном случае поиск заканчивается неуспехом. Для сопоставления объекта поиска с элементами просматриваемого списка необходим предикат, объектами которого являются эти объект поиска и список:

```
find_it(3 ,[1,2,3,4,5]).
```

Первый из объектов утверждения — 3 — есть объект поиска. Второй — это список `[1,2,3,4,5]`. Для выделения элемента из списка и его сравнения с объектом поиска можно применить метод разделения списка на голову и хвост. Стратегия поиска при этом будет состоять в рекурсивном выделении головы списка и ее сравнении с элементом поиска.

Правило поиска может сравнивать объект поиска и голову текущего списка. Саму операцию сравнения можно записать в виде:

```
find_it(Head,[Head|_]).
```

Этот вариант правила предполагает наличие соответствия между объектом поиска и головой списка. В данном случае поскольку осуществляется попытка найти соответствие между объектом поиска и головой списка, то нет необходимости заботиться о том, что происходит с хвостом. Если объект поиска и голова списка действительно соответствуют друг другу, то результатом сравнения явится успех; если же нет, то неуспех. Но если эти два элемента данных различны, то попытка сопоставления считается неуспешной, происходит откат и поиск другого правила или факта, с которыми можно снова попытаться найти соответствие. Для случая несовпадения объекта поиска и головы списка необходимо предусмотреть правило, которое выделяло бы из списка следующий по порядку элемент и делало бы его доступным для сравнения:

```
find_it(Head, [Head|_].
find_it(Head, [_,Tail]) :-
   find_it(Head, Tail).
```

Если правило `find_it(Head,[Head|_])` неуспешно, то происходит откат и делается попытка со вторым вариантом. При этом опять-таки присвоенный переменной `Tail` список разделяется на голову и хвост при помощи утверждения `find_it(Head,[Head|_])`, и этот процесс повторяется, пока данное утверждение не даст либо успех (в случае установления соответствия на очередной рекурсии), либо неуспех (в случае исчерпания списка).

Деление списков. При работе со списками достаточно часто требуется разделить список на несколько частей. Это бывает необходимо, когда для текущей обработки нужна лишь определенная часть исходного списка.

Рассмотрим предикат `split`, аргументами которого являются элемент данных и три списка:

```
split(Middle,L,L1,L2).
```

Элемент `Middle` здесь является *компаратором*, `L` — это исходный список, а `L1` и `L2` — подсписки, получающиеся в результате деления списка `L`. Если элемент исходного списка меньше или равен `Middle`, то он помещается в список `L1`, а если больше, — то в список `L2`.

Пример:

```
split(40,[30,50,20,25,65,95],L1,L2).
```

Правило здесь устроено следующим образом: очередной элемент извлекается из списка при помощи метода разделения списка на голову и хвост, а потом сравнивается с компаратором `Middle`. Если значение этого элемента меньше или равно значению компаратора, то элемент помещается в список `L1`, в противном случае — в список `L2`. В результате применения этого правила к списку `[30,50,20,25,65,95]` значениями списков `L1` и `L2` станут соответственно `[30,20,25]` и `[50,65,95]`. Само же это правило для разделения списка записывается на языке Prolog следующим образом:

```
split(Middle,[Head|Tail],[Head|L1],L2) :-
  Head <= Middle,
  split(Middle,Tail,L1,L2).
split(Middle,[Head|Tail],L1,[Head|L2]) :-
  Head > Middle,
  split(Middle,Tail,L1,L2),
split(_,[],[],[]).
```

Отметим, что метод деления списка на голову и хвост используется в данном правиле как для разделения исходного списка, так и для формирования выходных списков.

Присоединение списка. Слияние двух списков и получение, таким образом, третьего списка принадлежит к числу наиболее полезных при работе со списками операций.

В качестве примера рассмотрим две переменные `L1` и `L2`, представляющие списки и имеющие значения `[1,2,3]` и `[4,5]` соответственно. Тогда весь процесс слияния можно представить в виде такой совокупности действий:

1) список `L3` (результирующий) вначале пуст;
2) элементы списка `L2` пересылаются в `L3`; теперь значением `L3` будет `[4,5]`;
3) элементы списка `L1` пересылаются в `L3`; в результате он принимает значение `[1,2,3,4,5]`.

Структура правила для выполнения этих действий следующая:

```
append([],L,L).
append([N|L1],L2,[N|L3]) :-
   append(L1,L2,L3).
```

Если на его вход подать списки `L1=[1,2,3]` и `L2=[4,5]`, то сначала Prolog пытается удовлетворить первый вариант правила:

```
append([],L,L).
```

Чтобы сделать это, первый объект предиката должен быть пустым списком. Однако это не так. Внутренний процесс унификации Prolog, пытаясь удовлетворить второе правило `append`, раскручивает цепочку рекурсий до тех пор, пока не обнулит первый список. Элементы списка при этом последовательно пересылаются в стек. Когда первый объект предиката `append` окажется пустым списком, становится возможным применение первого варианта правила. Третий список при этом инициализируется вторым:

```
append([],[4,5],_).
append([],[4,5],[4,5]).
```

Теперь Prolog начинает сворачивать рекурсивные вызовы второго правила. Извлекаемые при этом из стека элементы помещаются один за другим в качестве головы к первому и третьему спискам. Следует особо отметить, что элементы извлекаются в обратном порядке (ведь это стек!), и что значение извлеченного из стека элемента присваивается переменной `N` одновременно в `[N|L1]` и `[N|L3]`.

Шаги данного процесса можно представить так:

```
append([],[4,5],[4,5])
append([3],[4,5],[3,4,5])
append([2,3],[4,5],[2,3,4,5])
append([1,2,3],[4,5],[1,2,3,4,5])
```

Сортировка списков представляет собой переупорядочивание элементов списка определенным образом для упрощения доступа к нужным элементам. Существует много способов сортировки списков, но обычно применяются три из них: *метод перестановки*, *метод вставки* и *метод выборки* (или их комбинации). Первый из этих методов заключается в попарной перестановке элементов списка до тех пор, пока они не выстроятся в нужном порядке. Второй осуществляется при помощи вставки элементов в список на требуемые места до тех пор, пока список не будет упорядочен. Третий метод включает в себя многократную выборку и перемещение элементов списка. Второй из указанных методов (метод вставки) особенно удобен для реализации на языке Prolog.

Рассмотрим список `[4,7,3,9]`, элементы которого расположены хаотично. Пусть мы хотим получить из него список `[3,4,7,9]`, упорядоченный по возрастанию. Опишем предикат, производящий нужную сортировку списка методом вставки.

Чтобы воспользоваться преимуществами мощного средства языка Prolog — *внутренней унификацией*, проведем сортировку хвоста. Правила, реализующие этот способ сортировки, имеют следующую структуру:

```
insert_sort([],[]).
insert_sort([X|Tail],Sorted_list) :-
   insert_sort(Tail,Sorted_Tail),
   insert(X,Sorted_Tail,Sorted_list).
```

```
insert(X,[Y|Sorted_list],[Y|Sorted_list1]) :-
   asc_order(X,Y), !,
   insert(X,Sorted_list,Sorted_list1).
insert(X,Sorted_list,[X|Sorted_list]).
asc_order(X,Y) :- X>Y.
```

Обсудим работу этих правил на примере списка `[4,7,3,9]`. Вначале Prolog применяет указанные выше правила к исходному списку; выходной список в этот момент еще не определен:

```
insert_sort([4,7,3,9],_).
```

Первая попытка удовлетворить правило `insert_sort` осуществляется с первым вариантом правила `insert_sort ([],[])`, т. е. для удовлетворения этого правила оба списка должны быть пустыми.

Второй вариант правила `insert_sort` трактует список как комбинацию головы списка и его хвоста. Внутренние унификационные процедуры языка Prolog пытаются сделать пустым исходный список. Устранение элементов списка начинается с головы списка и осуществляется рекурсивно. По мере того как Prolog пытается удовлетворить первое из правил, происходят рекурсивные вызовы `insert_sort`, при этом значениями `X` последовательно становятся все элементы исходного списка, которые затем помещаются в стек. В результате применения этой процедуры список становится пустым. Теперь первый вариант правила `insert_sort` производит обнуление выходного списка. Далее Prolog пытается удовлетворить второе правило из тела `insert_sort` — правило `insert`. Переменной `X` сначала присваивается первое взятое из стека значение 9, а правило `insert` принимает форму `insert(9,[],[9])`. Затем из стека извлекается следующий элемент — 3 — и испытывается первый вариант `insert`, а значит, и правило упорядочивания `asc_order(X,Y) :- X>Y`. Переменная `X` здесь имеет значение 3, а `Y` — значение 9. Так как это правило неуспешно, то вместе с ним неуспешен и первый вариант `insert`. Тогда при помощи второго варианта `insert 3` вставляется в выходной список слева от 9: `insert(3,[9],[3,9])`.

Далее происходит возврат к insert_sort; теперь оно имеет форму insert_sort([3,9],[3,9]). На следующем круге рекурсии происходит вставка очередного элемента из стека, а именно 7. В начале работы на этом круге правило insert имеет вид insert(7,[3,9],_) и происходит сравнение 7 и 3: asc_order(7,3):- 7>3.

Так как данное правило успешно, то элемент 3 убирается в стек, а insert вызывается рекурсивно еще раз, но уже с хвостом списка: insert(7,[9],_).

Так как правило asc_order(7,9):- 7>9 неуспешно, то испытывается второй вариант insert (успешно) и происходит возврат на предыдущие круги рекурсии сначала insert, а потом insert_sort. В результате 7 помещается в выходной список между элементами 3 и 9: insert(7,[3,9],[3,7,9]).

При возврате еще на один круг рекурсии insert_sort из стека извлекается элемент 4. Правило insert теперь выглядит как insert(4,[3,7,9],_). Далее мы получаем правило asc_order(4,3) :- 4>3. Оно успешно, следовательно, имеем insert(4,[7,9],_). Правило же asc_order(4,7):- 4>7 неуспешно. Это означает, что 4 окажется в выходном списке между 3 и 7:

```
insert(4,[3,7,9],[3,4,7,9]).
insert_sort([4,7,3,9],[3,4,7,9]).
```

Теперь в стеке больше нет элементов, а все рекурсии insert_sort уже свернуты.

Компоновка данных в список. Иногда при программировании определенных задач возникает необходимость собрать данные из базы данных в список для последующей их обработки. Prolog содержит встроенный предикат, позволяющий справиться с этой задачей:

```
findall(Variable_name, Predicate_expression,
        List_name).
```

Variable_name здесь обозначает объект входного предиката Predicate_expression, а List_name является именем переменной выходного списка. Эта переменная должна относиться к домену списков, объявленному в разделе domains.

Пример. Опишем предикат, содержащий сведения об очках, набранных футбольными командами. Необходимо сложить все очки и усреднить их.

```
/* Демонстрация предиката компоновки данных
в список для вычисления среднего значения*/
domains
   list = real *
predicates
   football (symbol,real)
   sum_list (list, real, integer).
   average_score
clauses
   /* факты (футбольная база данных) */
   football("Ohio State",116.0).
   football("Michigan",121.0).
   football("Michigan State",114.0).
   football("Purdue",99.0).
   football("UCLA",122.0).
average_score :-
    findall(Points,football(_,Points),Point_list),
    sum_list (Point_list, Sum, Number),
    Average = Sum / Number,
    write("Среднее значение= ",Average).
   sum_list ([],0,0).
   sum_list ([H|T], Sum, Number) :-
    sum_list(T,Sum1,Number1),
    Sum = H + Sum1,
    Number = Number1 + 1.
goal
   average_score.
```

2.7. Строки в языке Prolog

Строка в Prolog, как и в других языках, представляет собой набор символов. При программировании на Prolog символы могут быть «записаны» при помощи их кодов ASCII. Обратный слэш (\), непосредственно за которым следует десятичный код ASCII символа (*N*), интерпретируется как соответ-

ствующий символ. Для представления одиночного символа выражение \N должно быть заключено в апострофы: '\N'. Для представления же строки символов их коды ASCII помещаются друг за другом, а вся строка заключается в двойные кавычки: "\N\N\N". Например, запись «ABC» равносильна записи "\65\66\67".

2.7.1. Операции над строками

Длина строки измеряется полным количеством символов в строке. Prolog имеет встроенный предикат, который используется для нахождения длины строки:

```
srt_len (String_value, String_length).
```

Если в правиле srt_len("TODAY",L) переменная L не означена до начала обработки правила, то она получит значение длины строки "TODAY". Этим значением будет целое число. Если же до начала выполнения вызова предиката str_len обе его переменные уже означены, то предикат будет успешным только при условии, что значение String_length равно длине строки String_value. Например, если S имеет значение "ABC", а L имеет значение 3, то предикат str_len(S,L) успешен; в противном случае он неуспешен.

Конкатенация двух строк означает их объединение, т. е. образование одной новой строки. Например, результат конкатенации двух строк "one day" и "at a time" может быть равен "one day at a time" или "at a time one day". Эти две результирующие строки различны, так как образующие их строки были объединены в различном порядке. Prolog имеет встроенный предикат, который выполняет соединение (конкатенацию) двух строк и имеет синтаксис:

```
concat(Input_string1,Input_string2,Output_String).
```

При этом в нем должно иметься не менее двух любых входных объектов. Этот же предикат, если задана выходная строка, позволяет определить одну из исходных строк, участвующих в конкатенации, либо, если заданы все объекты, проверяет истинность операции.

Пример:

```
concat("TODAY","TOMORROW",S) ⇒ S="TODAYTOMORROW"
concat(S,"TOMORROW","TODAYTOMORROW")⇒ S="TODAY"
concat("TODAY",S,"TODAYTOMORROW")⇒ S="TOMORROW"
concat("TODAY","TOMORROW","TODAYTOMORROW") ⇒ истина
concat("TODAY","TOMORROW","TODAYTOMOR") ⇒ ложь
```

Создание подстрок. Подстрока — это строка, являющаяся копией некоторой части исходной строки. Например, двумя возможными подстроками строки `"Expert Systems"` являются `"Expert"` и `"Systems"`. Prolog имеет встроенный предикат, служащий для создания подстрок, синтаксис которого:

```
fronsrt(Number,Source_string,Substring1,Substring2).
```

Аргумент `Number` задает полное количество символов, которые должны быть скопированы в `Substring1` из исходной строки `Source_String`. Остальные символы строки `Source_string` будут скопированы в `Substring2`.

Пример. Утверждение:

```
frontstr(6,"Expert systems", String1, String2)
```

присваивает `String1` значение `"Expert"`, а `String2` — значение `"systems"`.

Создание символьных префиксов. Создать «префиксный» символ — это значит присоединить этот символ к началу строки. Например, присоединение префикса `'A'` к строке `"BCDEF"` дает строку `"ABCDEF"`. Эта операция в Prolog реализуется с помощью встроенного предиката:

```
frontchar(String,Char,Rest_of_string).
```

Объекту `String` присваивается значение, состоящее из `Char` и `Rest_of_string` («остаток строки»).

Указанный предикат также можно использовать для выделения одного символа (первого) из строки.

Пример:

```
frontchar(Str, 'F', "OX")  ⇒ Str="FOX"
frontchar("SPRING", C, "PRING")  ⇒ C='S'
frontchar("dBASE", 'd', X) ⇒ X="BASE"
```

Специальные строки. В Prolog определены специальные строки, используемые для определенных целей. Эти строки называются *именами* и используются для обозначения символических констант, доменов, предикатов и переменных. По определению, специальные строки в Prolog имеют следующие свойства:

- строка строится из прописных и строчных букв, цифр и символов подчеркивания;
- между символами не должно быть пробелов;
- строка начинается с буквы;
- строка не может начинаться с любого из специальных символов;
- строка не может содержать управляющих символов.

Prolog имеет встроенный предикат, позволяющий проверять, является ли строка специальной строкой Prolog. Синтаксис этого предиката следующий:

```
isname(String).
```

Если `String` — это специальная строка Prolog, то предикат будет успешным; в противном случае он будет неуспешным.

Преобразование данных. Prolog для преобразования данных из одного типа в другой предоставляет следующие предикаты:

```
upper_lower (U, L)
str_char (S, C)
str_int (S, I)
str_real (S, R)
char_int (C,I).
```

Их применение целесообразно, когда тип объектов встроенного предиката отличается от типа объектов предиката, определенного пользователем.

Все предикаты преобразования данных содержат два объекта, а имена этих предикатов указывают тип выполняемого преобразования. Например, `str_char` преобразует строку, состоящую из одного символа и имеющую тип `string`, в значение типа `char`. Имена предикатов также указывают порядок объектов, поскольку имеют два направления преобразования данных. Например, если в предикате `upper_low (S1,S2)` переменная `S1` уже означена и имеет значение `"STARS AND STRIPES"`, то данный предикат присваивает строку `"stars and stripes"` переменной `S2`. Но если `S2` означена, а `S1` — не означена, то значение `"STARS AND STRIPES"` получает переменная `S1`. В случае же, когда обе переменные означены, предикат успешен, если одна из переменных имеет значение строки, содержащееся в другой переменной, но записанной строчными буквами (т. е. фактически выполняется проверка их совпадения без учета регистра).

Предикат `str_char`, как уже было сказано, используется для преобразования объектов типа `string` в объекты типа `char`. Предикат `str_int` используется для преобразования строчного представления целых чисел в переменные типа `integer`. Предикат `str_real` используется для преобразования действительных чисел в строки. Предикат же `char_int` используется для присваивания числа (ASCII-кода символа) данному объекту.

2.8. Файлы в языке Prolog

Работа с данными, содержащимися в файлах, называется *файловой обработкой*. К числу наиболее типичных операций над файлами относятся создание файла, запись в файл и чтение из файла, а также модификация уже существующего файла и добавление к нему новых данных.

2.8.1. Предикаты Prolog для работы с файлами

Перечислим некоторые предикаты Prolog для работы с файлами:

- `deletefile(`*`Filename`*`)` — уничтожение файла;
- `save(`*`Filename`*`)` — сохранение файла на диске;

- renamefile(*OldFilename*, *NewFilename*) — переименование файла;
- existfile(*Filename*) — проверка наличия файла с заданным именем;
- disk(*Path*) — выбор накопителя и пути доступа к файлу;
- dir(*Path*, *Filespec*, *Filename*) — выдача списка файлов в заданном каталоге. В последних двух случаях переменной Path должен быть присвоен корректный путь доступа, а переменная Filespec задает расширение представляющей интерес группы файлов (данный предикат выдает список имен файлов с заданным расширением). Имя выбранного файла будет присвоено переменной Filename;
- openwrite(*FDomain*, *Filename*) — открытие файла для записи. Здесь FDomain — файловый домен, а Filename — имя файла на диске;
- openread(*FDomain*, *Filename*) — открытие файла для чтения;
- openmodify(*FDomain*, *Filename*) — открытие уже существующего файла для его модификации (при этом указатель помещается в начало этого файла);
- openappend(*FDomain*, *Filename*) — открытие уже существующего файла для дозаписи новых данных в его конец;
- writedevice(*FDomain*) — назначение устройства записи;
- readdevice(*FDomain*) — назначение устройства чтения;
- closefile(*FDomain*) — закрытие файла;
- flepos(*Logical_filename*, *File_position*, *Mode*) — помещение указателя в файле в нужную позицию. При этом параметру Mode можно присвоить одно из трех возможных значений: 0 — смещение (File_position) указано относительно начала файла, 1 — смещение указано относительно текущей позиции или 2 — смещение указано относительно конца файла.

2.8.2. Описание файлового домена

Чтобы использовать в программе на языке Prolog файлы, необходимо снабдить ее описанием *файлового домена*. Описание файлового домена с именем `datafile` выглядит следующим образом [4]:

```
domains
  file = datafile
```

В описании `file` можно указать несколько символических имен, но само это описание должно быть единственным. Если в программе вводится несколько символических имен файлов, то они разделяются между собой точкой с запятой:

```
file = datafile1; datafile2; datafile3
```

2.8.3. Запись в файл

Опишем последовательность действий, необходимых для записи информации в файл.

1. **Открытие файла.** Предикат `openwrite(datafile,"FILE1.DAT")`, где `datafile1` — это введенный пользователем файловый домен, а `FILE1.DAT` — имя файла на диске, устанавливает связь между объектами `datafile` и `FILE1.DAT`. После этого ссылки на `datafile1` будут означать обращение к `FILE1.DAT`, и эта связь останется в силе вплоть до закрытия файла. Если файл с именем `FILE1.DAT` к моменту вызова предиката `openwrite` уже присутствовал в каталоге, то его содержимое будет утрачено. Чтобы застраховаться от этого, можно сначала проверить наличие файла при помощи предиката `existfile("FILE1.DAT")` и принять соответствующие меры, если предикат `existfile` будет успешным.
2. **Назначение файла для записи.** Эту операцию выполняет предикат `writedevice(datafile)`. Возможно также указание вместо файла одного из устройств для записи (вывода информации): экрана (`screen`) или принтера (`printer`).

3. **Собственно запись в файл.** Можно использовать любые подходящие для этой цели предикаты, например, `write`: любой предикат `write` после выполнения `writedevice` будет выводить информацию не на экран, а в заданный файл.
4. Использование любых других предикатов и правил, отвечающих назначению программы.
5. **Закрытие файла.** Для этого служит предикат `closefile (datafile)`. Когда файл закрыт, операции чтения или записи для него уже недопустимы. Закрытие файла также защищает его содержимое от каких-либо манипуляций. Еще одним следствием закрытия файла является перевод указателя файла в его начало, — это может понадобиться при повторном открытии файла.

Пример фрагмента программы, поясняющий сказанное:

```
openwrite(datafile,"FILE1.DAT"),
writedevice(datafile),
< любые правила или предикаты записи в файл >
< любые другие правила или предикаты >
closefile(datafile).
```

2.8.4. Чтение из файла

Для чтения данных из файла требуется выполнить следующие действия.

1. **Открытие файла** — `openread(datafile,"FILE1.DAT").`
2. **Назначение файла для чтения** — `readdevice(datafile).` Можно также назначить устройство чтения (ввода данных) — клавиатуру (`keyboard`).
3. Собственно чтение из файла при помощи соотвествующего предиката или правила.
4. Использование произвольных предикатов и правил, отвечающих целям программы.
5. **Закрытие файла** — `closefile(datafile).`

Пример фрагмента, демонстрирующего сказанное:

```
openread(datafile,"FILE1.DAT"),
readdevice(datafile),
< любые правила или предикаты чтения из файла >
< любые другие правила или предикаты >
closefile(datafile).
```

2.8.5. Модификация существующего файла

Последовательность действий, требуемых для модификации уже существующего файла, несколько отличается от той, которая необходима для записи в файл или чтения из него. Прежде всего файл должен быть *открыт для модификации* (т. е. и для чтения, и для записи одновременно). Для этого служит предикат `openmodify(datafile,"FILE1.DAT")`, который успешен, только если файл уже имеется на диске. Когда файл открывается для модификации, указатель помещается в его начало.

Для модификации файла нужно выполнить следующие действия.

1. **Открытие файла** — `openmodify(datafile,"FILE1.DAT").`
2. **Переадресация вывода в файл** — `writedevice(datafile).`
3. **Запись в файл** новых данных.
4. Использование произвольных предикатов и правил, отвечающих целям программы.
5. **Закрытие файла** — `closefile(datafile).`

Пример:

```
openmodify(datafile,"FILE1.DAT"),
writedevice(datafile),
< правила для выборочной записи в файл >,
< любые другие правила или предикаты >
closefile(datafile).
```

2.8.6. Дозапись в конец уже существующего файла

Возможность записывать новые данные в конец уже существующего файла обеспечивается в Prolog предикатом `openappend`. Когда файл открывается для дозаписи, указатель файла автоматически помещается в его конец.

Для добавления новых данных в конец файла нужно выполнить следующие действия.

1. **Открытие файла** — `openappend(datafile,"FILE1.DAT").`
2. **Переадресация вывода в файл** — `writedevice(datafile).`
3. **Дозапись в файл** новых данных при помощи соответствующих правил.

4. **Использование произвольных предикатов и правил, отвечающих целям программы.**
5. **Закрытие файла** — `closefile(datafile).`

Пример:

```
openappend(datafile,"FILE1.DAT"),
writedevice(datafile),
< любые правила для дозаписи в файл >,
< любые другие правила или предикаты >
closefile(datafile).
```

Пример записи в файл

```
/* Вывод информации из статической базы на экран
   дисплея и в файл на диске. */
domains
  str = string
  file = datafile
predicates
  data(str)
  write_lines
goal
  openwrite(datafile,"SHAKE1.DAT"),
  write_lines,
  closefile(datafile).
clauses
  data("A drum, a drum!").
  data("Macbeth does come").
  data("The weird sisters, hand in hand,").
  data("Posters of the sea and land,").
  data("Thus do go about, about:").
  data("Thrice to thine and thrice to mine.").
  data("And thrice again, to make up nine.").
  write_lines :-
    data(Line),
    write(" ",Line),nl,
    writedevice(datafile),
    write(" ",Line),nl,
    writedevice(screen),
    fail.
  write_lines.
```

Последний предикат write_lines позволяет удовлетворить цель, если первый вариант правила был неуспешен ввиду исчерпания утверждений из базы.

Пример чтения данных

```
/* Чтение данных из файла и их вывод на экран */
domains
  str = string
  file = datafile
predicates
  read_write_lines
goal
  openread(datafile,"SHAKE1.DAT"),
  readdevice(datafile),
  read_write_lines,
  closefile(datafile).
clauses
  read_write_lines :-
    not(eof(datafile)),
    readln(Line),
    writedevice(screen),
    write(" ",Line),nl,
    read_write_lines.
  read_write_lines.
```

Правило read_write_lines использует встроенный предикат Prolog eof, который дает успех, если обнаружен признак конца файла. Если в процессе чтения данных достигается конец файла, то никакие считывания больше не будут производиться, — если, конечно, не переместить указатель файла на какую-либо позицию, предшествующую метке конца файла. Неуспешной будет любая подцель, пытающаяся произвести чтение из файла при указателе, стоящем на этой метке.

Пример чтения с клавиатуры и записи в файл

```
/* Считывание данных с клавиатуры (до "end")
   и их запись в файл на диске */
domains
  file = datafile
  dstring = string
```

```
predicates
  readin(dstring)
  create_a_file
goal
  create_a_file
clauses
  create_a_file :-
  write("Введите имя файла:"),
  readln(Filename),
  openwrite(datafile,Filename),
  writedevice(datafile),
  readln(Dstring),
  readin(Dstring),
  closefile(datafile).
  readin("end") :- !.
  readin(Dstring) :-
    write(Dstring),
    readln(Dstring1),
    readin(Dstring1).
```

В Prolog существует предикат, позволяющий поместить указатель в файле в нужную позицию, — `flepos(Logical_filename,File_position,Mode)`. Параметру `File_position` здесь должно быть присвоено целое число, обозначающее номер позиции в файле, где позже будет считан или записан символ. Параметру `Mode` можно присвоить одно из трех значений: 0, 1 или 2, которое определяет, как будет интерпретировано значение `File_position`:

0 — смещение относительно начала файла;
1 — смещение относительно текущей позиции;
2 — смещение относительно конца файла.

Например, в выражении `filepos(players,100,0)` параметр `players` — это имя логического файла, параметр `File_position` имеет значение 100, что указывает на то, что будет прочитан символ, отстоящий на 100 позиций, а значение 0 для параметра `Mode` указывает, что отсчет будет производиться от начала файла. В другом примере — `filepos(players,100,1)` — отсчет будет вестись уже относительно текущей позиции указателя: если этот предикат будет успешен, то указатель сдвинется на 100 позиций вперед.

2.9. Создание динамических баз данных в языке Prolog

2.9.1. Базы данных на Prolog

В Prolog имеются специальные средства для организации баз данных (БД), которые рассчитаны на работу с *реляционными базами данных*. Внутренние унификационные процедуры языка Prolog осуществляют автоматическую выборку фактов с нужными значениями известных параметров и присваивают значения еще не определенным. Механизм отката же позволяет находить все имеющиеся ответы на сделанный запрос.

Описание предикатов динамической базы данных выполняется следующим образом [4]:

```
database
  dplayer(name,team,position)
```

Напомним, что раздел `database` в Prolog предназначен для описания предикатов базы данных. Все различные утверждения этого предиката составляют *динамическую базу данных Prolog*. Она называется динамической, поскольку во время работы программы из нее можно удалять любые содержащиеся в ней утверждения, а также добавлять новые. В этом состоит ее отличие от «статических» баз данных, где утверждения являются частью кода программы и не могут быть изменены во время ее работы. Другая важная особенность динамической базы данных состоит в том, что она может быть записана на диск, а позже считана с диска в оперативную память. Важным является и то, что в динамической базе данных могут содержаться только факты (но не правила).

Иногда бывает предпочтительно хранить часть информации базы данных в виде утверждений статической БД; тогда эти данные заносятся в динамическую БД сразу после активизации программы. Для этой цели используются специальные предикаты. В целом предикаты статической БД имеют другое имя, но ту же самую форму представления данных, что и предикаты динамической. Например, предикат стати-

ческой БД, соответствующий предикату dplayer для динамической базы данных, может быть описан так:

```
predicates
  player(name,team,position)
clauses
  player("Dan Marino","Miami Dolphins","QB").
  player("Richard Dent","Chicago Bears","DE").
  player("Bernie Kosar","Cleveland Browns","QB").
  player("Doug Cosbie","Dallas Cowboys","TE").
  player("Mark Malone","Pittsburgh Steelers","QB").
```

2.9.2. Предикаты динамической базы данных в языке Prolog

В Prolog имеются специальные встроенные предикаты для работы с динамической базой данных: asserta, assertz, retract, save, consult, readterm и findall.

Предикаты asserta, assertz и retract позволяют занести факт в заданное место динамической БД или удалить из нее уже имеющийся факт. При этом предикат asserta заносит новый факт в базу данных, располагающуюся в оперативной памяти компьютера. Новый факт помещается *перед* всеми уже внесенными утверждениями данного предиката. Синтаксис:

```
asserta(Clause).
```

Например, чтобы поместить в БД утверждение

```
player("Bernie Kosar","Cleveland Browns","QB").
```

перед уже имеющимися там утверждениями, необходимо применить следующее предикатное выражение:

```
asserta(dplayer("Bernie Kosar",
                "Cleveland Browns","QB")).
```

Предикат assertz (так же, как и asserta) заносит новые утверждения в базу данных, но помещает новое утверждение *после* всех уже имеющихся в базе утверждений того же предиката. (Запомнить просто: последняя буква предиката — «a» или «z». Буква «a» стоит *перед* всеми другими буквами английского алфавита, а «z» — *после* всех букв; точно

так же эти предикаты размещают и добавляемое утверждение.) Синтаксис:

```
assertz(Clause).
```

Предикат `retract` удаляет утверждение из динамической БД. Его синтаксис:

```
retract(Existing_clause).
```

Например, для удаления из базы данных утверждения `dplayer("Doug Cosbie","Dallas Cowboys","TE")` необходимо написать выражение:

```
retract(dplayer("Doug Cosbie","Dallas Cowboys","TE")).
```

Так же, как `asserta` и `assertz`, предикат `retract` применим только в отношении фактов.

Для модификации базы данных можно использовать комбинацию выражений с предикатами `asserta`, `assertz` и `retract`.

Предикаты `save` и `consult` применяются для записи динамической БД в файл на диск и для загрузки содержимого файла в динамическую БД соответственно. При этом предикат `save` сохраняет находящуюся в оперативной памяти базу данных в текстовом файле. Синтаксис:

```
save(File_name).
```

Например, можно записать:

```
save("football.dba").
```

В результате все утверждения находящейся в оперативной памяти динамической БД будут записаны в файл `football.dba`. Если же файл с таким именем уже имелся на диске, то его прежнее содержимое будет стерто.

Файл БД может быть считан в память (загружен) при помощи предиката `consult`. Систаксис:

```
consult(File_name).
```

Предикат `consult` неуспешен, если файл с указанным именем отсутствует на диске, если этот файл содержит ошибки (например, при несоответствии синтаксиса предиката из

файла описаниям из раздела программы database) или если содержимое файла невозможно разместить в памяти из-за отсутствия места.

Предикат readterm используется для чтения из файла объектов, относящихся к определенному в программе домену. Синтаксис:

```
readterm(Domain,Term).,
```

где Domain задает имя домена, а Term — различные наборы значений объектов этого домена. Например, в предикатном выражении:

```
readterm(auto_record,auto(Name,Year,Price)).
```

Domain замещен на auto_record, а Term — на auto (Name,Year,Price), где терм auto определяет все наборы значений этого домена. При этом описание доменов должно выглядеть так:

```
domains
  name = string
  year = integer
  price = real
  auto_record = auto(name,year,price)
  file = auto_file
```

Для получения доступа к файлу сначала необходимо воспользоваться предикатами openread и readdevice, после чего можно применить readterm.

Пример:

```
/* Демонстрация примера работающей базы данных.
   База данных допускает следующие операции:
   добавление, удаление, выборку и просмотр данных.
   Эта программа создает базу данных и содержит
   ее в оперативной памяти */
database
  dplayer (symbol,symbol,byte,byte)
   /* предикат включает в себя имя игрока,
  название команды, игровой номер и количество
  сыгранных сезонов*/
```

```
predicates
  player (symbol,symbol,byte,byte)
  redirect /*пересылка данных в динамическую БД*/
  rule
  menu (byte) /*организует обработку выбранного
               пункта меню*/
clauses
  player("Dan Marino","Miami Dolphins",13,4).
  player("Richard Dent","Chicago Bears",95,4).
  player("Bernie Kosar","Cleveland Browns",19,2,).
  player("Doug Cosbie","Dallas Cowboys",84, 8).
  player("Mark Malone","Pittsburgh Steelers",16,7,).

  rule:-write("1.Добавление данных в БД"),nl,
        write("2.Удаление данных об указанном
               игроке"),nl,
        write("3.Поиск сведений об игроках
               указанной команды"),nl,
        write("4.Просмотр всей БД"),nl,
        write("5.Выход"),nl,
        readint(N),
        N<>5,
        menu(N),
        rule.
  rule.

  menu(1):-write("Введите имя, название команды,
                  игровой номер и количество
                  сыгранных сезонов."),nl,
           write("Имя: "),readln(Fam),
           write("Название команды:"),readln(Team),
           write("Игровой номер:"),readint(Number),
           write("Количество сыгранных сезонов: "),
                  readint(Plays),
           assertz(dplayer(Fam,Team,Number,Plays)).
  menu(2):-write("Укажите имя игрока для удаления
                  сведений о нем:"),nl,
           readln (Fam),
           retract(dplayer(Fam,_,_,_)).
```

```
  menu(2):- write("Такого игрока нет!"),nl,menu(2).
  menu(3):- write("Укажите название команды для
                   поиска информации об игроках:"),nl,
            readln (Team),
            write("Игроки этой команды:"),nl,
            dplayer(X,Team,_,_),
            write(X),nl,
            fail.
  menu(3).
  menu(4):- dplayer(F,T,N,K),
            write("Имя: ",F),nl,
            write("Название команды: ",T), nl,
            write("Игровой номер: ", N), nl,
            write("Количество сыгранных сезонов: ",
                  K),nl,nl,
            fail.
  menu(4).

  redirect:- player(X,Y,Z,P),
             assertz(dplayer(X,Y,Z,P)),
             fail.
  redirect:- write("Пересылка данных успешно завершена.
                    Нажмите ENTER"),nl,readln(_).

goal
  redirect,
  rule.
```

2.10. Создание экспертных систем

2.10.1. Структура экспертной системы

Чтобы выполнять работу эсперта, компьютерная программа должна быть способна решать задачи посредством логического вывода и получать при этом достаточно надежные результаты. При этом программа должна иметь доступ к системе фактов, называемой *базой знаний*, а также должна во время консультации уметь выводить заключения из информации, имеющейся в базе знаний. Некоторые экспертные системы могут кроме этого использовать новую информацию, добавляемую во время консультации.

Таким образом, экспертную систему можно представить состоящей из трех частей [4]:

1) базы знаний (БЗ);
2) механизма вывода (МВ);
3) системы пользовательского интерфейса (СПИ).

База знаний — это основная, центральная часть экспертной системы. Она содержит правила, описывающие отношения (или явления), методы и знания для решения задач из области применения данной экспертной системы.

Механизм вывода содержит принципы и правила работы. Фактически механизм вывода запускает экспертную систему в работу, определяя, какие правила нужно вызвать, и организуя доступ к ним в базе знаний. Механизм вывода также выполняет правила, определяет, когда найдено приемлемое решение, и передает результаты программе интерфейса с пользователем.

Интерфейс — это та часть экспертной системы, которая непосредственно взаимодействует с пользователем. Система интерфейса с пользователем принимает информацию от него. Затем система интерфейса, основываясь на виде и характере информации, введенной пользователем, передает необходимые данные механизму вывода. А когда механизм вывода возвращает знания, выведенные из базы знаний, интерфейс передает их обратно пользователю в удобной для него форме.

2.10.2. Представление знаний

Представление знаний — это множество соглашений по синтаксису и семантике, согласно которым описываются объекты. Хорошее правило при проектировании представления знаний — организация знаний в такой форме, которая позволяет легко осуществлять доступ с помощью простых и естественных механизмов.

Один из способов представления знаний — классификация и помещение фактов и чисел (фрагментов фактического знания) в правила Prolog. Это представление подходит для использования в *экспертных системах, базирующихся на правилах.*

В настоящее время системы, базирующиеся на правилах, наиболее популярны. Они разработаны и используются в ши-

роком диапазоне приложений от науки и инженерной работы до бизнеса. Поэтому именно такие системы мы будем подробно рассматривать в этом разделе.

2.10.3. Методы вывода

Метод вывода — это систематический способ для доказательства того, что из множества предположений следует некоторое заключение. Этот систематический способ закодирован в *правилах вывода*, которые специфицируют принятую логику получения заключения. Вывод осуществляется посредством поиска и сопоставления по образцу. При этом механизм вывода инициализирует процесс сопоставления начиная с «верхнего» правила. Обращение к правилу называется его «*вызовом*»; вызов соответствующих правил в процессе сопоставления продолжается до тех пор, пока не произошло сопоставление или не исчерпана вся база знаний, а сопоставление не найдено. Во втором случае трансформированные запросы являются значениями, которые сопоставляются со значениями, находящимися в базе знаний. Если механизм вывода обнаруживает, что можно вызвать более одного правила, то необходимо осуществить определенный выбор, при этом приоритет обычно отдается либо правилам, которые более специфицированы, либо правилам, которые учитывают больше текущих данных. Этот процесс называется *разрешением конфликта*.

2.10.4. Система пользовательского интерфейса

Система пользовательского интерфейса обеспечивает взаимодействие между экспертной системой и пользователем. Такое взаимодействие обычно включает в себя следующие функции:

1) обработку данных, полученных с клавиатуры, и отображение вводимых и выводимых данных на экране дисплея;
2) поддержку диалога между пользователем и системой;
3) распознавание ситуаций непонимания между пользователем и системой;
4) обеспечение «дружественности» системы по отношению к пользователю.

Система пользовательского интерфейса должна эффективно обрабатывать ввод и вывод. Для этого требуется обрабатывать вводимые и выводимые данные быстро, представляя их четко и в наглядной форме. Кроме того, система интерфейса должна поддерживать соответствующий диалог между пользователем и системой. Консультация должна завершаться четким утверждением, выдаваемым системой, и объяснением последовательности вывода, приведшей к этому утверждению.

2.10.5. Экспертная система, базирующаяся на правилах

Экспертная система, базирующаяся на правилах (на языке Prolog) содержит множество правил, которые вызываются посредством входных данных в момент сопоставления. Такая экспертная система также содержит интерпретатор в механизме вывода, который выбирает и активизирует различные модули системы. Работу этого интерпретатора можно описать как последовательность трех шагов:

1) интерпретатор сопоставляет образец правила с элементами данных в базе знаний;
2) если можно вызвать более одного правила, то интерпретатор использует механизм разрешения конфликта для выбора правила;
3) интерпретатор применяет выбранное правило, чтобы найти ответ на заданный вопрос.

Этот процесс интерпретации является циклическим и называется *циклом «распознавание—действие»* [4]. В системе, базирующейся на правилах, количество продукционных правил определяет размер базы знаний. Некоторые наиболее сложные системы имеют базы знаний из более 5000 продукционных правил.

Пример. Построим экспертную систему для идентификации породы собак, которая должна помочь потенциальному хозяину выбрать породу собаки в соответствии с определенными критериями. Предположим, что пользователь сообщил множество характеристик собаки в ответ на вопросы

экспертной системы. Интерпретатор работает в цикле «распознавание–действие». Если введенные характеристики сопоставимы с характеристиками какой-либо породы собаки, составляющими часть базы знаний, то вызывается соответствующее продукционное правило и в результате идентифицируется порода, а затем результат сообщается пользователю. Если же порода не идентифицирована, то это тоже сообщается пользователю.

```
/*Декларации базы данных. База данных будет хранить
ответы пользователя на вопросы системы пользователь-
ского интерфейса. Эти данные представляют собой
ответы "Да" или "Нет". */
database
  dpos(symbol)
  dneg(symbol)
  /*Предикаты для выполнения вывода и для взаимодей-
  ствия с пользователем */
predicates
  do_expert
  do_consult
  dog_is(symbol)
  clear_facts
  prop(symbol)
clauses
  do_expert:- write("Отвечайте на вопросы, выбирая
               характеристики собаки (y/n)"),nl,nl,
              do_consult.

  /*Консультирующее правило do_consult имеет две
  две альтернативные формы. Первая взаимодействует
  с механизмом вывода: если результат цикла
  "распознавание–действие" положительный, то
  результат сообщается пользователю. Вторая форма
  сообщает об отрицательном результате */
  do_consult:- dog_is(X),!,
               write("Вам подойдет собака породы ",
                     X),nl,readln (_),
               clear_facts.
```

```
  do_consult:- write("Извините, но мы не можем Вам
                      помочь."),nl,
               readln (_),
               clear_facts.

/*Механизм вывода*/
  /*Правило prop используется для сопоставления
  данных пользователя с данными в продукционных
  правилах. Правило также производит добавление
  предложений с ответами "y" ("да") и "n" ("нет")
  для использования при сопоставлении с образцом */
  prop(X):- dpos(X),!.
  prop(X):- not(dneg(X)),
            write ("Устраивает ли Вас
                    характеристика?",X),nl,
             readln(A),
             A="y",
             asserta(dpos(X)).
prop(X):-   asserta(dneg(X)),
            fail.
clear_facts:- retract(dpos(_)),
              fail.
clear_facts:- retract(dneg(_)),
              fail.

/*Продукционные правила*/
  dog_is("Английский бульдог"):-
    prop("короткая шерсть"),
    prop("рост 55 см"),
    prop("низко посажен хвост"),
    prop("хороший характер").
  dog_is("Гончая"):-
    prop("короткая шерсть"),
    prop("рост 55 см"),
    prop("длинные уши"),
    prop("хороший характер").
  dog_is("Дог"):-
    prop("короткая шерсть"),
    prop("низко посажен хвост"),
    prop("хороший характер"),
    prop("вес 45 кг").
```

```
  dog_is("Коккер-спаниель"):-
    prop("длинная шерсть"),
    prop("рост 55 см"),
    prop("низко посажен хвост"),
    prop("длинные уши"),
    prop("хороший характер").
  dog_is("Ирландский сеттер"):-
    prop("длинная шерсть"),
    prop("рост 75 см"),
    prop("длинные уши").
  dog_is("Сенбернар"):-
    prop("длинная шерсть"),
    prop("низко посажен хвост"),
    prop("хороший характер"),
    prop("вес 45 кг").
goal
  do_expert.
```

Контрольные вопросы и задания к главе 2

1. Перечислите известные вам основные методологии программирования.
2. Охарактеризуйте методологию императивного программирования.
3. Какова основа методологии объектно-ориентированного программирования?
4. В чем состоят отличия методологии функционального программирования?
5. Какова основа методологии логического программирования?
6. Охарактеризуйте методологию программирования в ограничениях.
7. Что такое высказывание?
8. Что такое терм? Приведите примеры термов.
9. Что такое резолюция? В чем состоит основа метода резолюции?
10. Объясните на примере механизм резолюции.
11. Как резолюцию используют для доказательства теорем?

12. Приведите примеры резолюции в языке Prolog. Объясните, что именно делает система языка Prolog с вашей программой.

13. Что такое унификация переменных?

14. Что такое конкретизация переменных?

15. Что такое бэктрекинг?

16. Перечислите основные разделы программы на языке Prolog и укажите их назначение.

17. Какая цель называется внутренней, а какая — внешней?

18. Перечислите отличительные особенности внешних и внутренних целей.

19. Какой объект называют составным? Как описываются составные объекты?

20. Что такое альтернативный домен? Как он описывается?

21. Каково назначение метода отката после неудачи? Какой предикат реализует этот метод?

22. Каково назначение метода отсечения и отката? Какой предикат реализует этот метод?

23. Какое правило называют правилом рекурсии? Каков синтаксис записи такого правила?

24. Что такое список в языке Prolog? Какова структура и организация списка? Приведите примеры списков.

25. Что называют «головой» списка, а что — его «хвостом»?

26. Как описать предикат списка и списочный домен?

27. Какой встроенный предикат позволяет собрать данные из базы данных в список для их последующей обработки?

28. Что называют строкой? Как задать строковую константу?

29. Перечислите известные вам операции над списками. Какими предикатами они реализуются?

30. Перечислите предикаты Prolog для работы с файлами. Каково назначение каждого из них?

31. Как описывается файловый домен?

32. Укажите последовательность действий, необходимых для записи информации в файл.

33. Укажите последовательность действий, необходимых для чтения данных из файла.

34. Укажите последовательность действий, требуемых для модификации уже существующего файла.

35. Перечислите шаги, необходимые для добавления новых данных в конец файла.

36. Какая база данных называется динамической? Как описать предикат динамической базы данных?

37. Перечислите известные вам предикаты для занесения факта в заданное место динамической базы данных и для удаления из нее уже имеющегося факта.

38. Перечислите предикаты для записи динамической базы данных в файл на диск и для загрузки содержимого файла в динамическую базу данных.

39. Какова структура экспертной системы, базирующейся на правилах?

40. Каково назначение интерпретатора в механизме вывода? Опишите работу этого интерпретатора.

Литература к главе 2

1. *Дейкстра Э.* Дисциплина программирования. — М.: Мир, 1978.

2. *Клоксин У., Меллиш К.* Программирование на языке Пролог / Пер. с англ. — М.: Мир, 1987.

3. *Малпас Дж.* Реляционный язык Пролог и его применение / Пер. с англ. Под. ред. В. Н. Соболева. — М.: Наука, 1990.

4. *Соломон Д.* Использование Турбо-Пролога / Пер. с англ. — М.: Мир, 1993.

5. *Себеста Р.* Основные концепции языков программирования. — М.: Вильямс, 2001.

6. *Тей А., Грибомон П., Луи Ж. и др.* Логический подход к искусственному интеллекту: от классической логики к логическому программированию / Пер. с фр. — М.: Мир, 1990.

7. Язык Пролог в пятом поколении ЭВМ: Сб. статей (1983–1986 гг.) / Сост. Н. И. Ильинский. — М.: Мир, 1988.

ГЛАВА 3
НЕЙРОННЫЕ СЕТИ

3.1. Введение в нейронные сети

Еще одно базовое направление развития искусственного интеллекта основано не на моделировании процесса принятия решения человеком (как в экспертных системах), а на попытке создать *нейронную модель мозга*. В силу слабой изученности физиологии человеческого мозга адекватность такой модели представляется весьма сомнительной. Тем не менее даже те малые (а возможно, и ошибочные) знания физиологии мозга, которые были получены в 1960-е гг., позволили создать работоспособные нейронные модели [1].

Теория искусственных нейронных сетей (*НС*) зародилась еще в 1940-х гг., и к 1960-м гг. уже были разработаны *однослойные НС* (*перцептроны*), которые в ряде случаев оказались способны обучаться, осуществлять предсказания и распознавать образы. Появилась надежда, что механизм работы мозга «угадан» правильно, и скоро удастся создать настоящий искусственный интеллект, — надо лишь создать (на программном или аппаратном уровне) побольше нейронов. Однако перцептроны хорошо показали себя именно «в ряде случаев», тогда как во многих других, казалось бы, похожих ситуациях они работать почему-то отказывались. Один из «отцов-основателей» теории нейронных сетей, Марвин Минский, используя точные математические методы, доказал, что существует теоретический предел (причем довольно низкий) возможностей однослойных сетей, и они не способны решать многие простые задачи. К тому же он высказал собственное мнение, что нейронные сети перспектив не имеют. Авторитет Минского был настолько велик, а его расчеты — настолько убедительны, что все исследования в этой области повсеместно были свернуты, и в течение почти 20 лет интерес к ним не проявлялся, а энтузиастов-одиночек никто не хотел слушать. Только в 1980-х гг. в этой области

произошел прорыв благодаря революционным работам Джона Хопфилда и Тейво Кохонена. Многослойные нейронные сети нового поколения успешно справлялись с задачами, которые были недоступны для перцептронов. Началось лавинообразное развитие данного направления; к этому моменту искусственный интеллект уже рассматривался как вполне утилитарная область с достаточно четко очерченными задачами.

Элементная база позволила создать мощные *нейрокомпьютеры* и *программные нейропакеты* для распознавания образов, прогнозирования и решения ряда других задач, в которых входные данные были неполны, зашумлены и даже противоречивы [4]. Значительное финансирование эти исследования получили даже не столько благодаря широкому внедрению НС-систем в военное дело, сколько из-за их способности делать весьма точные прогнозы в социально-экономической сфере (в бизнесе). Так, современные нейросистемы способны предсказывать колебания курсов валют и акций на достаточном промежутке времени точнее самых опытных брокеров, учитывая при этом массу косвенных параметров — от солнечной активности до политической ситуации в стране. Поэтому те же брокеры сегодня уже не обходятся без использования различных НС-систем — от простых программных реализаций за 300 долларов для ноутбуков до дорогих нейронных суперкомпьютеров. При этом такие системы, как правило, предполагают обязательное наличие доступа в сеть Интернет, что необходимо для обеспечения следующих задач:

- получение входных данных для анализа (например, котировок акций в различных системах валют, индексов инфляции, текущих параметров рекламной активности и политической ситуации и множества других);
- реализация интерфейса (брокер может работать с большим корпоративным нейрокомпьютером через сеть Интернет);
- доступ к базам данных, необходимым для обучения нейросистемы (например, данных о биржевой ситуации за какой-либо прошедший период). Чем большее количество примеров будет представлено системе для обучения, тем более точных прогнозов можно от нее добиться. Эта информация является достаточно дорогой, но существуют базы с данными за десятки лет.

Однако подобные нейронные сети имеют и некоторые специфические недостатки [3]. Прежде всего они связаны с недостатками механизмов обучения нейронных сетей. Не вдаваясь в детали, отметим, что все разнообразные механизмы обучения имеют свои слабые места, которые могут привести к неправильной настройке системы. Другой недостаток состоит в том, что нейронные сети не могут объяснять свои действия (процесс получения того или иного решения), что является сильной стороной экспертных систем.

Наряду с прогнозированием еще одной очень полезной способностью нейронных сетей, как уже отмечалось, является распознавание образов в условиях зашумленной входной информации. И здесь опять-таки, кроме военного применения (обнаружение, слежение и т. п.), для нейронных сетей существует масса приложений в сфере бизнеса, среди которых можно назвать определение разнообразных рисков и поддержку принятия решений. Последнее в перспективе может оказаться наиболее эффективным при объединении нейронных сетей и экспертных систем в единый комплекс (так же, как правое полушарие головного мозга человека отвечает за первичную обработку входной информации от органов чувств и рефлекторную деятельность, а левое — за логический анализ и принятие решений). При этом первичная обработка нейронной сетью скорее всего потребует наибольших ресурсов. Так, например, с математической точки зрения мозг футболиста в момент удара по мячу производит гораздо больше вычислительных операций, чем мозг шахматиста за всю партию.

За утилитарным использованием нейронных сетей собственно моделирование работы мозга отошло на второй план. Здесь продвинуться удалось не так сильно, и успехи в практическом применении мало что доказывают. Даже самые мощные нейрокомпьютеры не сравнимы по своей сложности даже с нервной системой муравья, — куда уж там до человека! Да и, с другой стороны, достижения нейробиологов базируются в основном на изучении морских зайчиков (двустворчатых моллюсков). Работа же человеческого мозга связана с электромагнитной активностью и другими слабо изученными полями, роль которых почти неизвестна. К тому же нейроны отнюдь не однородны и не равноценны даже у того же мор-

ского зайчика. Человеческий мозг — это, наверное, самое сложное, что создала природа, и его изучение — долгий и трудный процесс даже на простом физиологическом уровне. Вспомним хотя бы, как долго ученые, а вместе с ними и все остальные обыватели заблуждались, считая, что нервные клетки «не восстанавливаются». В силу всего этого очень сомнительна популярная идея о том, что если удастся считать всю информацию с мозга человека и перенести ее в компьютер, то мы получим «виртуальную копию личности», которая будет «жить» в машине или в Глобальной сети, даруя бессмертие данному индивиду...

Таким образом, отождествлять искусственный интеллект с мышлением человека можно лишь весьма условно. Пожалуй, самым большим их сходством является способность совершать ошибки. Познание человеком самого себя идет медленнее, чем развитие технологий ИИ, которые стали самостоятельным научным направлением (хотя сама научная терминология сохраняет «человекообразный» стиль — «нечеткая логика», «генетические алгоритмы» и т. д.).

В последние несколько лет мы наблюдаем буквально «взрыв» интереса к нейронным сетям, которые успешно применяются в самых различных областях — в бизнесе, медицине, технике, геологии, физике. Нейронные сети вошли в практику везде, где нужно решать задачи прогнозирования, классификации или управления. Такой впечатляющий успех определяется несколькими причинами [3].

Богатые возможности. Нейронные сети — это исключительно мощный метод моделирования, позволяющий воспроизводить чрезвычайно сложные зависимости. В частности, нейронные сети нелинейны по своей природе. На протяжении многих лет линейное моделирование было основным методом моделирования в большинстве областей, поскольку для него хорошо разработаны процедуры оптимизации. В задачах же, где линейная аппроксимация неудовлетворительна (а таких задач достаточно много), линейные модели работают плохо. Кроме того, нейронные сети справляются с «проклятием размерности», которое не позволяет моделировать линейные зависимости в случае большого числа переменных.

Простота в использовании. Нейронные сети *учатся на примерах*. Пользователь нейронной сети подбирает представительные данные, а затем запускает *алгоритм обучения*, который автоматически воспринимает структуру данных. При этом от пользователя, конечно, требуется какой-то набор эвристических знаний о том, как нужно отбирать и подготавливать данные, выбирать нужную архитектуру сети и интерпретировать результаты, однако уровень знаний, необходимый для успешного применения нейронных сетей, гораздо скромнее, чем, например, при использовании традиционных методов статистики.

Нейронные сети привлекательны и с интуитивной точки зрения, поскольку они основаны на примитивной биологической модели нервных систем. В будущем развитие таких нейробиологических моделей может действительно привести к созданию мыслящих компьютеров либо к реализации возможности прямого подключения традиционных компьютеров к мозгу, когда нейросеть выполняет функции некоего «промежуточного звена» для распознавания «мыслей» пользователя (управляющих сигналов) среди снятой с мозга полной энцефалограммы; возможно, такая нейросеть будет «обучаться» взаимодействию с конкретным пользователем («хозяином») еще с младенчества.

Нейронные сети возникли из исследований в области искусственного интеллекта, а именно из попыток воспроизвести способность биологических нервных систем обучаться и исправлять ошибки, моделируя низкоуровневую структуру мозга.

Основной областью исследований по искусственному интеллекту в 1960–1980-е гг. были экспертные системы, которые базировались на высокоуровневом моделировании процесса мышления (в частности, на представлении, что процесс нашего мышления построен на манипуляциях с символами). Однако очень скоро стало ясно, что такие системы хотя и могут принести пользу в некоторых областях, но не охватывают некоторые ключевые аспекты человеческого интеллекта. Согласно одной из распространенных точек зрения, причина в том, что они не в состоянии воспроизвести структуру мозга, а чтобы создать искусственный интеллект, необходимо построить систему с похожей архитектурой.

Мозг человека состоит из очень большого числа (приблизительно 10 000 000 000) *нейронов*, соединенных многочисленными связями (в среднем — несколько тысяч связей на один нейрон, однако это значение может сильно меняться). **Нейрон** — это специальная клетка, способная распространять электрохимические сигналы. Нейрон имеет разветвленную структуру вводных каналов информации (*дендриты*), *ядро* и разветвляющийся выходной канал (*аксон*). Аксоны такой клетки соединяются с дендритами других нейронов клеток с помощью *синапсов*. Будучи активированным, нейрон посылает электрохимический сигнал по своему аксону, и через синапсы этот сигнал достигает других нейронов, которые могут, в свою очередь, активироваться. Нейрон активируется, когда суммарный уровень сигналов, пришедших в его ядро из дендритов, превысит определенный уровень (*порог активации*).

Интенсивность сигнала, получаемого нейроном (а следовательно, и возможность его активации), сильно зависит от активности синапсов. Каждый синапс представляет собой промежуток (*синаптическую щель*) между аксоном и дендритом, и специальные химические вещества (*медиаторы*) передают сигнал через этот промежуток. Один из самых авторитетных исследователей нейросистем, Дональд Хебб, сформулировал постулат, что обучение заключается в первую очередь в изменениях «силы» синаптических связей. Например, в классическом опыте Павлова каждый раз непосредственно перед кормлением собаки звонил колокольчик, и собака быстро научилась связывать звонок колокольчика с пищей. Это произошло потому, что синаптические связи между участками коры головного мозга, ответственными за слух, и слюнными железами усилились, так что при возбуждении коры мозга звуком колокольчика у собаки начиналось слюноотделение.

Таким образом, будучи построенным из очень большого числа совсем простых элементов (каждый из которых фактически вычисляет взвешенную сумму входных сигналов и если суммарный вход превышает определенный уровень, то передает дальше двоичный сигнал), мозг способен решать чрезвычайно сложные задачи.

3.2. Искусственная модель нейрона

Составные элементы нейросети

Самый важный элемент нейросистемы — это *адаптивный сумматор* [3]. Он вычисляет скалярное произведение вектора входного сигнала x на вектор параметров. Адаптивным мы называем его из-за наличия *вектора настраиваемых параметров* (рис 3.1).

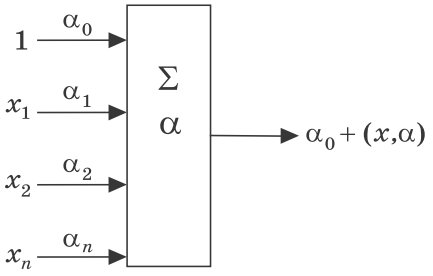


Рис. 3.1. Адаптивный сумматор

Следующий по важности элемент нейросистемы — *нелинейный преобразователь сигнала* (рис. 3.2). Он получает скалярный входной сигнал x и преобразует его в значение функции $\varphi(x)$.

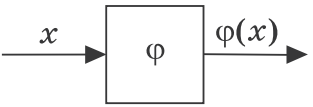


Рис. 3.2. Нелинейный преобразователь сигнала

Точка ветвления (рис. 3.3) служит для рассылки одного сигнала по нескольким адресам. Она получает скалярный входной сигнал x и передает его на все свои выходы.

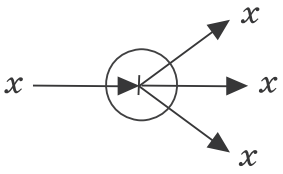


Рис. 3.3. Точка ветвления

Линейная связь (*синапс*; рис. 3.4) обычно отдельно от сумматора не встречается, но для некоторых рассуждений бывает удобно выделить ее как отдельный элемент. Он умножает входной сигнал x на «вес синапса».

Рис. 3.4. Синапс

Наконец, *стандартный формальный нейрон* (рис. 3.5) составлен из входного сумматора, нелинейного преобразователя и точки ветвления на выходе.

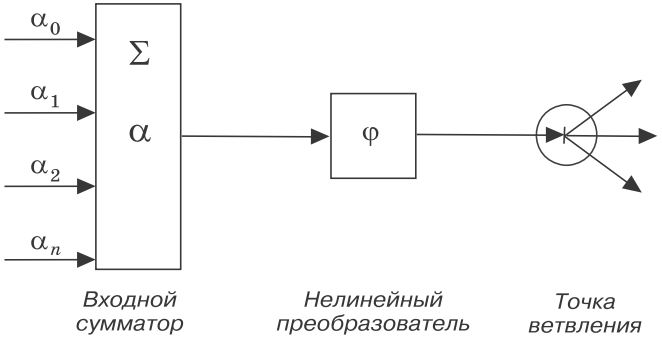


Рис. 3.5. Стандартный формальный нейрон

Чтобы отразить суть биологических нейронных систем, *определение искусственного нейрона* можно дать следующим образом [3]:

- он получает входные сигналы (исходные данные либо выходные сигналы от других нейронов нейронной сети) через несколько входных каналов;
- каждый входной сигнал проходит через соединение, имеющее определенную интенсивность (*вес*), соответствующую синаптической активности биологического нейрона;
- с каждым нейроном связано определенное пороговое значение: вычисляется взвешенная сумма значений сигналов на входах, из нее вычитается пороговое значение и тем самым вычисляется *величина активации нейрона* (она также называется *постсинаптическим потенциалом нейрона — PSP*);
- сигнал активации преобразуется с помощью *функции активации* (или *передаточной функции*) в выходной сигнал нейрона.

Если при этом использовать ступенчатую функцию активации (т. е. когда выход нейрона равен нулю, если на вход подано отрицательное значение, и единице, если значение на входе нулевое или положительное), то такой нейрон будет работать точно так же, как описанный ранее биологический нейрон. Заметим, что вычесть пороговое значение из взвешенной суммы и сравнить результат с нулем — это то же самое, что сравнивать взвешенную сумму с пороговым значением, но реализуется гораздо проще, поэтому пороговые функции (сравнение с нулевым порогом) в искусственных нейронных сетях используются редко. Кроме того, важно учесть, что веса могут быть отрицательными, — это означает, что такой синапс оказывает на нейрон не возбуждающее, а тормозящее воздействие (и в живом мозге действительно присутствуют такие тормозящие нейроны).

Это было описание отдельного нейрона. Теперь возникает вопрос: как соединять нейроны друг с другом? Если нейросеть предполагается для чего-то использовать, то у нее дол-

жны быть входы (принимающие значения интересующих нас переменных из внешнего мира) и выходы (прогнозы или управляющие сигналы). Входы и выходы такой нейросети соответствуют имеющимся в живом организме сенсорным и двигательным нервам, — например, идущим от глаз и к рукам, соответственно. Кроме того, в нейросети может присутствовать ряд промежуточных (скрытых) нейронов, выполняющих какие-либо внутренние функции. При этом входные, скрытые и выходные нейроны должны быть связаны между собой.

Ключевой вопрос здесь — *обратная связь*. Простейшая сеть имеет *структуру с прямой передачей сигнала*: сигналы проходят от входов через скрытые элементы и рано или поздно поступают на выход. Такая структура имеет устойчивое поведение. Если же сеть является *рекуррентной* (т. е. содержит связи, ведущие назад от дальних к более ближним нейронам), то она может быть неустойчивой и иметь очень сложную динамику поведения. Рекуррентные сети представляют большой интерес для исследователей в области нейронных сетей, однако при решении практических задач (по крайней мере до сих пор) наиболее полезными оказались структуры с прямой передачей.

Типичный пример сети с прямой передачей сигнала показан на рис. 3.6. Здесь нейроны регулярным образом органи-

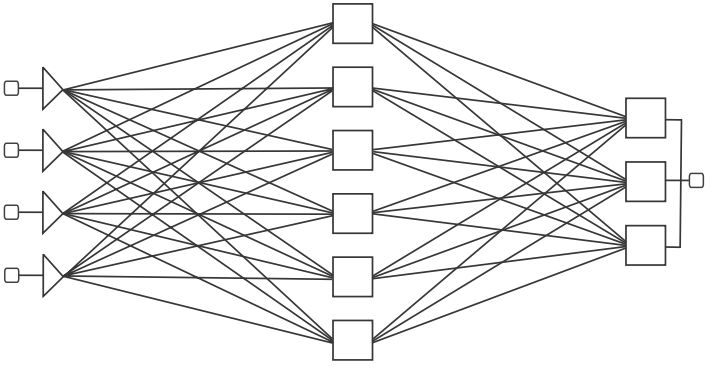


Рис. 3.6. Пример нейронной сети с прямой передачей сигнала

зованы в *слои*, при этом входной слой служит для ввода значений входных переменных, а каждый из скрытых и выходных нейронов соединен со всеми элементами предыдущего слоя. (Можно было бы рассматривать и сети, в которых нейроны связаны только с некоторыми из нейронов предыдущего слоя, однако для большинства практических приложений использование сети с полной системой связей предпочтительнее.)

При работе (использовании) сети в ее входные элементы подаются значения входных переменных. Затем последовательно вступают в работу нейроны промежуточных и выходного слоев: каждый из них вычисляет свое значение активации, беря взвешенную сумму выходов элементов предыдущего слоя и вычитая из нее свое пороговое значение, а затем полученное значение активации преобразуется с помощью функции активации и в результате получается значение сигнала на выходе нейрона. После того как вся сеть закончит работу, выходные значения элементов выходного слоя принимаются за выходные значения всей сети в целом.

3.3. Применение нейронных сетей

На практике нейросети используются как программные продукты, выполняемые на обычных компьютерах, либо как специализированные аппаратно-программные комплексы [2]. Заметим, что в первом случае встроенный параллелизм нейросетевых алгоритмов чаще всего не используется, поскольку для многих задач анализа и обобщения баз данных особого быстродействия не требуется — для них вполне хватает производительности современных универсальных процессоров. В таких приложениях используется исключительно способность нейросетей к обучению и к извлечению закономерностей, скрытых в больших массивах информации. Для второй же группы приложений, обычно связанных с обработкой сигналов в реальном времени, параллелизм нейровычислений является критическим фактором.

Перечислим основные задачи, решаемые нейронными сетями.

1. *Распределенная ассоциативная память.* Распределенная память означает, что веса связей нейронов имеют статус информации без специфической ассоциации части информации с отдельным нейроном. Ассоциативная память означает, что нейронная сеть способна выдать на выход полный образ по предъявленной на входе его части.
2. *Распознавание образов.* Задачи распознавания образов требуют способности одновременно обрабатывать большое количество входной информации и выдавать категорический или обобщенный ответ. Для этого нейронная сеть должна обладать внутренним параллелизмом.
3. *Адаптивное управление.*
4. *Прогнозирование.*
5. *Экспертные системы.*
6. *Оптимизация* (т. е. поиск максимума функционала при наличии ограничений на его параметры).

В настоящее время нейросети находят широкое применение в различных областях, таких как экономика (предсказание показателей биржевого рынка, предсказание финансовых временных рядов), робототехника (распознавание оптических и звуковых сигналов, самообучение), визуализация многомерных данных, ассоциативный поиск текстовой информации и др.

Нейронные сети представляют интерес для достаточно большого числа специалистов:

- для компьютерщиков нейросети открывают область новых методов для решения сложных задач;
- физики используют нейросети для моделирования явлений в статистической механике и для решения многих других задач;
- нейрофизиологи могут использовать нейронные сети для моделирования и исследования функций мозга;
- психологи получают в свое распоряжение механизм для тестирования моделей некоторых своих психологических теорий;

- другие специалисты (особенно коммерческих и индустриальных направлений) также могут интересоваться нейронными сетями по самым разнообразным причинам прежде всего благодаря достигаемым с их помощью новым возможностям прогнозирования и визуализации данных.

3.4. Обучение нейросети

Способность к обучению является фундаментальным свойством мозга. В контексте же нейросетей процесс обучения может рассматриваться как настройка архитектуры сети и весов связей для эффективного выполнения некоторой специальной задачи. Обычно нейронная сеть должна настроить свои веса связей по имеющейся *обучающей выборке* [1], причем функционирование сети улучшается по мере итеративной настройки весовых коэффициентов. Свойство нейросетей обучаться на примерах делает их более привлекательными по сравнению с системами, которые работают по жестко определенному набору правил функционирования, сформулированных экспертами.

Для конструирования процесса обучения нейросети прежде всего необходимо иметь *модель внешней среды*, в которой должна функционировать эта нейронная сеть, т. е. знать доступную для сети информацию. Эта модель определяет *парадигму обучения*. Во-вторых, необходимо понять, как именно следует модифицировать весовые параметры сети, т. е. какие *правила обучения* управляют процессом настройки. *Алгоритм обучения* обозначает процедуру, в которой используются правила обучения для настройки весов.

Существуют три *парадигмы обучения нейросетей*: «с учителем», «без учителя» (самообучение) и смешанная [1].

В первом случае нейронная сеть располагает правильными ответами (требуемыми выходами сети) для каждого входного примера, а веса настраиваются так, чтобы сеть производила ответы, как можно более близкие к известным правильным ответам. Усиленный же вариант обуче-

ния «с учителем» предполагает, что известна только критическая оценка правильности выхода нейронной сети, но не сами правильные значения выхода.

«Обучение без учителя» не требует знания правильных ответов для каждого примера обучающей выборки. В этом случае раскрывается лишь внутренняя структура данных или корреляции между образцами в системе данных, что позволяет распределить образцы по категориям.

При смешанном обучении часть весов определяется посредством обучения «с учителем», в то время как остальные веса формируются с помощью самообучения.

Теория обучения нейросетей рассматривает три фундаментальных свойства, связанных с обучением на примерах: емкость, сложность образцов и вычислительная сложность. При этом под *емкостью* понимается, сколько образцов может запомнить сеть и какие функции и границы принятия решений могут быть на ней сформированы. *Сложность образцов* определяет количество обучающих примеров, необходимых для достижения способности сети к обобщению. Слишком малое количество таких примеров может вызвать «переобученность» сети, когда она хорошо функционирует на примерах обучающей выборки, но плохо работает на тестовых примерах, подчиненных тому же статистическому распределению.

Известны четыре основных типа правил обучения: коррекция по ошибке, машина Больцмана, правило Хебба и обучение методом соревнования.

Правило коррекции по ошибке. При обучении «с учителем» для каждого входного примера задан желаемый выход d, однако реальный выход сети y может не совпадать с желаемым. Принцип коррекции по ошибке при обучении состоит в использовании разностного сигнала $(d - y)$ для модификации весов, обеспечивающей постепенное уменьшение ошибки. Такое обучение производится только в случае, когда нейросеть ошибается. При этом существуют различные модификации этого алгоритма обучения, которые здесь подробно не рассматриваются.

Обучение Больцмана. Представляет собой стохастическое правило обучения, которое следует из принципов теории информации и термодинамических принципов. Целью обучения Больцмана является такая настройка весовых коэффициентов, при которой состояния нейронов внешнего слоя удовлетворяют желаемому распределению вероятностей. Обучение Больцмана можно рассматривать как специальный случай коррекции по ошибке, в котором под ошибкой понимается расхождение корреляций состояний в двух режимах.

Правило Хебба. Наиболее старым обучающим правилом является постулат обучения Хебба. Хебб опирался на следующие нейрофизиологические наблюдения: если нейроны с обеих сторон синапса активизируются одновременно и регулярно, то сила синаптической связи возрастает. Важной особенностью этого правила является то, что изменение синаптического веса здесь зависит только от активности нейронов, которые связаны данным синапсом.

Обучение методом соревнования. В отличие от обучения Хебба, в котором множество выходных нейронов может возбуждаться одновременно, при соревновательном обучении выходные нейроны соревнуются между собой за активизацию. Это явление известно как правило «победитель берет все». Подобное обучение имеет место в биологических нейронных сетях. Обучение посредством соревнования позволяет кластеризовать входные данные: аналогичные входные примеры группируются сетью в соответствии с корреляциями и представляются одним элементом.

При обучении методом соревнования модифицируются только веса «победившего» нейрона. Эффект от этого правила достигается за счет такого изменения сохраненного в сети образца (вектора весов связей «победившего» нейрона), при котором он становится чуть ближе ко входному примеру.

Контрольные вопросы и задания к главе 3

1. В чем заключается суть направления развития искусственного интеллекта, основанного на попытке создать нейронную модель мозга?
2. Каковы современные аспекты применения нейросистем?
3. Каковы недостатки нейронных сетей?
4. В чем заключаются преимущества нейронных сетей?
5. Из каких элементов состоит модель искусственного нейрона?
6. Как работает искусственный нейрон?
7. Как строятся нейронные сети?
8. Какие задачи решаются с помощью нейронных сетей?
9. Как производится обучение нейронной сети?
10. Какие типы правил обучения нейросетей вы знаете?

Литература к главе 3

1. *Галушкин А. И.* Теория нейронных сетей. — М.: Издательское предприятие редакции журнала «Радиотехника», 2000.
2. Нейронные сети в системах автоматизации / В. И. Архангельский, И. Н. Богаенко, Г. Г. Грабовский и др. — Киев: Техника, 1999.
3. *Терехов С. А.* Лекции по теории и приложениям искусственных нейронных сетей: http://alifenarod.ru/lectures/neural/Neu_index.htm.

Минимальные системные требования определяются соответствующими требованиями программ Adobe Reader версии не ниже 11-й либо Adobe Digital Editions версии не ниже 4.5 для платформ Windows, Mac OS, Android и iOS; экран 10"

Учебное электронное издание

Серия: «Педагогическое образование»

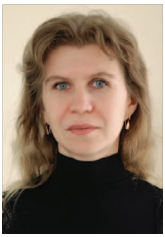
ОСНОВЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА
Учебное пособие

Ведущий редактор *Д. Усенков*
Художники *Н. Новак, С. Инфантэ*
Технический редактор *Е. Денюкова*
Корректор *Л. Макарова*
Компьютерная верстка: *С. Янковая*

Подписано к использованию 24.03.20.
Формат 125×200 мм

Издательство «Лаборатория знаний»
125167, Москва, проезд Аэропорта, д. 3
Телефон: (499)157-5272
e-mail: info@pilotLZ.ru, http://www.pilotLZ.ru

БОРОВСКАЯ ЕЛЕНА ВЛАДИМИРОВНА
Старший преподаватель кафедры информатики и методики преподавания информатики Челябинского государственного педагогического университета.
Область интересов: проблемы модульно-рейтинговой системы контроля и оценки учебных достижений студентов в условиях управления качеством в вузе.



ДАВЫДОВА НАДЕЖДА АЛЕКСЕЕВНА
Кандидат педагогических наук по специальности «Теория и методика обучения и воспитания (информатика, уровень общего образования)», доцент кафедры информатики и методики преподавания информатики Челябинского государственного педагогического университета.
Области интересов: технология формирования содержания образования по информатике в профильных классах общеобразовательных школ, интеллектуальные обучающие системы.

Учебное пособие знакомит читателей с историей искусственного интеллекта, моделями представления знаний, экспертными системами и нейронными сетями. Описаны основные направления и методы, применяемые при анализе, разработке и реализации интеллектуальных систем. Рассмотрены модели представления знаний и методы работы с ними, методы разработки и создания экспертных систем. Книга поможет читателю овладеть навыками логического проектирования баз данных предметной области и программирования на языке Prolog.
Книга предназначена для студентов и преподавателей педагогических вузов, учителей общеобразовательных школ, гимназий и лицеев.