DESIGNING AND PROGRAMMING PERSONAL EXPERT SYSTEMS

CARL TOWNSEND
AND DENNIS FEUCHT

TAB TAB BOOKS Inc.
Blue Ridge Summit, PA 17214

681.3.06
Т 23

К.ТАУНСЕНД
Д.ФОХТ

ПРОЕКТИРОВАНИЕ И ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ЭКСПЕРТНЫХ СИСТЕМ НА ПЕРСОНАЛЬНЫХ ЭВМ

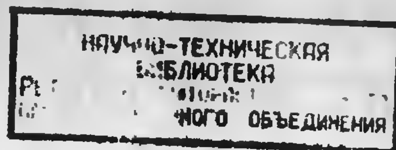
Перевод с английского
В.А.Кондратенко, С.В.Трубицына

Предисловие Г.С.Осипова

МОСКВА
"ФИНАНСЫ И СТАТИСТИКА"
1990

ББК 24.4.1
Т12

234 486



Таунсенд К., Фохт Д.
Т12 Проектирование и программная реализация экспертных систем на персональных ЭВМ: Пер. с англ./Предисл. Г.С. Осипова. - М.: Финансы и статистика, 1990. - 320 с.: ил.

ISBN 5-279-00255-0.

В книге доступно изложены основные методы и приемы программирования экспертных систем. Технология программирования экспертных систем описана с применением языка Форт. Стандарт Форт-83 имеется на отечественных микроЭВМ ДВК-2М, ЕС 1840, "Электроника-85". Большой интерес представляют приведенные в книге тексты Форт-программ, реализующих процедуры обработки списков и методы логического программирования.

Для специалистов в области экспертных систем, профессиональных программистов, преподавателей и студентов вузов.

Т $\frac{2404010000 - 056}{010(01) - 91}$ 133 – 89

ББК 24.4.1

(С) 1986 by Carl Townsend and Dennis Feucht
(С) Г.С. Осипов, предисловие, 1990

ISBN 5-279-00255-0 (СССР)
ISBN 0-8306-2692-1 (США)

ПРЕДИСЛОВИЕ К РУССКОМУ ИЗДАНИЮ

Сегодня всем, кто работает в области информатики или интересуется этой новой областью науки, известен термин "экспертные системы". Экспертными системами (ЭС) называют компьютерные программы, способные накапливать знания, которые содержатся в различных источниках, и моделировать процесс экспертизы, т.е. решение специалистами той или иной области неформализуемых задач на основе своего профессионального опыта.

Книга американских авторов К.Таунсенда и Д.Фохта представляет собой введение в круг основных идей искусственного интеллекта (ИИ) и методов программной реализации элементов ЭС на ЭВМ. Большинство рассмотренных в книге методов иллюстрируются текстами программ, написанных на Форте - оригинальном языке, созданном специально для персональных компьютеров.

Путь от исследовательских проектов нескольких научных центров США до признания специалистами по информатике и пользователями в качестве одной из ведущих информационных технологий экспертные системы "прошли" значительно быстрее, чем их предшественники - информационные системы и базы данных. Причинами такого уникального успеха являются, во-первых, многообразие сфер приложений, включающее такие далекие друг от друга области, как юриспруденция и производство космической техники, сельское хозяйство и создание систем противоракетной обороны, а во-вторых, возможность использования экспертных систем непрофессионалами в качестве инструмента в своей повседневной работе.

С появлением экспертных систем искусственный интеллект (ИИ), одну из ветвей которого они составляют, перестал считаться чисто теоретической научной дисциплиной и стал рассматриваться как стратегически важное направление исследований. Идеи и результаты исследований в области ИИ положили за рубежом начало целой отрасли электронной промышленности. Если прикладная задача может быть решена на основе знаний экспертов и эти знания можно определенным способом выявить и представить на некотором формальном языке, то применение ЭС оказывается весьма эф-

фективным. Как показывает опыт, вложенные в разработку экспертных систем средства не только быстро окупаются, но и приносят значительную прибыль.

Признание пользователей, позволившее экспертным системам быстро занять достойное место среди других средств информатики, во многом обусловлено успехом первых проектов, в частности, таких, как MYCIN, PROSPECTOR, DENDRAL и др. Первые проекты завершились созданием экспертных систем, ныне по праву считающихся классическими. Методы их разработки использованы при построении огромного числа ЭС в самых различных областях. По данным зарубежных изданий в 1988 г. насчитывалось более 5 тыс. видов ЭС и инструментальных средств для их построения. Постоянно расширяющийся рынок ЭС привлек к ним внимание многих фирм, которые поставили их производство на промышленную основу. Ряд университетов и научных центров развернули исследования, направленные на совершенствование архитектуры и методов разработки ЭС. К концу 1987 г. в этой отрасли уже было занято более 600 организаций.

Становление индустрии экспертных систем в свою очередь стимулировало дальнейший прогресс исследований по ИИ. Сформировалась новая область информатики, инженерия знаний, которая имела целью создание технологии выявления знаний и наполнения ими ЭС. Возникла и соответствующая новая специальность - инженер знаний. При проектировании конкретной ЭС в его функции входят построение концептуальной модели предметной области, выбор эффективных способов представления знаний и механизмов вывода.

В условиях расширяющегося спроса на ЭС многие фирмы в США сочли заманчивым привлечь для разработки экспертных систем более мощные программные и аппаратные средства. В качестве аппаратных средств широкое распространение получили специализированные рабочие станции, обладающие по сравнению с универсальными ЭВМ более высоким быстродействием. Однако такая стратегия развития ЭС, вопреки ожиданиям, не только не заинтересовала новых заказчиков, но и стала препятствием в расширении их сектора на рынке ИИ в США. С конца 1986 г. некоторые фирмы-изготовители ЭС начали испытывать затруднения в сбыте своих программных продуктов, реализованных на рабочих станциях, - спрос на экспертные системы снизился. Причины этого, как отмечают зарубежные специалисты, заключены в высокой стоимости специализированных рабочих станций и реализованных на их основе инструментальных средств, а также в сложности работы с ними.

Еще одна серьезная причина кроется в непонимании назначения ЭС и, в какой-то степени, в идеализации их возможностей, что выражается в попытке применить экспертные системы для решения таких творческих задач, для которых точно неизвестны

механизмы получения решения. Но экспертные системы, разумеется, - не волшебный ключик, способный открыть любой ларец, да они никогда и не претендовали на подобную роль. ЭС могут оказаться действительно полезными только в тех случаях, когда хорошо известны способы поиска решения, т.е. когда эксперт может точно описать логику решения задачи.

Проблемы сбыта, обусловленные такой дезориентацией пользователей, заставили разработчиков пересмотреть концепцию построения и применения своих программных продуктов. Это проявилось, во-первых, в переходе от спецпроцессоров к универсальным компьютерам и, во-вторых, в отказе от использования специальных языков программирования, ориентированных на задачи ИИ, в пользу традиционных, таких, как Си и Паскаль. В результате заказчики получили возможность создавать ЭС на имеющихся у них ЭВМ и не приобретать дорогостоящие рабочие станции. Предпринятые шаги вызвали в США новую волну заказов на ЭС, в том числе весьма крупных от Министерства обороны, и позволили стабилизировать их положение на рынке программных средств[1].

В связи с высоким интересом к ЭС со стороны специалистов различных областей знания и теми надеждами, которые на них возлагаются, естественно возникает вопрос: каковы тенденции развития экспертных систем в ближайшем будущем, т.е. какие исследования в данной области будут реализованы на практике? Судя по ориентации работ ведущих научных центров у нас в стране и за рубежом, наиболее перспективными и плодотворными с точки зрения приложений можно считать следующие направления.

1. Интеграция ЭС с традиционными пакетами программ самого различного назначения: вычислительной математики, оптимизации, обработки информации (табличные процессоры и СУБД) и создание гибридных ЭС. Такой подход позволяет, во-первых, использовать ЭС в качестве внешней по отношению к упомянутым пакетам управляющей программы, которая осуществляет вызов нужного пакета в зависимости от вида решаемой задачи, и, во-вторых, дает возможность учитывать качественные факторы при решении количественных или аналитических задач.

Объединение ЭС с базами данных и электронными таблицами открывает новые возможности в автоматизации делопроизводства и экономического анализа на предприятиях и в учреждениях, дает руководителям, плановикам и статистикам мощный инструмент обобщения и многоаспектного анализа разнородной информации. Особое место занимает интеграция ЭС с системами автоматизированного проектирования (САПР). В составе интеллектуальных САПР экспертные системы смогут взять на себя часть важных функций проектировщика по согласованию параметров различных

[1] Электроника. - 1988. - N 12-13. - С.70.

элементов проекта, а также генерацию вариантов какой-либо части проекта и выбору из них того, который в наибольшей степени отвечает требованиям технического задания.

Не менее важным представляется и объединение ЭС с обучающими системами. В процессе обучения на ЭС возлагается выполнение анализа ответов ученика и выбор оптимальной стратегии взаимодействия с ним, которая дает ему возможность приобрести необходимые навыки за минимальное время. Обучающие системы с элементами ИИ - одно из наиболее перспективных приложений ЭС, поскольку успехи в этой области позволяют значительно повысить качество обучения и резко снизить затраты на него (в некоторых случаях до 10-12 раз).

2. Создание ЭС реального времени. Экспертные системы подобного рода оказываются крайне необходимыми при управлении непрерывными процессами. Они дают возможность повысить надежность и эффективность управления не только обычными технологическими линиями, но и такими важными объектами, как, например, ядерные реакторы, на которых необходима автоматизированная "подстраховка" действий операторов в экстремальных ситуациях.

3. Построение распределенных ЭС. Ряд сложных задач требует для своего решения привлечения знаний из различных предметных областей. Согласованная обработка разнородных знаний может быть выполнена в распределенной экспертной системе. Взаимодействие между составляющими ее автономными ЭС осуществляется путем передачи сообщений через специальный блок, называемый доской объявлений. Работа распределенной ЭС имитирует коллективное решение сложной проблемы группой специалистов, каждый из которых знаком лишь с одним ее аспектом.

4. Разработка динамических ЭС. Системы данного класса позволяют моделировать динамические предметные области. Изменения, произошедшие в таких областях после начала решения задачи, оказывают влияние на окончательное заключение и поэтому должны учитываться непосредственно в процессе вывода (возможно, после каждого его шага). Появление динамических ЭС даст возможность автоматизировать ряд важных задач мониторинга, которые не могут быть решены с помощью традиционных экспертных систем.

На развитие ЭС оказывает безусловное влияние прогресс во всех смежных областях информатики и вычислительной техники, и особенно в микроэлектронике, продолжающей развиваться необычайно быстрыми темпами. Можно ожидать, что разработка ЭС на базе быстродействующих микропроцессоров и систем оптической памяти откроет новую страницу в их применении. Однако наиболее широкая перспектива повышения эффективности ЭС связывается с появлением транспьютеров - однокристальных микропроцессоров с быстродействием до 10 млн. операций в секунду и средст-

вами коммуникации, позволяющими объединять их в сети различной архитектуры. В сети транспьютеров могут параллельно выполняться несколько независимых процессов, в том числе и поиск решения. Иными словами, возникает реальная возможность применять экспертные системы для решения задач очень высокой сложности.

Несмотря на то что в настоящее время в основном уже определены принципы, на которых могут базироваться ЭС перечисленных выше классов и даже созданы соответствующие прототипы, в целом проблемы построения экспертных систем с нетрадиционной архитектурой пока еще далеки от своего решения. Немало открытых вопросов остается и в сфере разработки традиционных ЭС, также нуждающихся в совершенствовании. Поэтому столь важным представляется выпуск хороших книг по ЭС, способных привлечь в эту активно развивающуюся отрасль информатики свежие силы разработчиков. Книга, которую вы держите в руках, несомненно, принадлежит к их числу.

Нужно отметить, что издательства сегодня не обходят вниманием тему ЭС и книги по данной проблематике периодически появляются на прилавках магазинов. В них более или менее детально рассматриваются назначение, структура, области применения и принципы построения ЭС. Однако этап собственно программирования ЭС остается как бы "за кадром", и разработчик должен самостоятельно преодолевать возникающие здесь трудности. В некоторой степени восполнить пробел в русскоязычной литературе, посвященной упомянутым выше проблемам, поможет подготовленный к выпуску издательством "Финансы и статистика" ряд переводов[1], которые можно рассматривать как практические руководства по реализации ЭС на распространенных языках программирования. Они хорошо дополняют ранее опубликованные работы, так как дают читателю возможность не только проследить весь путь создания ЭС, но и приобрести необходимые программистские навыки.

Наметившийся в последнее время в публикациях сдвиг от изложения концептуальных основ ЭС к описанию методов их программной реализации неоспоримо свидетельствует о том, что уже накоплен достаточный опыт разработок, который может быть передан даже начинающим. Американские специалисты К.Таунсенд и Д.Фохт относятся к числу тех, кто взял на себя эту непростую задачу.

---

[1]Помимо предлагаемого читателю издания, к ним относятся следующие книги: Левин Р., Дранг Д., Эделсон Б. Практическое введение в технологию искусственного интеллекта и экспертных систем с иллюстрациями на Бейсике. – М.: Финансы и статистика, 1990; Сойер Б., Фостер Д. Программирование экспертных систем на Паскале. – М.: Финансы и статистика, 1989.

Предлагаемая вашему вниманию книга состоит из двух частей. Первая часть представляет собой доступно написанное введение в проблематику ИИ, включая небольшой экскурс в его историю. Здесь в основном рассматриваются традиционные вопросы - назначение ЭС, их отличительные особенности, методы представления знаний в ЭС. Особо необходимо отметить, к сожалению, краткую, но исключительно важную последнюю главу первой части, где обсуждаются проблемы идентификации предметной области, а также выбора подходящего способа извлечения и представления знаний, т.е. инженерии знаний.

Вторая часть книги посвящена методам программирования, применяемым при разработке ЭС. Авторы подробно описывают операции над списками и процедуры логического вывода. У прагматически настроенного читателя может возникнуть вопрос, стоит ли тратить время на изучение столь сложных элементов программирования, если существуют оболочки ЭС, которые нужно лишь наполнить предметными знаниями для получения готовой системы. На этот вопрос следует ответить положительно по двум причинам.

Во-первых, знакомство с внутренним устройством ЭС и схемой ее работы позволит выбрать наиболее подходящий способ представления знаний, что отразится не только на времени поиска решения, но и может увеличить число задач, решение которых окажется возможным. Заметим попутно, что выбор адекватного представления знаний является стратегической задачей, а последствия его неудачного выбора могут оказаться катастрофическими[1].

Во-вторых, поскольку чисто экспертные и чисто вычислительные задачи на практике встречаются не так уж часто, может возникнуть необходимость объединения программных средств для обработки знаний и для проведения расчетов. Создание интерфейса между оболочками ЭС, представляющими собой сложные, замкнутые комплексы, с традиционными пакетами программ - дело нелегкое. Но даже решив проблему, вы получите громоздкую и плохо поддающуюся управлению систему, отладка которой займет немало времени из-за появления непредсказуемых ошибок.

Более подходящим в такой ситуации может оказаться надстройка механизмов вывода над существующим программным обеспечением, для чего желательно использовать тот язык, на котором ведется разработка прикладного пакета. Именно по этой причине полезно знать способы программной реализации одного из важнейших компонентов ЭС - блока логического вывода.

Следует также иметь в виду, что обработка списков и процедуры логического вывода составляют ядро таких общепризнанных языков ИИ, как Лисп и Пролог. Поэтому, зная, как реализованы

[1]См.: Вудс У.А. Основные проблемы представления знаний. - ТИИЭР, 1986. - Т. 74. - № 10.

лежащие в их основе механизмы, вы будете лучше представлять возможности и ограничения указанных языков и сможете более эффективно применять их для решения своих задач.

В качестве инструментария для разработки приведенных в книге программ авторы выбрали Форт - сравнительно "молодой" язык, приобретающий в последнее время все большее число сторонников. В СССР он менее популярен, чем, например, в США, но с выпуском необходимой литературы[1] и увеличением количества трансляторов для различных типов компьютеров Форт, надо полагать, вскоре займет подобающее ему место. Привлекательными свойствами Форта, который можно рассматривать как высокоуровневый ассемблер, являются диалоговый режим работы, высокая эффективность написанных на нем программ, а также наличие компилирующего и интерпретирующего режимов их выполнения.

Отличительная особенность Форта состоит в том, что он позволяет конструировать из базовых команд (называемых словами) языковые средства требуемой предметной ориентации. Это делает его легко расширяемым (как и Лисп) при сохранении несравненно более высокого быстродействия. Отмеченное обстоятельство послужило причиной повышенного внимания к Форту со стороны специалистов по ИИ - именно ему в последнее время все чаще отдается предпочтение. Аргументом в пользу Форта, по сравнению с другими языками программирования, служит также компактность транслятора, благодаря чему его можно применять на персональных компьютерах с небольшим объемом оперативной памяти. И хотя появление 32-разрядных микропроцессоров открывает широкие перспективы для использования любых языков, роль Форта не снизится до тех пор, пока при разработке программ первостепенное значение будут иметь их такие характеристики, как быстродействие, занимаемый объем памяти и переносимость.

Книга К.Таунсенда и Д.Фохта адресована широкому кругу читателей - и специалистам, занятым программированием ЭС, и тем, кто только начинает работать в области ИИ и желает изучить его основы. Начинающие найдут в книге необходимые сведения об истоках ИИ и его проблематике, получат представление о том, что такое немонотонные рассуждения, объектно-ориентированное программирование и параллельные процессы. Профессиональным программистам будет небезынтересно узнать о способах реализации логического вывода средствами языка со стековой организацией вычислений. Книга написана в доступной форме и содержит большое количество примеров. Приведенные листинги программ дадут терпеливому читателю возможность досконально разобраться в изложенных методах реализации ЭС.

[1]См.: Броуди Л. Начальный курс программирования на языке Форт. - М.: Финансы и статистика, 1989.

По-видимому, у читателя вызовет интерес приведенный в книге перечень наиболее распространенных ЭС и инструментальных средств, который дает определенное представление о программных продуктах, имеющихся на рынке в США. К сожалению, в этом списке не нашли отражение отечественные инструментальные средства для построения ЭС. Ввиду невозможности рассказать обо всех системах, упомянем лишь о некоторых из них.

Система DEM, разработанная во Всесоюзном институте научно-технической информации под руководством В.К. Финна, предназначена для выявления эмпирических зависимостей причинно-следственного типа в базах данных с неполной информацией. С ее помощью созданы ЭС, работающие в области фармакологии, экологии и технической диагностики.

Система АРИАДНА, разработанная во Всесоюзном научно-исследовательском институте системных исследований В.К.Моргоевым, ориентирована на поддержку процесса приобретения знаний в интерактивном режиме с участием эксперта и инженера знаний. Система позволяет осуществить быстрое прототипирование базы знаний путем уточнения постановки задачи и выявления ограничений.

Система SIMER + MIR, разработанная в Институте программных систем АН СССР под руководством автора этого предисловия, представляет собой технологию создания ЭС для задач диагностики и прогнозирования в медицине, экологии и других предметных областях. Система включает в себя модуль прямого приобретения знаний SIMER, реализующий управляемый диалог с экспертом, модуль MIR, моделирующий рассуждения типа аргументации, и модуль "+", обеспечивающий адаптацию решателя MIR к базе знаний, которая представлена в виде неоднородной семантической сети.

Система ПиЭС (программный инструментарий для ЭС) разработана в Вычислительном центре АН СССР под руководством В.Ф. Хорошевского. ПиЭС предоставляет разработчику совокупность программных средств проектирования экспертных систем и поддерживает весь процесс их создания. В распоряжении пользователя системы имеются языки представления знаний нескольких уровней, библиотека стандартных механизмов управления выводом и средства для конструирования новых управляющих механизмов. Особенностями системы является наличие средств общения на ограниченном естественном языке, многооконной графики и сетевых меню, а также подсистемы сборки новой ЭС из существующих и созданных с помощью ПиЭС блоков.

В Международном центре информатики и электроники (ИнтерЭВМ) разработан под руководством Э.В. Попова комплекс инструментальных средств для создания экспертных систем, состоящий из оболочек ЭКО и НЭКС-2. Обе оболочки работают на персональных компьютерах типа IBM PC AT. Для представления процедурных знаний в них используются правила. Декларативные знания в системе ЭКО представляются в виде конструкций типа объект-атрибут-значение, а в системе НЭКС-2 - в виде фреймов. Инструментальный комплекс располагает средствами для преобразования знаний из одной структуры представления в другую. Построенные с помощью этих оболочек ЭС позволяют проводить обработку неполных, недостоверных и противоречивых знаний и включают блок поддержания истинности.

Система СПЭИС (система проектирования экспертных интеллектуальных систем) разработана во Всесоюзном научно-исследовательском институте системных исследований О.В. Ковригиным. СПЭИС - это реализованная на языке Лисп оболочка для создания ЭС, в которой поддерживается иерархия языков представления знаний. Базовый формализм представления знаний - фреймы, включающие продукции. В системе имеется совокупность инструментальных модулей, позволяющих собирать прикладные системы, а также проводить консультации, сопровождаемые графической иллюстрацией процесса вывода.

В заключение хотелось бы отметить, что, прочитав книгу, вы не только получите общее представление об искусственном интеллекте, но и сможете самостоятельно составить программы основных компонентов "персональной" экспертной системы.

Тираж книги отпечатан с оригинал-макета, подготовленного переводчиками с помощью текстового процессора Microsoft Word 5.0 на персональном компьютере IBM PC AT. При создании оригинал-макета переводчики пользовались консультациями Г.В.Сенина, которому они выражают свою признательность. Набор шрифтов для лазерного принтера был предоставлен совместным предприятием "Параграф". Гл.1-6 книги перевел С.В. Трубицын, гл.7-11 и приложения - В.А. Кондратенко.

Г.С. ОСИПОВ,
кандидат физ. - мат. наук

ПРЕДИСЛОВИЕ

"Можно ожидать, что наступающая эра искусственного интеллекта окажет влияние на деятельность "белых воротничков" в той же степени, в какой промышленные автоматы в свое время повлияли на деятельность "синих спецовок".

Джон Диболъд

"База данных содержит данные; база знаний включает в себя как сами данные, так и описание их свойств. Сегодня такое разделение выглядит естественным. Очень скоро компьютеры станут по меньшей мере интерактивными консультантами для специалистов, получив наиболее широкое распространение в области медицины".

Джеймс Мартин

Появление экспертных систем - один из наиболее интересных этапов во всей истории существования ЭВМ. Число разработок, специалистов и программных продуктов в этой области растет экспоненциально. В 1980 г. все исследования были полностью сконцентрированы в университетах. В 1981 г. Япония объявила о своем намерении обогнать США в развитии компьютерной технологии, поставив целью создание ЭВМ пятого поколения. По мнению японских ученых, эти ЭВМ должны произвести революцию в промышленности. Ученые США взяли на вооружение другой принцип - широкомасштабность: из 500 компаний почти половина уже начали свои собственные исследования по созданию экспертных систем. Несколько компаний приступили к совместной разработке сложных систем, основанных на знаниях, причем около 10% фирм нашли им практическое применение.

Если вы хотите приобрести в личное пользование лазерный диск с тем, чтобы превратить свой компьютер в постоянно готового прийти вам на помощь эксперта, то ждать этого вам придется еще

очень и очень долго. Технологически подобная задача может быть решена уже через несколько лет, однако повсеместно наш уровень развития таков, что мы пока не готовы к тем переменам, которые вызовет внедрение в практику лазерного диска. Более реалистично ожидать, что в ближайшее время экспертные системы станут лишь средством для расширения знаний и повышения квалификации самих экспертов.

Предлагаемая вашему вниманию книга представляет собой введение в методы разработки "думающих" машин будущего. Мы познакомим вас с некоторыми закономерностями процесса мышления, а также с основными концепциями экспертных систем и способами их построения. Вы найдете в книге полную распечатку программы, реализующую небольшую экспертную систему, с которой можете начать свое путешествие в страну искусственного интеллекта, причем для этого вам не потребуется ничего, кроме бытового или профессионального компьютера.

Первая часть книги задумана как учебный курс, цель которого - познакомить читателя с начальными сведениями по экспертным системам, поскольку предполагается, что читатель обладает лишь минимальными знаниями в области искусственного интеллекта (ИИ). Здесь приводится большое число примеров. Если же вы захотите более детально изучить программы, то загляните в приложения. В первой главе рассматриваются элементы искусственного интеллекта. В гл.2 приводится общая информация о системах, основанных на знаниях, или экспертных системах. В гл.3 описываются системы конкретного типа - продукционные. В гл.4 дается обзор различных способов представления знаний, начиная со структуры продукционных систем, первоначально уже упоминавшихся в гл.3. Глава 5 посвящена методам построения баз знаний.

Вторая часть книги содержит описание системы, основанной на знаниях, которая может быть разработана средствами языка Форт. Глава 6 представляет собой введение в этот язык, являющийся программной средой для создания рассматриваемой здесь экспертной системы. В гл.7 обсуждается обработка списков - ядро символьной обработки. В гл.8 приводятся методы программирования, а в гл.9 - элементы языка Пролог, который на практике используется для реализации систем, основанных на знаниях. В гл.10 и 11 освещаются некоторые современные проблемы искусственного интеллекта.

Приложения включают библиографию, листинг программы экспертной системы, описанной в книге, перечень коммерческих экспертных систем, имеющихся на рынке в настоящее время, словарь языка Форт и текст дополнительной программы, позволяющей преобразовать Пролог-систему в интерпретатор правил.

Программа экспертной системы, представленная в книге, может быть использована как основа для создания небольших баз знаний, ориентированных на решение различных задач, в том чис-

ле и упомянутых в первой части книги. Во второй части речь идет о более сложных понятиях, применение которых возможно лишь при наличии определенного опыта в программировании. Приведенная в приложении программа написана на языке Форт-83. Для экспериментов с нею необходимо располагать какой-либо версией Форт-системы. Выбрать подходящую систему вам поможет библиография, где перечислены существующие интерпретаторы и компиляторы языка Форт[1].

Надеемся, что эта книга окажется для вас полезной. В заключение мы выражаем глубокую признательность фирме Tektronix и Майклу Фрейлингу за помощь и поддержку при написании книги.

К.Таунсенд, Д.Фохт

[1] Перечень выполненных в нашей стране реализаций языка Форт можно найти в книге: Баранов С.Н., Ноздрунов Н.Р. Язык Форт и его реализации. - Л.: Машиностроение, 1988. - *Прим. перев.*

ЧАСТЬ 1

экспертные системы - системы, основанные на знаниях

Первая часть книги вводит вас в круг основных понятий искусственного интеллекта и инженерии знаний. Она построена как учебный курс. Здесь вы найдете примеры и упражнения, которые помогут вам приобрести необходимый опыт. При желании более углубленно изучить тот или иной вопрос обратитесь к приложению, где указаны источники дополнительной информации.

Для освоения материала книги достаточно начальных сведений по математике и программированию. Большинство терминов и понятий объясняется по мере их появления в тексте. Если вы уже имеете представление об инженерии знаний, то для экономии времени можете ограничиться беглым просмотром первой главы и прочитать остальные главы, чтобы освоить терминологию, которая употребляется и во второй части книги.

Карл Таунсенд

НАУЧНО-ТЕХНИЧЕСКАЯ
БИБЛИОТЕКА
РЫБИНСКОГО ... ПРОИЗВОДСТВЕННОГО ОБЪЕДИНЕНИЯ
234 486

Глава 1

ВВЕДЕНИЕ В ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

Уже на заре цивилизации философы и естествоиспытатели стремились познать механизмы мышления человека, чтобы создать искусственные формы интеллекта. Так, греческие боги, упоминаемые в мифах Древней Греции за несколько столетий до возникновения христианства, были подобны людям, но обладали неземной красотой, умом и силой. К XIV веку человек изобрел часы, украшенные механически управляемыми фигурами - прообразами современных роботов. Они кивали головами, перемещались, били в гонг и кланялись в заранее определенные моменты времени. К XIX веку ученые открыли возможность использования "разумных" машин в производственных целях, создав печатный станок, который позволил напечатать Библию быстрее, чем это сделали бы несколько сотен переписчиков.

ЧТО ТАКОЕ МЫШЛЕНИЕ?

На всех этапах своего развития "думающие" машины вызывали многообразные, порой противоречивые эмоции. Многие рассматривали такую машину как основу могущества, путь к процветанию и изобилию, как средство самоусовершенствования. Другие видели в машинах библейского Ваала - кровожадного идола, злого и беспринципного. "Думающая" машина и на самом деле может вести себя безнравственно, поскольку в отличие от человека не может определить, правильно она поступает или нет. Все современные машины принимают решения на основе базовых правил, которые явно или неявно заложил в нее человек. Если о машине говорят, что она интеллектуальна, то это означает, что ее способы принятия решений похожи, по крайней мере до некоторой степени, на способ, который применяет человек.

Классическое толкование понятия интеллекта было предложено А.М.Тьюрингом немногим более трех десятков лет назад. Проблема формулируется в терминах имитационной игры, основанной на популярной игре под названием "Английская гостиная". Человек (А) и интеллектуальная машина (В) помещаются в разные комнаты. Экзаменатор (С), задавая вопросы, должен определить, кто ему отвечает - человек или машина. Если он не может распознать, кто есть кто, то считается, что машина "интеллектуальна". Конечно, экзаменатор должен попытаться выработать некоторую стратегию диалога, которая бы позволила ему оценить, способен ли обьект (испытуемый) познавать и творить или только повторять, как попугай, запомненные фразы. Приведем классический пример диалога для теста Тьюринга.

Экзаменатор. Если в первой строке сонета "Сравню ль тебя я с летним днем" вместо выражения "с летним днем" подставить "с весенним днем", то изменится ли что-нибудь в стихотворении?

Испытуемый. Нарушится ритм стиха.

Экзаменатор. А если подставить выражение "с зимним днем"? В этом случае ритм сохранится.

Испытуемый. Да, но кто же захочет, чтобы его сравнивали с зимним днем?

Эказаменатор. Могли бы вы сказать, что образ мистера Пиквика ассоциируется у вас с Рождеством?

Испытуемый. В некотором смысле - да.

Экзаменатор. Однако Рождество - зимний день, и я не думаю, что мистер Пиквик возражал бы против такого сравнения.

Испытуемый. Неужели вы это серьезно? Под "зимним днем" в сонете понимается обычный зимний день, а не такой праздник, как Рождество.

Подобный осмысленный диалог был бы, конечно, невозможен для любой ныне существующей или разрабатываемой машины.

ПРОЦЕСС МЫШЛЕНИЯ

Процесс мышления, протекающий в человеческом сознании, невероятно сложен. Одна ячейка человеческого глаза способна выполнить за 10 мс обработку, эквивалентную решению системы из 500 нелинейных дифференциальных уравнений со 100 переменными. Суперкомпьютеру Cray-1, самому быстродействующему на сегодня компьютеру в мире, потребовалось бы несколько минут для решения этих уравнений. Поскольку глаз человека насчитывает не менее 10 млн. ячеек и каждая из них взаимодействует с другими, суперкомпьютер Cray-1 затратил бы по меньшей мере 100 лет, чтобы воспроизвести те процессы, которые происходят в глазу ежесекундно.

Данные из внешнего мира воспринимаются человеком с помощью одного из пяти органов чувств (таких, как зрение) и затем

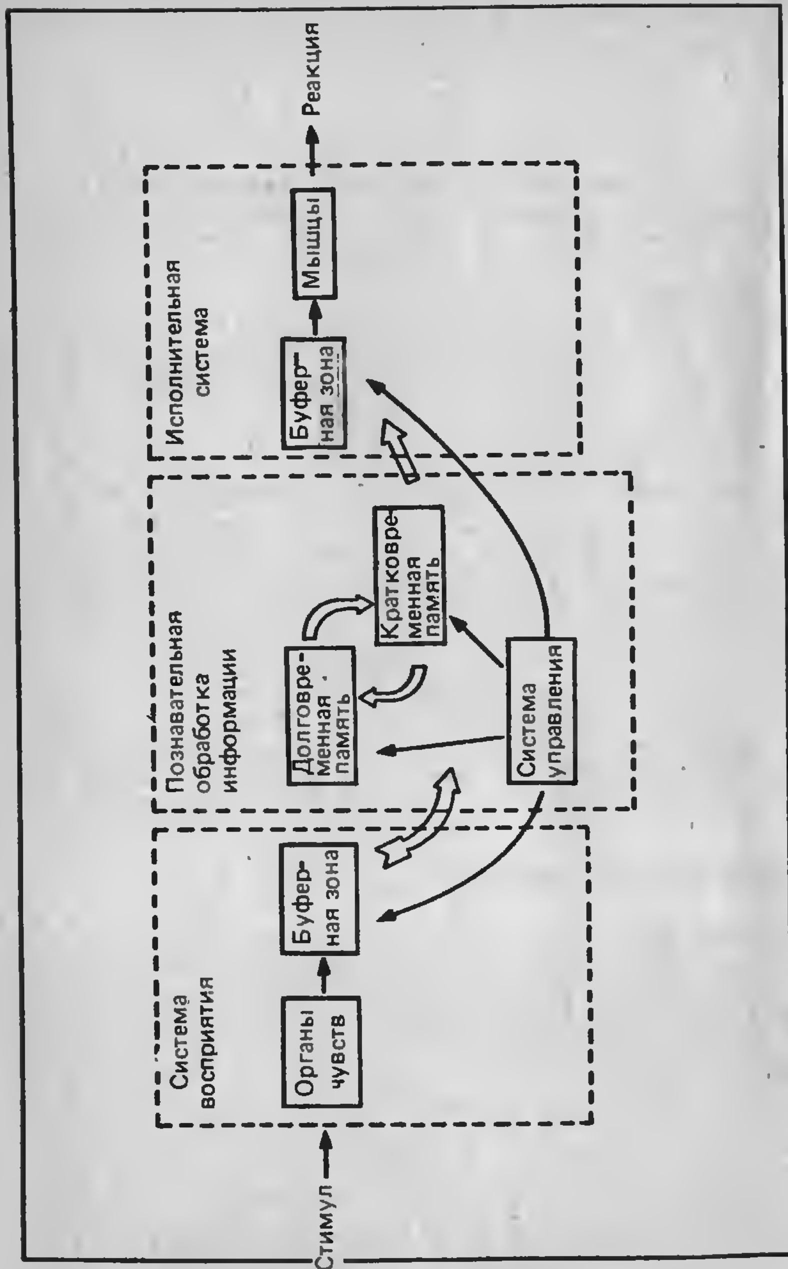


Рис. 1.1. Система обработки информации у человека

помещаются в буфер кратковременной памяти для анализа. В другой области памяти (долговременной) хранятся символы и смысловые связи между ними, которые используются для объяснения новой информации, поступающей из кратковременной памяти (рис.1.1). В долговременной памяти хранятся не столько факты и данные, сколько объекты и связи между ними, т.е. символьные образы. Большие объемы данных постоянно записываются ("дампируются") в кратковременную память, и мы непрерывно анализируем и фильтруем получаемую информацию для того, чтобы определить степень ее важности и то, как она соотносится с образами, хранящимися в долговременной памяти.

Вероятно, наиболее удивительным является тот факт, что ученые уже установили физическое местоположение всех областей памяти в мозгу. Отсюда, естественно, следует, что имеющиеся в мозгу области памяти объединяются в единую структуру,в которой и выполняется обработка поступающей информации (рис. 1.2).

Доступ к информации в долговременной памяти осуществляется весьма эффективно. Практически любой элемент данных может быть извлечен в течение цикла обработки продолжительностью не более 70 мс и затем преобразован. Например, мы "инстинктивно" отдергиваем руку от горячей печки или резко поворачиваем руль автомобиля при возникновении препятствия на дороге, используя образы, ранее запомненные в долговременной памяти.

Механизмы запоминания информации в долговременной памяти представляют собой самостоятельную и достаточно обширную тему. Требуется приблизительно 7с для "записи" одного образа в долговременную память и установления всех связей, необходимых для извлечения этого образа в будущем. Перемещение данных из кратковременной памяти в долговременную занимает приблизительно 15-20 мин. Если человек в автомобильной катастрофе получил мозговую травму, то долговременная память может восстановиться почти полностью. Однако вся информация, поступившая в последние 15-20 мин до катастрофы, будет утрачена и никогда не восстановится.

Можно провести аналогию между кратковременной памятью человека и оперативной памятью ЭВМ, для которой отключение электропитания означает полное уничтожение всех данных. Долговременная память похожа, скорее, на дисковую память, где образы существуют в виде циркулирующих электрохимических импульсов и физических нейронных взаимодействий. Человек часто полностью выздоравливает после повреждения мозга и уничтожения нейронов в результате автомобильной катастрофы, если у него не разрушены речевые центры или нейроны, управляющие двигательной системой. Нейроны других типов, даже если они частично повреждаются при травме, могут сохранять работоспособность благодаря запомненной ранее информации.

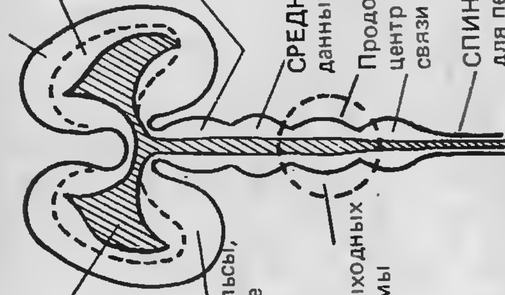


Рис. 1.2. Иерархическая организация мозга человека

ОРГАНИЗАЦИЯ ХРАНЕНИЯ ИНФОРМАЦИИ В ЧЕЛОВЕЧЕСКОЙ ПАМЯТИ

Эта тема отражает функциональный аспект долговременной памяти и потому представляет наибольший интерес для разработчиков интеллектуальных машин. Способ хранения символьных образов в долговременной памяти во многом схож со способом хранения числовой информации в базе данных сетевого типа. Реализованная на ЭВМ база данных сетового типа может быть использована для запоминания совокупности элементов, поднаборов, наборов и моделей данных. Элементы данных "принадлежат" поднаборам, наборам и моделям. Поднаборы в свою очередь "принадлежат" наборам и моделям. Конкретная модель составляется из одного или нескольких наборов, поднаборов и элементов данных (рис. 1.3).

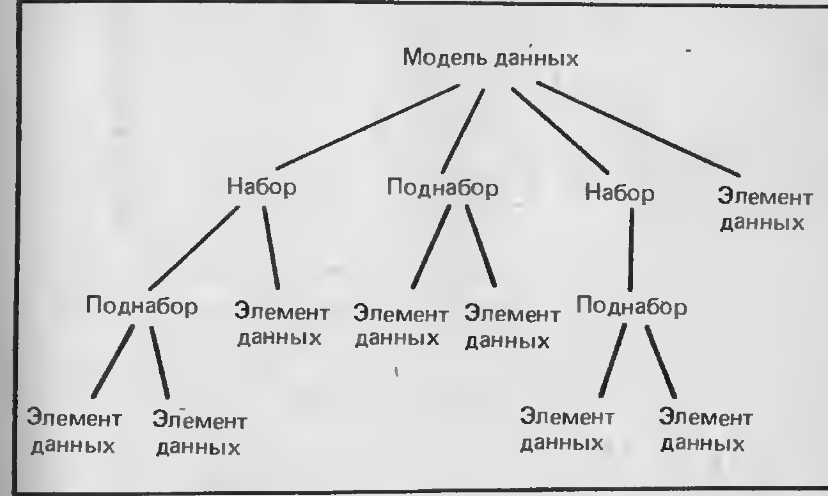


Рис. 1.3. Структура сетевой базы данных

Между моделями и наборами имеется связь типа "родитель-потомок". В отличие от общепринятых родственных связей набор может являться потомком более чем одной модели. База данных сетевого типа обладает сложной системой указателей, что облегчает пользователю (например, управляющему производством) ведение всей совокупности информационных объектов. При выборке какого-либо элемента совокупности (скажем, набора) управляющий должен иметь также возможность извлечь все связанные с ним данные (поднаборы, элементы данных и т.д.). Человеческая память хранит не числовые данные, как, например, номера элементов некоторой совокупности, а образы или символы. Хотя в памяти тоже существует система указателей, позволяющая нам быст-

ро извлечь любой нужный символ и все данные, которые с ним связаны. Однако мозг организован более совершенно и символьные образы в нем объединены в так называемые чанки - наборы фактов и связей между ними, запомненные и извлекаемые как единое целое. Чанки хранятся совместно со взаимосвязями между ними. В каждый момент времени человек может обрабатывать и интерпретировать не более четырех-семи чанков. Чтобы убедиться в этом, возьмите карандаш и бумагу, прочтите следующую строку, закройте книгу и попытайтесь написать на листе бумаги то, что вы прочли:

Нлонмйкртс зрзэйнк вибим гвссмзл

Теперь проверьте написанное и подсчитайте, сколько символов воспроизведено правильно. Вероятно, их окажется от четырех до семи.

Повторите эксперимент со следующим предложением:

Экспертные системы почти разумны.

Проверьте результат. На сей раз вы, видимо, написали все буквы правильно. Обратите внимание на то, что оба предложения состоят из одинакового числа букв и слов. Почему же так отличаются результаты? Во втором случае вы объединили все объекты (буквы) в четыре чанка и в действительности запомнили только их и связи между ними, что соответствует допустимому для человеческого мозга диапазону. В первом же примере вы должны были запомнить 30 чанков, т.е. выйти за пределы возможного.

Используя ту же стратегию, опытный шахматист может, бегло взглянув на доску, запомнить положение всех фигур. Конечно, он запоминает не положение каждой фигуры, а их комбинации (чанки) и поэтому восстанавливает расположение фигур на доске совершенно безошибочно.

Способность формировать чанки отличает эксперта в конкретной *предметной области*, или *домене*, от неэксперта. Эксперт упорно развивает свою способность объединять в чанки большие объемы данных и устанавливать иерархические связи между ними для того, чтобы быстро извлекать эти данные из памяти и с их помощью распознавать новые ситуации по мере поступления информации о них в мозг.

Средний специалист в конкретной предметной области помнит от 50000 до 100000 чанков, которые могут быть использованы для решения той или иной проблемы. Накопление в памяти человека и построение указателей для такого объема данных требуют от 10 до 20 лет.

МИР ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Проблематика искусственного интеллекта довольно обширна. Можно считать, что исследования в этой области проводились едва ли не с самого зарождения цивилизации. Начало современного этапа в развитии науки об искусственном интеллекте, вероятно, может быть датировано 1956 г., когда Клод Шеннон из фирмы Bell Laboratories и Марвин Минский из Массачусетского технологического института встретились в Дортмундском колледже с другими пионерами информатики для того, чтобы "снять покрывало" с первой в мире экспертной системы "Логик-теоретик" Аллена Ньюэлла. Список дисциплин по искусственному интеллекту постоянно увеличивается. Сегодня в него входят представление знаний, решение задач, экспертные системы, средства общения с ЭВМ на естественном языке, обучение, когнитивное моделирование, стратегические игры, обработка визуальной информации и робототехника (рис. 1.4). Искусственный интеллект является составной частью информатики (computer science), и его основной проблемой является воспроизведение на ЭВМ человеческих способов рассуждения и решения задач.

Представление знаний

Представление знаний, вероятно, является наиболее важной областью исследований по искусственному интеллекту. Это краеугольный камень всех остальных дисциплин. Данной теме мы посвящаем две главы книги (гл. 4 и 5). Знания имеют форму описаний объектов, взаимосвязей и процедур. Наличие адекватных знаний и способность их эффективно использовать означают "умение". Мозг человека очень хорошо приспособлен для символьной обработки, но при выполнении вычислений становится беспомощным даже по сравнению с маленьким калькулятором. Могут ли компьютеры воспроизвести символьную обработку, осуществляемую человеческим мозгом, и если да, то каким образом?

Создание общей теории или метода представления знаний является стратегической проблемой. Такая теория открыла бы возможность накопления знаний, которые нужны нам ежедневно для решения все новых и новых задач. Однако для достижения поставленной цели необходимо прежде всего найти способ выражения общих закономерностей нашего мира, в чем и состоит суть проблемы представления знаний.

Решение задач

Решение задач сводится к поиску пути из некоторой исходной точки в целевую. Человек делает это весьма эффективно с помо-

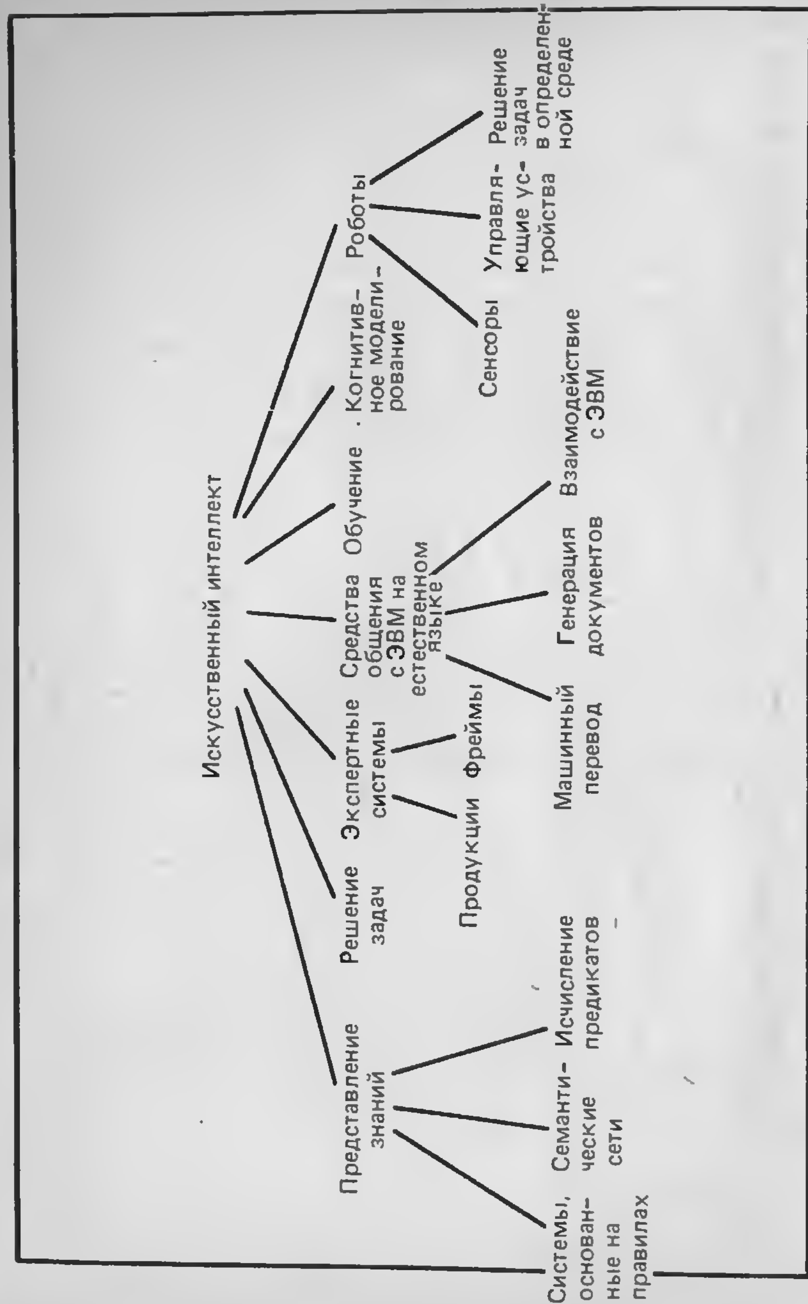


Рис. 1.4. Проблемы искусственного интеллекта

щью дедуктивного логического вывода (рассуждений), процедурального анализа, аналогии и индукции. Люди способны также учиться на собственном опыте. Компьютеры, по крайней мере в настоящее время, в общем случае решают задачи только с использованием дедуктивного логического вывода и процедурального анализа.

Тип задачи определяет метод, наиболее подходящий для ее решения. Задачи, которые сводятся к процедуральному анализу, вообще говоря, лучше всего решаются на компьютере. Учетные задачи, ведение счетов, анализ поступления наличных денег могут служить примерами процедуральных задач, решаемых компьютером быстрее и надежнее, чем человеком. Задачи же, связанные с использованием аналогии или индукции, эффективнее решаются человеком. Задачи, требующие дедуктивных рассуждений, представляются наиболее вероятными кандидатами для решения с помощью экспертных систем (систем, основанных на знаниях), которые и являются предметом нашего изучения.

Некоторые исследования в области решения задач концентрируются на разработке эффективных игровых аппаратных и программных средств, ориентированных на сложные игры, для которых нужны навыки (умения), приобретенные в других играх, таких, как шахматы и шашки. Уже разработаны шахматные программы для персональных компьютеров, соответствующие среднему уровню, и шахматные программы для больших компьютеров, играющие на уровне мастера.

Стратегии решения задач могут быть исследованы путем развития игр, которые требуют привлечения творческих способностей, вырабатывают умение у игрока или воспроизводят некоторые аспекты решения проблемы человеком. При разработке такой игры должны быть рассмотрены следующие факторы.

Допущение. Основная парадигма игры должна включать ряд ключевых решений:

* должен ли быть в игре один победитель (как в шахматах) или могут выиграть все партнеры?

* каково число играющих?

* является ли игра детерминированной (не нужно бросать жребий или отсутствует генератор случайных чисел), частично детерминированной или недетерминированной?

* является ли вся прошлая информация доступной (как в шахматах) или часть информации скрывается (как в картах)?

Представление. В каком виде игра будет представлена? В случае шахмат это означает, что необходимо определить вид фигур, их положение на доске и значения. В некоторых ситуациях игра может происходить в фантастическом мире, который является расширением реального мира. "Законы" игры могут быть определены ее создателем.

Цель. Что считать целью игры? В шахматах цель состоит в том, чтобы поставить мат противнику. Должны быть также преду-

смотрены средства, которые позволяли бы распознать, достигнуто целевое состояние или нет. Цель может быть видима игрокам (как в шахматах) или скрыта (как в приключенческих играх фирмы Infocom).

Правила игры. Какие из допустимых в игре ходов законны? В шахматах это определенные ходы, которые доступны для каждого типа фигур, так же как и групповые ходы (например, рокировки), допустимые при вполне конкретных условиях. Правила могут быть известны (как в шахматах) или неизвестны (как в приключенческих играх фирмы Infocom).

Стратегия управления. Если игра реализована на компьютере, то он должен иметь средства для "обдумывания" следующего (наилучшего) хода.

Экспертные системы

Экспертные системы представляют собой класс компьютерных программ, которые выдают советы, проводят анализ, выполняют классификацию, дают консультации и ставят диагноз. Они ориентированы на решение задач, обычно требующих проведения экспертизы человеком-специалистом. В отличие от машинных программ, использующих процедуральный анализ, экспертные системы решают задачи в узкой предметной области (конкретной области экспертизы) на основе дедуктивных рассуждений. Такие системы часто оказываются способными найти решение задач, которые неструктурированы и плохо определены. Они справляются с отсутствием структурированности путем привлечения эвристик, т.е. правил, взятых "с потолка", что может быть полезным в тех ситуациях, когда недостаток необходимых знаний или времени исключает возможность проведения полного анализа. Системам, основанным на знаниях, и посвящена главным образом эта книга.

Средства общения с ЭВМ на естественном языке

Машины обладают своим собственным языком для представления знаний и решения задач. Язык можно определить как набор символов, используемых для представления знаний (семантика), и правил, предназначенных для обработки этих символов (синтаксис) и решения задач. Человек работает наиболее эффективно, если он владеет специальными языками, которые развиваются до уровня потребностей конкретной предметной области. Когда вы приезжаете в другую страну, кто-то должен постоянно преобразовывать ваши знания, переводя их на язык этой страны. Верно и обратное - вы накапливаете информацию путем преобразования знаний, сформулированных на иностранном языке, на ваш родной язык и запоминаете их в своей базе знаний. Помощь в переводе

вам могут оказать современные электронные средства, помещающиеся в кармане.

При наличии такой интеллектуальной машины путешествие в другую страну становится более приятным. Машина имеет собственный язык, и люди часто находят свое призвание в том, чтобы научиться формулировать задачи в понятных для компьютера терминах. Точно так же компьютер выдает полученное решение на своем языке, если только оно предварительно не переведено обратно на язык пользователя.

Если правила перевода выражены в виде совокупности знаний (символов и процедур), то логично предположить, что могут быть разработаны средства, позволяющие компьютеру понимать постановку задачи на естественном языке, а затем на естественном же языке выдавать ее решение. Это основная тема исследований по разработке средств общения с ЭВМ на естественном языке. Цель исследований - установить принципы взаимодействия между людьми и на их основе создать машины, с которыми можно было бы общаться, как с людьми. Здесь можно выделить четыре ключевые проблемы.

Машинный перевод - использование компьютеров для перевода текстов с одного языка на другой.

Информационный поиск - обеспечение с помощью компьютеров доступа к информации по конкретной тематике, хранящейся в большой базе данных. В настоящее время возможен только поиск по ключевым словам. Компьютеры не в состоянии найти информацию по контексту или по аналогии.

Генерация документов - применение компьютеров для преобразования документов, имеющих определенную форму или заданных на специализированном языке, в эквивалентный документ в другой форме или на другом языке. (Например, автоматическое преобразование руководства для инженера по работе с компьютером в такое же руководство, которое может быть понятно врачу, не имеющему специальной подготовки.)

Взаимодействие с компьютером - организация диалога между неподготовленным пользователем и компьютером.

Важность разработки средств взаимодействия с компьютерами на естественном языке обусловлена необходимостью их сочетания с экспертными системами, ориентированными на неподготовленного пользователя. Роботы и другие механические формы искусственного интеллекта иногда оснащаются развитыми средствами общения на естественном языке, что упрощает их использование.

Одним из наиболее интересных примеров общения на естественном языке является система команд в игровых программах фирмы Infocom, имеющихся в большинстве персональных компьютеров. Некоторые из этих игр располагают обширным словарем, насчитывающим более 1000 слов, и дают возможность пользователю

общаться с компьютером на подмножестве английского языка (рис.1.5).

Распечатка диалога с системой "ИСПОЛНИТЕЛЬ ЖЕЛАНИЙ"
Гипотетический диалог для начинающих
Все авторские права сохраняет фирма Infokom, 1985
Редакция 68 / Порядковый номер 85050

****

>ПОСМОТРИ ВОКРУГ

Южное окончание моста. Южная сторона крытого моста, который соединяет берега реки Фестерон. Тропинки ведут на восток и на запад, дорога в южном направлении ведет к деревне.

>ИДИ НА ВОСТОК

Устье реки. Это место, где река Фестерон впадает в озеро. Дорога в южном направлении проходит по берегу реки и поворачивает на восток вдоль реки. Узкая тропинка ведет вверх к вершине Сторожевого холма.

>ПОДНИМИСЬ НА ХОЛМ

Холм Сторожевой. Вы стоите на вершине высокого, лишенного растительности холма. Вы можете увидеть деревню Фестерон, затерявшуюся среди деревьев. По северному склону холма, извиваясь, спускается вниз к реке узкая тропинка. Венчает холм пень, на месте которого когда-то стоял большой, величественный дуб. Недалеко от пня в траве лежит подкова.

>ПОДНИМИ ПОДКОВУ

Поднял.

>НА ЧТО ЭТО ПОХОЖЕ?

То, что Вы держите, напоминает подкову, золотую монету, кусочек шоколада, морскую раковину, мертвого морского конька, фиолетовый значок и зонтик.

>ПОСЛУШАЙ РАКОВИНУ И ПОДУЙ В НЕЕ

Можно услышать шум океанского прибоя. Раковина издает приглушенный звук.

>СЯДЬ НА ПЕНЬ

С этого пня открывается прекрасный вид на окрестности.

>СПУСТИСЬ ВНИЗ С ХОЛМА

(Сначала поднимитесь с пня.)

Устье реки.

Рис. 1.5. Пример общения на естественном языке в игре с компьютерной системой (с разрешения фирмы Infocom)

Обучение

Родителям ребенка, испытывающего затруднения в учебе, по собственному горькому опыту знакомо чувство растущего раздражения от недостатка у него способностей, поскольку им приходится сотни раз объяснять ему одно и то же. Однако хороший учитель обладает замечательной способностью проявлять выдержку, повторяя ребенку объяснение столько раз, сколько это нужно для того, чтобы тот освоил таблицу умножения, грамматику языка, поворот на лыжах или научился ездить на велосипеде. В ситуации, когда компьютер выступает в роли ученика, а пользователь - в роли учителя, у последнего возникает чувство досады, если при каждом запуске программы приходится всякий раз заново вводить начальные значения или воспроизводить какую-либо последовательность шагов для получения желаемого результата. Почему компьютер не может обучиться даже простейшим процедурам ?

Настоящая глава написана с помощью текстового редактора Microsoft Word. Эта программа позволяет создать множество процедур форматирования строк, абзацев и заголовков, которые впоследствии могут повторно применяться для оформления других документов, глав и даже книг. Вы можете обучить этому программу с помощью "типовых страниц". Записанные в память они вызываются путем нажатия нескольких клавиш. К тому же при работе орфографического корректора возникает впечатление, что он обладает некоторой разновидностью естественного интеллекта. Если вы допустили орфографическую ошибку, то можете дать задание программе-корректору найти слово, которое, по ее мнению, вы намеревались использовать, и устранить ошибку, обратившись к словарю. Программа может распознавать новые слова и запоминать их в электронном словаре.

Сегодня уже многие осознали, что способностью обучения должна быть наделена практически каждая прикладная программа, которая может понадобиться пользователю. Десять лет назад большая часть обработки данных в компаниях проводилась программистами вычислительных центров. Программисты фактически выполняли роль жрецов, являясь как бы связующим звеном между непостижимым компьютером и теми, кто использовал полученные данные и принимал решения.

С появлением персонального компьютера взаимоотношения между пользователем и вычислительной техникой, а следовательно, и роль программиста резко изменились. Вместо того чтобы заставлять пользователя преодолевать сложности программирования, проще обучить компьютер сложностям выполнения конкретной задачи, стоящей перед пользователем. Со временем программные продукты, такие, как электронные таблицы, системы управления базами данных и текстовые процессоры, будут расширены компонентами, обладающими свойствами систем искусственного

интеллекта (средствами обучения и общения на естественном языке, а также средствами, реализующими возможности систем, основанных на знаниях), а машинная память и аппаратура станут дешевле. Это, конечно, не означает, что необходимость в программистах отпадет, но несколько меняет их роль во взаимоотношениях между компьютером и пользователями.

Когнитивное моделирование

Целью когнитивного моделирования является разработка теории, концепций и моделей человеческого мышления и его функций. Оно помогает нам не только диагностировать и лечить психические заболевания, но и выявлять процессы, протекающие в сознании человека при решении задач. Отсюда вовсе не следует, что лучшими компьютерами являются те, которые моделируют работу человеческого мозга. Однако мы сможем сделать вывод о том, какого типа компьютеры нам нужны, как спроектировать компьютер, который бы расширил возможности мышления человека и позволил бы ему более эффективно решать задачи.

Обработка визуальной информации и робототехника

С появлением первых автоматов (XIV в.) люди были пленены идеей построения электрических и механических устройств, которые могли бы действовать подобно человеку. Вероятно, наиболее известным из ранних автоматов является искусственная утка Вокансона (1738 г.). Она могла хлопать крыльями, пить воду, клевать зерно и даже имитировать отправление естественных потребностей организма благодаря искусно сделанной системе пищеварения. Современные роботы уже облегчили труд (особенно неквалифицированный) многих рабочих, занятых в сфере производства, безупречно выполняя свою работу и не прерываясь на перекуры и чаепития. На предприятиях фирмы IBM в 60-е годы мне довелось наблюдать, как машины проектируют и строят компьютеры следующего поколения практически без участия человека. Точно так же создаются сейчас и современные компьютеры типа Macintosh и IBM PC. Исследования в области робототехники входят как составная часть в исследования по искусственному интеллекту, ставящих целью оснастить компьютеры средствами визуальной обработки и манипулирования объектами в некоторой среде. Эти исследования ведутся в трех основных направлениях:

* разработка воспринимающих элементов (в частности, для визуальной информации) и распознавание информации, поступающей от систем восприятия;

* создание манипуляторов и систем управления ими;
* выявление эвристик для решения задач перемещения в пространстве и манипулирования объектами (планирование деятельности).

В будущем роботы станут еще более интеллектуальными. В 50-е годы Айзек Азимов установил три закона робототехники, ставшие ныне классическими. Теперь они упоминаются во многих фантастических произведениях и до сих пор напоминают нам о социальных и этических проблемах, связанных с использованием роботов:

1. Робот не может причинить вред человеку или допустить это своим бездействием.

2. Робот обязан выполнять все приказания человека за исключением тех, которые противоречат первому закону.

3. Робот должен принимать все меры для самосохранения, за исключением тех случаев, когда это противоречит первому или второму закону.

СОВРЕМЕННЫЙ УРОВЕНЬ РАЗВИТИЯ ЭКСПЕРТНЫХ СИСТЕМ

Экспертные системы, реализованные на больших ЭВМ , уже сейчас не уступают по качеству вырабатываемых решений экспертам, специализирующимся в конкретных областях знаний. Фирма Digital Equipment Corporation (DEC) использует экспертные системы для прогнозирования спроса покупателей и определения конфигурации компьютерных систем по заказу пользователей. Эти экспертные системы доказали свою коммерческую эффективность, давая фирме экономию в 200000 дол. в месяц. Но еще более важным оказалось то, что применение системы повысило оперативность процедуры установки компьютера. Так, без экспертной системы отсутствие десятидолларового кабеля при установке компьютера могло надолго задержать его ввод в эксплуатацию и, как следствие, привести к неоправданно высоким расходам фирмы DEC на данном этапе работы. Управление процессом установки с помощью экспертной системы позволяет определять уникальные конфигурации компьютера и составлять для них перечень всех необходимых компонентов и технологических операций, чтобы проводить установку без задержек. Другая экспертная система (PROSPECTOR) обнаружила залежи молибдена стоимостью 100 млн. дол., наличие которых не предполагал ни один из девяти экспертов, участвующих в построении базы знаний. Система MYCIN ставит медицинские диагнозы, совпадающие с заключениями врачей-специалистов.

Большинство исследователей считают возможности персональных компьютеров недостаточными для реализации эффективных

2 Зак. № 299

экспертных систем какого бы то ни было назначения. Машина, оснащаемая средствами искусственного интеллекта, должна обладать памятью значительного объема (исчисляемого мегабайтами) и быстродействующими процессорами. При этом может возникнуть необходимость объединения нескольких процессоров в параллельные структуры. Языки программирования пока еще более пригодны для выполнения процедур числовой, а не символьной обработки, требуемой для решения большинства задач. Практически отсутствуют средства взаимодействия с прикладными программами на естественном языке, за исключением нескольких примитивных программных продуктов типа CLOUT фирмы Microrim и Savvy фирмы Excalibur. Промышленные роботы уже начали свое вторжение на предприятия, но пройдет еще немало лет, прежде чем роботы-домохозяйки станут для нас столь же привычными, как домашние животные.

Однако все это не пугает ученых. Причина их бесстрашия легко объяснима. Стремительное снижение стоимости производства и разработки новых процессоров и языков, в большей степени ориентированных на символьную обработку, вскоре сделает применение средств искусственного интеллекта экономически целесообразным для решения задач в целом ряде областей. На создание хороших экспертных систем понадобится что-нибудь около 10 человеко-лет. Другими словами, многое из того, что намечается реализовать в ближайшее десятилетие на персональных компьютерах, уже сейчас проходит апробацию в исследовательских лабораториях. Даже с помощью нынешней примитивной технологии мы можем начать работать над созданием моделей и концепций, которые вскоре обретут жизнь. Разрабатываемые сегодня теории, модели, программы и аппаратура - это большой вклад в создание систем будущего.

Примером того, как средства искусственного интеллекта могут впоследствии повлиять на разработку программного обеспечения для персональных компьютеров, является новая СУБД Paradox фирмы Ansa. Она не относится к числу систем, основанных на знаниях, однако использует методы искусственного интеллекта для контроля за выполнением неявно заданного алгоритма решения задачи. С системой Paradox может работать человек, не знакомый с базами данных, программированием или инженерией знаний. Пользователь определяет задачу путем задания примеров, после чего Paradox на основе некоторых эвристик (см. гл. 2) самостоятельно составляет процедуру ее решения.

УПРАЖНЕНИЯ

1. Повторите упоминавшееся ранее задание на образование чанков применительно к приведенному ниже предложению. Прочитайте предложение в течение 10 секунд, затем закройте книгу и воспроизведите его на бумаге.

Инженерия знаний является очень молодой наукой и пока еще не имеет общепринятых стандартов.

Теперь проведите тот же эксперимент с предложением, не имеющим смысла:

Способен время работа часто любит Сэм наука дисциплина недостаток адекватный необыкновенный цвет.

В первом случае вы, вероятно, успешно запомнили все слова, хотя в предложении было 12 чанков, что превышает объем кратковременной памяти. Почему это оказалось возможным? Чем объясните свою неудачу во втором случае, ведь предложение содержало то же число слов? В чем состоит способ образования чанков знаний у человека?

2. Что делает знания полезными и что определяет их ценность?

3. Каковы специфические функциональные особенности мышления людей, являющихся специалистами в различных областях профессиональной деятельности (в частности, занимающихся решением задач, посвятивших себя творческой деятельности, преподавателей)? В каких из указанных областей навыки можно совершенствовать путем тренировки и обучения, а в каких они зависят только от наличия способностей?

Глава 2

СИСТЕМЫ, ОСНОВАННЫЕ НА ЗНАНИЯХ

Разработка систем, основанных на знаниях, является составной частью исследований по искусственному интеллекту и имеет целью создание компьютерных методов решения проблем, обычно требующих привлечения специалистов. В этой главе вы получите начальные сведения о применении и ограничениях систем, основанных на знаниях, а также познакомитесь с основными принципами обработки символьной информации.

ЧТО ТАКОЕ СИСТЕМА, ОСНОВАННАЯ НА ЗНАНИЯХ?

Эдвард Фейгенбаум, ведущий специалист в области систем, основанных на знаниях, из Стэнфордского университета, определяет эти системы как "интеллектуальные компьютерные программы, использующие знания и процедуры вывода для решения проблем, которые настолько сложны, что для их решения необходимо привлечение эксперта". Терминология по искусственному интеллекту пока еще окончательно не установилась, поэтому словосочетания "экспертные системы" и "системы, основанные на знаниях" мы будем употреблять как синонимы. В системах, основанных на знаниях, правила (или эвристики), по которым решаются проблемы в конкретной предметной области, хранятся в базе знаний. Проблемы ставятся перед системой в виде совокупности фактов, описывающих некоторую ситуацию, и система с помощью базы знаний пытается вывести заключение из этих фактов (рис. 2.1). Эвристики представляют собой правила вывода, которые позволяют находить решения по известным фактам.

Например, при установлении медицинского диагноза о пациенте могут быть известны следующие факты: слюнные железы распухли; температура высокая; слюноотделение снижено; лимфатические узлы на шее увеличены; сосание лимона вызывает боль или

вообще невозможно. По этим фактам врач определяет, что пациент болен свинкой. Тот же самый диагноз могла бы поставить и компьютерная система, основанная на знаниях, если бы в ней хранились необходимые правила (эвристики). Можно сказать, что качество экспертной системы определяется размером и качеством базы знаний (правил или эвристик). Система функционирует в следующем циклическом режиме: выбор (запрос) данных или результатов анализов, наблюдение, интерпретация результатов, усвоение новой информации, выдвижение с помощью правил временных гипотез и затем выбор следующей порции данных или результатов анализов (рис. 2.2). Такой процесс продолжается до тех пор, пока не поступит информация, достаточная для окончательного заключения.

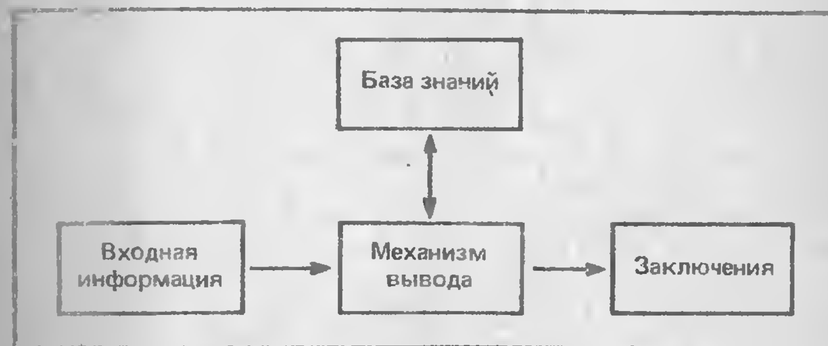


Рис. 2.1. Схема функционирования системы, основанной на знаниях

Более простые системы, основанные на знаниях, функционируют в режиме диалога, называемом *режимом консультации*. После запуска система задает пользователю ряд вопросов о решаемой задаче, требующих краткого ответа: "да" или "нет". Ответы служат для установления фактов, по которым может быть выведено окончательное заключение (рис. 2.3).

В любой момент времени в системе содержатся три типа знаний:

* *Структурированные знания* - статические знания о предметной области. После того как эти знания выявлены, они уже не изменяются.
* *Структурированные динамические знания* - изменяемые знания о предметной области. Они обновляются по мере выявления новой информации. В предыдущем примере связь между четырьмя приведенными фактами и заключением относится к структурированным динамическим знаниям.

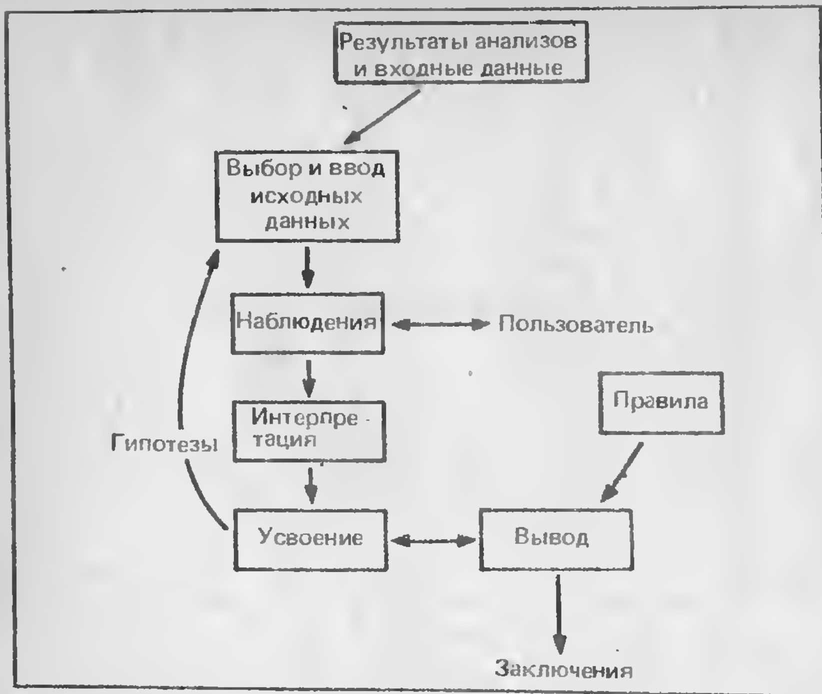


Рис. 2.2. Цикл функционирования системы, основанной на знаниях

(Начальный этап диалога, в процессе которого устанавливается имя пациента, его возраст, местонахождение очага инфекции и другие характеристики).
9. Анализ мазка проведен в то же время, когда был получен образец исследуемой бактериальной культуры?
ДА
10. Обнаружены ли какие-нибудь микроорганизмы в мазке?
НЕТ
11. Были ли отобраны для посева культуры, порождающие микроорганизмы, относительно которых Вы не будете просить совета?
ДА
12. Можно ли предположить, что очаг инфекции у пациента находится не там, откуда получены образцы культуры?
НЕТ.
13. Получает ли пациент лекарство, включающее антимикробный компонент?

Рис. 2.3. Фрагмент одного из сеансов консультации, проведенной экспертной системой

* *Рабочие знания* - знания, применяемые для решения конкретной задачи или проведения консультации. К ним относится, в частности, информация из рассмотренного выше примера о том, что у пациента жар.

Все перечисленные выше знания хранятся в базе знаний. Для ее построения требуется провести опрос специалистов, являющихся экспертами в конкретной предметной области, а затем систематизировать, организовать и снабдить эти знания указателями, чтобы впоследствии их можно было легко извлечь из базы знаний.

ОТЛИЧИТЕЛЬНЫЕ ОСОБЕННОСТИ

Системы, основанные на знаниях, имеют свои особенности, отличающие их от систем других типов.

1. Экспертиза может проводиться только в одной конкретной области. Так, программа, предназначенная для определения конфигурации систем ЭВМ, не может ставить медицинские диагнозы.

2. База знаний и механизм вывода являются различными компонентами. Действительно, часто оказывается возможным сочетать механизм вывода с другими базами знаний для создания новых экспертных систем. Например, программа анализа инфекции в крови может быть применена в пульманологии путем замены базы знаний, используемой с тем же самым механизмом вывода.

3. Наиболее подходящая область применения - решение задач дедуктивным методом. Например, правила или эвристики выражаются в виде пар посылок и заключений типа "если - то" (см. следующую главу).

4. Эти системы могут объяснять ход решения задачи понятным пользователю способом. Обычно мы не принимаем ответ эксперта, если на вопрос "Почему ?" не можем получить логичный ответ. Точно так же мы должны иметь возможность спросить систему, основанную на знаниях, как было получено конкретное заключение.

5. Выходные результаты являются качественными (а не количественными).

6. Системы, основанные на знаниях, строятся по модульному принципу, что позволяет постепенно наращивать их базы знаний.

ОБЛАСТИ ПРИМЕНЕНИЯ

Области применения систем, основанных на знаниях, могут быть сгруппированы в несколько основных классов: медицинская диагностика, прогнозирование, планирование, интерпретация, контроль и управление, диагностика неисправностей в механических и электрических устройствах, обучение.

Медицинская диагностика

Диагностические системы используются для установления связи между нарушениями деятельности организма и их возможными причинами. Наиболее известна диагностическая система MYCIN, которая предназначена для диагностики и наблюдения за состоянием больного при менингите и бактериальных инфекциях. Ее первая версия была разработана в Станфордском университете в середине 70-х годов. В настоящее время эта система ставит диагноз на уровне врача-специалиста. Она имеет расширенную базу знаний, благодаря чему может применяться и в других областях медицины.

Прогнозирование

Прогнозирующие системы предсказывают возможные результаты или события на основе данных о текущем состоянии объекта. Программная система "Завоевание Уолл-стрита" может проанализировать конъюнктуру рынка и с помощью статистических методов алгоритмов разработать для вас план капиталовложений на перспективу. Она не относится к числу систем, основанных на знаниях, поскольку использует процедуры и алгоритмы традиционного программирования. Хотя пока еще отсутствуют экспертные системы, которые способны за счет своей информации о конъюнктуре рынка помочь вам увеличить капитал, прогнозирующие системы уже сегодня могут предсказывать погоду, урожайность и поток пассажиров. Даже на персональном компьютере, установив простую систему, основанную на знаниях, вы можете получить очень точный местный прогноз погоды.

Планирование

Планирующие системы предназначены для достижения конкретных целей при решении задач с большим числом переменных. Далласская фирма Infomart впервые в торговой практике предоставляет в распоряжение покупателей 13 рабочих станций, установленных в холле своего оффиса, на которых проводятся бесплатные 15-минутные консультации с целью помочь покупателям выбрать компьютер, в наибольшей степени отвечающий их потребностям и бюджету. Эта система использует пакет программ Personal Consultant фирмы Texas Instruments и разработана отделением Вычислительной техники компании Boeing, расположенным в Сиэтле. Кроме того, компания Boeing применяет экспертные системы для проектирования космических станций, а также для выявления причин отказов самолетных двигателей и ремонта вертолетов. Экспертная система XCON, созданная фирмой DEC, служит для определения

или изменения конфигурации компьютерных систем типа VAX в соответствии с требованиями покупателя. Фирма DEC в настоящее время разрабатывает более мощную систему XSEL, включающую базу знаний системы XCON, с целью оказания помощи покупателям при выборе вычислительных систем с нужной конфигурацией. В отличие от XCON система XSEL является интерактивной.

Интерпретация

Интерпретирующие системы обладают способностью получать определенные заключения на основе результатов наблюдения. Система PROSPECTOR, одна из наиболее известных систем интерпретирующего типа, объединяет знания девяти экспертов. Используя сочетание всех девяти методов экспертизы, системе удалось обнаружить залежи руды стоимостью в миллионы долларов, причем наличие этих залежей не предполагал ни один из девяти экспертов. Другая интерпретирующая система - HASP/SIAP. Она определяет местоположение и типы судов в Тихом океане по данным акустических систем слежения.

Контроль и управление

Системы, основанные на знаниях, могут применяться в качестве интеллектуальных систем контроля и принимать решения, анализируя данные, поступающие от нескольких источников. Такие системы уже работают на атомных электростанциях, управляют воздушным движением и осуществляют медицинский контроль. Они могут быть также полезны при регулировании финансовой деятельности предприятия и оказывать помощь при выработке решений в критических ситуациях.

Диагностика неисправностей в механических и электрических устройствах

В этой сфере системы, основанные на знаниях, незаменимы как при ремонте механических и электрических машин (автомобилей, дизельных локомотивов и т.д.), так и при устранении неисправностей и ошибок в аппаратном и программном обеспечении компьютеров. В частности, несколько лет назад в фирме General Electric специалисты по ремонту дизельных локомотивов должны были выйти на пенсию. Чтобы сохранить для фирмы их опыт и знания, администрация приняла решение построить экспертную систему, в которой были бы представлены все инженерные знания, накопленные за последние годы.

Обучение

Системы, основанные на знаниях, могут входить составной частью в компьютерные системы обучения. Система получает информацию о деятельности некоторого объекта (например, студента) и анализирует его поведение. База знаний изменяется в соответствии с поведением объекта. Примером такого обучения может служить компьютерная игра, сложность которой автоматически увеличивается по мере возрастания степени квалификации играющего. Одной из наиболее интересных обучающих экспертных систем является разработанная Д. Ленатом система EURISKO, которая использует простые эвристики. Эта система была опробована в игре Т. Тревеллера, имитирующей боевые действия. Суть игры состоит в том, чтобы определить состав флотилии, способной нанести поражение противнику в условиях неизменяемого множества правил. Система EURISKO включила в состав флотилии небольшие, способные провести быструю атаку корабли и одно очень маленькое скоростное судно и постоянно выигрывала в течение трех лет, несмотря на то что в стремлении воспрепятствовать этому правила игры меняли каждый год.

Большинство экспертных систем включают знания, по содержанию которых их можно отнести одновременно к нескольким типам. Например, обучающая система может также обладать знаниями, позволяющими выполнять диагностику и планирование. Она определяет способности обучаемого по основным направлениям курса, а затем с учетом полученных данных составляет учебный план. Управляющая система может применяться для целей контроля, диагностики, прогнозирования и планирования. Система, обеспечивающая сохранность жилища, может следить за окружающей обстановкой, распознавать происходящие события (например, открылось окно), выдавать прогноз (вор взломщик намеревается проникнуть в дом) и составлять план действий (вызвать полицию).

КРИТЕРИИ ИСПОЛЬЗОВАНИЯ

Существует ряд прикладных задач, которые решаются с помощью систем, основанных на знаниях, более успешно, чем любыми другими средствами. При определении целесообразности применения таких систем нужно руководствоваться следующими критериями.

1. Данные и знания надежны и не меняются со временем.
2. Пространство (или область) возможных решений относительно невелико.
3. В процессе решения задачи должны использоваться формальные рассуждения. Существующие системы, основанные на знаниях, пока еще не пригодны для решения задач методами проведения аналогии или абстрагирования (человеческий мозг справляется с этим лучше). В свою очередь традиционные компьютерные программы оказываются эффективнее систем, основанных на знаниях, в тех случаях, когда решение задачи связано с привлечением процедурного анализа. Системы, основанные на знаниях, более подходят для решения задач, где требуются формальные рассуждения.
4. Должен быть по крайней мере один эксперт, который способен явно сформулировать свои знания и объяснить методы применения этих знаний для решения задач.

В табл. 2.1 приведены сравнительные свойства прикладных задач, по наличию которых можно судить о целесообразности использования для их решения экспертных систем.

В целом экспертные системы не рекомендуется применять для решения следующих типов задач:

* математических, решаемых обычно путем формальных преобразований и процедурного анализа;
* задач распознавания, поскольку в общем случае они решаются численными методами;
* задач, знания о методах решения которых отсутствуют (невозможно построить базу знаний).

Таблица 2.1

Критерии применимости экспертных систем

Применимы	Неприменимы
Не могут быть построены строгие алгоритмы или процедуры, но существуют эвристические методы решения	Имеются эффективные алгоритмические методы
Есть эксперты, способные решить задачу	Отсутствуют эксперты или их число недостаточно
По своему характеру задачи относятся к области диагностики, интерпретации или прогнозирования	Задачи носят вычислительный характер
Доступные данные "зашумлены"	Известны точные факты и строгие процедуры
Задачи решаются методом формальных рассуждений	Задачи решаются процедурными методами, с помощью аналогии или интуитивно
Знания статичны (неизменны)	Знания динамичны (меняются со временем)

ОГРАНИЧЕНИЯ

Даже лучшие из существующих экспертных систем, которые эффективно функционируют как на больших, так и на мини-ЭВМ, имеют определенные ограничения по сравнению с человеком - экспертом.

1. Большинство экспертных систем не вполне пригодны для применения конечным пользователем. Если вы не имеете некоторого опыта работы с такими системами, то у вас могут возникнуть серьезные трудности. Многие системы оказываются доступными только тем экспертам, которые создавали их базы знаний.

2. Вопросно-ответный режим, обычно принятый в таких системах, замедляет получение решений. Например, без системы MYCIN врач может (а часто и должен) принять решение значительно быстрее, чем с ее помощью.

3. Навыки системы не возрастают после сеанса экспертизы.

4. Все еще остается проблемой приведение знаний, полученных от эксперта, к виду, обеспечивающему их эффективную машинную реализацию.

5. Экспертные системы не способны обучаться, не обладают здравым смыслом. Домашние кошки способны обучаться даже без специальной дрессировки, ребенок в состоянии легко уяснить, что он станет мокрым, если опрокинет на себя стакан с водой, однако если начать выливать кофе на клавиатуру компьютера, у него не хватит "ума" отодвинуть ее.

6. Экспертные системы неприменимы в больших предметных областях. Их использование ограничивается предметными областями, в которых эксперт может принять решение за время от нескольких минут до нескольких часов.

7. В тех областях, где отсутствуют эксперты (например, в астрологии), применение экспертных систем оказывается невозможным.

8. Имеет смысл привлекать экспертные системы только для решения когнитивных задач. Теннис и езда на велосипеде не могут являться предметной областью для экспертных систем, однако такие системы можно использовать при формировании футбольных команд.

9. Человек-эксперт при решении задач обычно обращается к своей интуиции или здравому смыслу, если отсутствуют формальные методы решения или аналоги таких задач. Это подтверждает следующий пример. Как-то один банковский служащий был направлен в Южноамериканский банк, который просил предоставить ему заем, и посвятил несколько дней изучению его финансовых документов. По всем формальным показателям получалось, что этот банк является достаточно надежным. Однако банковский служащий, основываясь на своем опыте и интуиции, рекомендовал отказать в заеме. Через несколько недель банк "лопнул".

Системы, основанные на знаниях, оказываются неэффективными при необходимости проведения скрупулезного анализа, когда число "решений" зависит от тысяч различных возможностей и многих переменных, которые изменяются во времени. В таких случаях лучше использовать базы данных с интерфейсом на естественном языке.

ПРЕИМУЩЕСТВА

Системы, основанные на знаниях, имеют определенные преимущества перед человеком-экспертом.

1. У них нет предубеждений.

2. Они не делают поспешных выводов.

3. Эти системы работают систематизированно, рассматривая все детали, часто выбирая наилучшую альтернативу из всех возможных.

4. База знаний может быть очень и очень большой. Врач имеет ограниченную базу знаний, и если данные долгое время не используются, то они забываются и навсегда теряются. Например, сельский врач может ошибиться при распознавании конкретного заболевания ввиду его уникальности или потому, что он никогда не встречался с ним прежде. Ничего подобного не может произойти с компьютерной экспертной системой. Будучи введены в машину один раз, знания сохраняются навсегда.

5. Системы, основанные на знаниях, устойчивы к "помехам". Эксперт пользуется побочными знаниями и легко поддается влиянию внешних факторов, которые непосредственно не связаны с решаемой задачей. Экспертные системы, не обремененные знаниями из других областей, по своей природе менее подвержены "шумам". Со временем системы, основанные на знаниях, могут рассматриваться пользователями как разновидность средств тиражирования - новый способ записи и распространения знаний. Подобно другим видам компьютерных программ они не могут заменить человека в решении задач, а скорее напоминают орудия труда, которые дают ему возможность решать задачи быстрее и эффективнее. Эти системы не заменяют специалиста, а являются инструментом в его руках.

ВОЗМОЖНОСТИ РЕАЛИЗАЦИИ НА ПЕРСОНАЛЬНЫХ КОМПЬЮТЕРАХ

Системы, основанные на знаниях, как правило, требуют для своей реализации большого объема памяти ЭВМ и быстродействующих процессоров. Значительная часть памяти расходуется на хранение базы знаний и эвристик, которые используются для получения решения. Собственно же "программа" совсем невелика (в

последующих главах при непосредственном знакомстве с экспертными системами, описанными в книге, вы узнаете, что собственно программа занимает объем памяти компьютера IBM PC приблизительно в 30000 байт, т.е. намного меньший, чем редакторы, электронные таблицы или системы управления базами данных).

Однако большие объемы памяти необходимы для хранения баз знаний экспертных систем. В персональном компьютере с памятью объемом 640000 байт может храниться не более нескольких сотен правил и пары дюжин заключений. Поэтому в настоящее время такие системы (одна из них описана в нашей книге) могут применяться только для решения задач из очень небольших предметных областей, для разработки прототипов больших систем и обучения азам построения экспертных систем. Экспертные системы, реализованные на персональных компьютерах, имеют несколько возможных областей применения:

* расчет почтовых расходов и установление кратчайшего пути транспортировки почтового отправления в зависимости от его веса и пункта назначения;
* анализ альтернативных вариантов телефонной связи с целью выявления варианта с наименьшей стоимостью;
* диагностика отказов автоматики;
* анализ спроса покупателей на компьютеры и определение конфигурации систем мини-ЭВМ;
* местный прогноз погоды;
* обеспечение безопасности;
* использование солнечной энергии;
* анализ отчетов о поездках (командировках) для получения обобщенных выводов;
* анализ индивидуальных стратегий капиталовложений;
* интерпретация результатов электрофореза.

Размеры базы знаний не всегда определяют качество системы по критерию "стоимость-эффективность". Шолом М. Вейс и Казимир А. Куликовски из Рутжерского университета и Роберт С.Гален из Колумбийского университета совместно разработали в 1980 г. ориентированную на микроЭВМ систему интерпретации результатов электрофореза, которая использовала всего 82 правила. Как выяснилось, эта система дает приемлемые результаты практически во всех случаях ее применения.

Ограничения экспертных систем, реализованных на персональных компьютерах, обусловлены только размером их памяти и быстродействием процессоров и, по существу, являются временными. Сегодня рынок экспертных систем для персональных компьютеров постоянно расширяется. Уже более ста компаний, желая удовлетворить растущий спрос покупателей, включили в состав своей продукции экспертные системы. Можно провести интересную параллель между разработкой в наши дни экспертных систем для

персональных компьютеров и созданием несколько лет назад СУБД для компьютеров этого же типа. При проектировании первых СУБД для персональных компьютеров разработчики использовали принципы, положенные в основу уже существующего программного обеспечения для больших ЭВМ, и пытались перенести имеющиеся программные продукты на микрокомпьютеры. Однако все их попытки потерпели неудачу, чего нельзя сказать о СУБД нового типа dBASE II,которой даже не пытались придать стандартный вид реляционной СУБД. Многие концепции, разработанные для больших ЭВМ, оказались применимы и к микрокомпьютерам, но принципы построения и эксплуатации СУБД радикально отличаются от ранее принятых. Не происходит ли то же самое и с экспертными системами?

Одной из наиболее популярных экспертных систем для персональных компьютеров считается Expert-Easy, функционирующая по совершенно иной схеме, нежели традиционные экспертные системы. Конкурентоспособными из всех фирм-производителей экспертных систем могут стать только те, которые проявляют изобретательность в поиске новых путей систематического применения знаний и методов инженерии знаний при решении задач. Использование почти всех существующих ныне экспертных систем требует наличия определенных навыков программирования. Для того чтобы экспертные системы когда-нибудь действительно стали полезным инструментом в руках бизнесменов и специалистов из самых различных областей, необходимо развить эти системы до такого уровня, при котором возможно общение с ними на естественном языке, что сделает их доступными не только программистам, но и пользователям.

ЭВРИСТИЧЕСКИЕ И АЛГОРИТМИЧЕСКИЕ МЕТОДЫ РЕШЕНИЯ ЗАДАЧ

В настоящее время ЭВМ используются преимущественно в тех областях, где для решаемых задач существуют точно определенные методы, или *алгоритмы*. *Алгоритм* представляет собой систематическую процедуру, которая может быть применена для решения некоторой задачи. Он гарантирует правильное завершение вычислений и при повторных применениях будет давать те же результаты. Так, вычисление ежемесячных платежей за купленный в кредит автомобиль и есть тот самый алгоритм, или правильно определенная процедура. Для реализации такого алгоритма нужны некоторые входные данные: величина кредита, число месяцев, на которые он предоставлен, норма процента. На основе этих данных процедура (алгоритм) может вычислить размер месячного платежа. Одни и те же входные данные всегда приводят к одному и тому же результату.

Напротив, обнаружение неисправности в вашем автомобиле требует совсем иного подхода к решению задачи. Допустим, однажды утром ваш автомобиль не завелся - не работает стартер. Вы предполагаете, что имеются неполадки в системе электропитания, и пытаетесь найти ту часть системы, которая не работает: стартер двигателя, тяговое реле стартера, реле-регулятор или аккумулятор. Если у вас есть подозрения, что неполадки в аккумуляторе, то следует проверить его контакты, степень заряженности, а также плотность электролита. Со временем вы обнаружите истинную причину неисправности.

При решении этой проблемы вы опирались на *эвристики*, т.е. применяли приемы, которые повышают эффективность процесса решения задачи, даже если их природа не может быть строго объяснена. Эвристики уменьшают время решения задачи путем сокращения числа переборов в заданном пространстве поиска. Так, в примере с автомобилем нет необходимости проверять уровень охлаждающей жидкости в радиаторе, поскольку уже на первых шагах выяснилось, что неполадки - в системе электропитания стартера. Хотя эвристики не гарантируют нахождение решения, они позволяют сократить время, требуемое для решения очень сложных задач. Их можно сравнить с фильтрами в процессе сопоставления по образцу, которые дают возможность системе сфокусировать внимание лишь на нескольких основных образцах.

Задача обнаружения неполадок в автомобиле не может быть решена с помощью процедурных методов, как это принято в традиционном программировании. По своей природе она существенно отличается от задачи вычисления ежемесячных платежей за купленный в кредит автомобиль. Задача обнаружения неисправностей довольно плохо структурирована, и поэтому практически невозможно построить какую-либо процедуру, гарантирующую ее решение. Мы можем рассматривать эту задачу как поиск в пространстве состояний, где каждое состояние может стать частичным решением, т.е. составляющей конечного решения. Осуществляя поиск, мы надеемся, что приближаемся к конечной цели, хотя твердо уверенными в этом быть не можем. Однако можно сформировать такие эвристики, которые помогут нам минимизировать число переборов в пространстве состояний и в случае удачи приблизить нас к цели - запуску двигателя автомобиля.

На ранних этапах большинство исследований по искусственному интеллекту было направлено на построение эвристик для решения задач, что привело к тупиковой ситуации. В настоящее время акцент в этой области сместился. Сегодня исследования ориентированы на разработку способов определения и организации (построения чанков) знаний в структуры, используемые для решения плохо структурированных проблем, которые не под силу даже специалистам.

СИМВОЛЬНАЯ И ЧИСЛОВАЯ ОБРАБОТКА ДАННЫХ

Почти все современные компьютеры осуществляют числовую обработку данных, и по таким параметрам, как надежность и скорость вычислений, они делают это лучше, чем человек. И наоборот, человеческий мозг более адаптирован к символьной обработке, чем мощнейшие компьютеры. Шахматист, играя партию, использует символьную обработку. Возможности для хорошего хода повышаются, если рассматривать расположение фигур на доске как совокупность типовых позиций, или чанков. Шахматист сравнивает совокупность позиций с образами, хранящимися в его долговременной памяти. Его мысль направлена на создание ситуации, в которой он может поставить мат противнику. Компьютер, напротив, медлителен и неэффективен в символьной обработке, поскольку выиграть партию он может, только перебрав практически все возможные ходы (здесь он более искусен, чем человек). В своем стремлении к победе компьютер применяет некоторые эвристики, которые позволяют ему минимизировать число ходов, подлежащих просмотру.

Создание экспертных систем можно рассматривать как попытку наделить компьютер способностью выполнять символьную обработку, аналогичную той, которую выполняет человек, и тем самым поднять компьютер в пределах ограниченной предметной области до уровня эксперта. Под *предметной областью* здесь следует понимать совокупность взаимосвязанных знаний. Различают *прескриптивные* (предписывающие) и *дескриптивные* (описательные) системы. Шахматная программа является прескриптивной, поскольку использует предварительно заданный алгоритм и процедуру вычисления оценки любого возможного хода. На каждом шаге она должна выбирать ход с наибольшей оценкой, опираясь на эвристики, чтобы уменьшить число ходов, для которых необходимо вычислять оценки. В дескриптивной системе, напротив, процесс решения задачи управляется не процедурой, а знаниями, хранимыми в базе знаний. Процедура фактически существует, но задана неявно. Принципы функционирования таких систем подробно описаны в следующей главе. Большинство языков программирования, например Бейсик, Паскаль, Си, Фортран, ориентированы на поддержку прескриптивных процессов решения задач. Как вы увидите в гл.6 и 10, для поддержки дескриптивных процессов решения необходимы специальные языки (и элементы аппаратуры).

Символ можно определить как некоторый компонент структуры знаний. Примеры символов - четыре, Джек, Спарроу, 3.1416, дочь. Одной из важных предпосылок любого исследования по искусственному интеллекту является разработка формальных языков для описания символьных преобразований. В гл.4 мы покажем вам,

как такие формальные языки могут быть использованы для представления знаний.

РАССУЖДЕНИЯ С РАСШИРЯЮЩИМСЯ И УМЕНЬШАЮЩИМСЯ МНОЖЕСТВОМ ЗАКЛЮЧЕНИЙ

Традиционные вычислительные системы хорошо приспособлены для выполнения рассуждений с расширяющимся множеством заключений. При небольшом числе входных данных эти компьютеры могут породить практически бесконечное число выходных данных (результатов). Человеку же и системам, основанным на знаниях, более свойственны рассуждения с уменьшающимся множеством заключений, когда он получает единичные результаты на основе больших массивов данных. Например, ученый может затратить годы на анализ данных о каком-либо явлении для получения результата, который будет удостоен Нобелевской премии.

Таблица 2.2

Сравнение свойств обычных ЭВМ и компьютеров для реализации экспертных систем

Свойства обычных компьютеров	Свойства компьютеров для экспертных систем
Процедурно управляемый поток вычислений	Поток вычислений, управляемый данными
Числовая обработка	Символьная обработка
Рассуждения с расширяющимся множеством выводов	Рассуждения с уменьшающимся множеством выводов
Фон-неймановская архитектура	Архитектура, отличная от фон-неймановской
Алгоритмическая обработка	Эвристическая обработка
Обслуживание программистами	Обслуживание инженерами знаний
Последовательная обработка	Интерактивная и параллельная обработка
Структурированный процесс проектирования (линейный)	Интерактивный процесс проектирования (циклический)

ВЫВОДЫ

Системы, основанные на знаниях, представляют собой особый тип компьютерных систем, которые во многом отличаются от традиционных. Они используют декларативно заданные алгоритмы, символьную обработку и рассуждения с уменьшающимся множеством выводов (табл. 2.2). Новая архитектура компьютеров, подходящая для реализации таких систем, еще только начинает создаваться, однако уже сейчас ясно, что она должна быть ориентирована на параллельную , а не на последовательную обработку, которую выполняют существующие машины. Значительным вкладом в развитие экспертных систем будет разработка выразительных способов представления знаний и более глубокое изучение механизмов решения задач, особенно тех из них, которые неточно определены и плохо структурированы.

УПРАЖНЕНИЯ

1. Для каждой из перечисленных ниже прикладных областей укажите, какие способы решения задач окажутся наиболее эффективными - с помощью систем, основанных на знаниях, или традиционное программирование, и почему:

а) управление предприятием;
б) ремонт ЭВМ;
в) классификация видов в биологии (птиц, цветов и т.д.);
г) составление меню;
д) объяснение результатов химического анализа крови при внутренних заболеваниях;
е) планирование учебных программ;
ж) анализ возможных вариантов капиталовложений (финансовое планирование).

2. Назовите известную вам прикладную задачу, которая в настоящее время требует привлечения специалистов. Могут ли специалисты использовать для решения этой задачи экспертную систему? Поясните свой ответ.

3. Представьте прикладную область, в которой существующие экспертные знания постоянно увеличиваются, что со временем делает невозможной их эффективную обработку человеком. Что произойдет в таком случае с накопленными знаниями и как их применить?

4. Какая из выполняемых вами в настоящее время задач, которая сегодня вам по силам, несколько десятков лет назад потребовала бы привлечения специалиста?

Глава 3

СТРУКТУРА СИСТЕМ, ОСНОВАННЫХ НА ЗНАНИЯХ

В настоящей главе рассматривается конкретный тип систем, основанных на знаниях, - *продукционные системы*. Их также иногда называют системами, основанными на правилах, или системами с выводом, использующим сопоставление по образцу. Здесь описываются компоненты этих систем и схемы их функционирования. В заключение главы дается краткий обзор некоторых промышленных систем, разработанных с помощью такой модели представления знаний. В следующей главе мы обсудим другие типы систем, основанных на знаниях.

ПРОДУКЦИОННЫЕ СИСТЕМЫ

Вычислительный процесс в продукционных системах существенно отличается от классической фон-неймановской схемы, принятой в большинстве компьютеров. В компьютерах с фон-неймановской архитектурой и программы, и данные при вычислениях находятся вместе в одной области памяти. Программа представляет собой последовательность действий, записанных в памяти, по преобразованию данных для получения требуемого результата. Программа может даже преобразовывать самою себя как данные, изменяя тем самым последовательность своих действий. В противоположность этому продукционная система почти не имеет процедурных компонентов и практически полностью управляется данными, т.е. является дескриптивной. Продукционная система включает три основных составляющих: базу правил, рабочую память и механизм вывода. В дополнение к ним для поддержки работы системы и реализации интеллектуального взаимодействия с пользователем в нее обычно входят еще и подсистема приобретения знаний, средства общения на естественном языке, а также подсистема объяснения (рис.3.1).

Рис. 3.1. Структура экспертной системы продукционного типа

Память для хранения правил (база правил)

Если вы когда-нибудь наблюдали за тем, как подходит к решению задачи поиска неисправности в автомобиле профессионал (здесь мы возвращаемся к примеру из предыдущей главы), то заметили, что для ее решения он использует определенную стратегию или эвристики. Даже плохо представляя себе истинную причину неполадок, он все равно осуществляет поиск вполне целенаправленно и не производит впечатления человека, который не знает, что предпринять в конкретной ситуации.

Обнаружив, что анализируемая система (в данном случае автомобиль) неисправна, специалист пытается определить ту ее подсистему, в которой могут быть неполадки (подсистема электропитания, охлаждения, подачи топлива и т.д.). После выявления такой подсистемы он приступает к поиску неисправных компонентов (рис. 3.2). Этот процесс выглядит так, как будто специалист постоянно обращается к неявно заданному множеству правил, получая промежуточные заключения при выполнении каких-либо условий. Промежуточные заключения в свою очередь становятся условиями для вывода следующих заключений.

Например, в рассматриваемом случае (с упрощенными правилами) цепь рассуждений будет выглядеть приблизительно так:

ЕСЛИ	двигатель не заводится
И	стартер двигателя не работает
ТО	неполадки - в системе электропитания стартера
ЕСЛИ	двигатель не заводится
И	стартер двигателя не работает
ТО	неполадки - в системе подачи топлива
ЕСЛИ	неполадки в системе подачи топлива
И	показатель уровня топлива находится на нуле
ВЫВОД	газовая камера пуста
ЕСЛИ	неполадки в системе электропитания стартера
И	нарушены контакты аккумулятора
ВЫВОД	плохо присоединен аккумулятор

Таким образом, наша система, основанная на знаниях, в качестве одного из компонентов должна включать *память для хранения правил*, которая содержит набор срабатывающих в определенных ситуациях правил, имеющих форму ЕСЛИ-ТО. Такие конструкции получили название *продукционных правил*. Каждое правило складывается из двух частей. Первая из них - *антецедент*, или *посылка правил*, - состоит из элементарных предложений, соединенных логическими связками И, ИЛИ и т.д. Вторая часть, называемая *консеквентом*, или *заключением*, состоит из одного или нескольких предложений, которые образуют выдаваемое правилом решение либо указывают на действие, подлежащее выполнению (рис. 3.3). Антецедент представляет собой *образец правила*, предназначенного для распознавания ситуации, когда оно должно *сработать*. Правило срабатывает, если факты из рабочей памяти (см. следующий раздел) при сопоставлении совпали с образцом, после чего правило считается отработавшим.

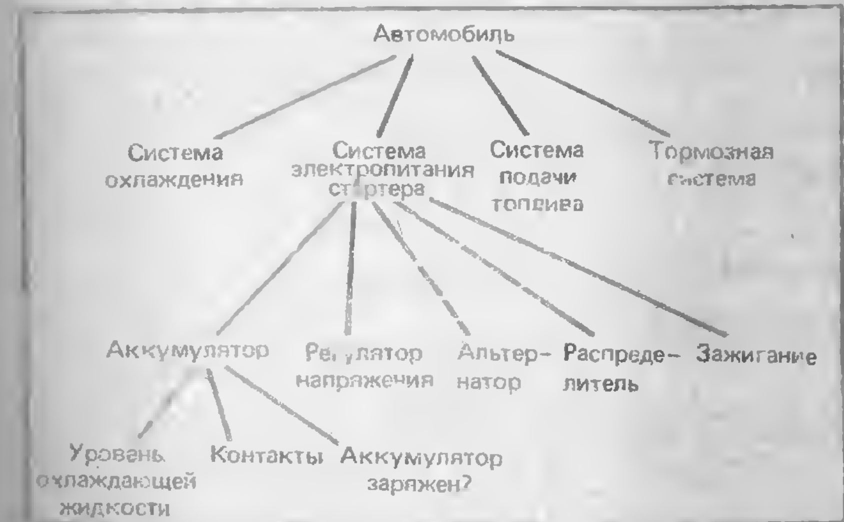


Рис. 3.2 Иерархическая схема поиска неисправностей в автомобиле

ЕСЛИ	двигатель не заводится	антецедент
И	стартер двигателя не работает	
ТО	неполадки - в системе электропитания стартера	консеквент

Рис. 3.3. Пример продукционного правила

Каждое правило в этом простом примере содержит *атрибуты* и *значения*. Так, в первом правиле выражение "не заводится" является значением, а слово "двигатель" - атрибутом (рис. 3.4). Любое правило состоит из одной или нескольких пар атрибут - значение и заключения.

Продукционную модель, послужившую нам для построения базы правил, по сути можно рассматривать как вариант модели "стимул-реакция", которая была предложена в психологии и теории автоматов задолго до появления экспертных систем. Многие ученые полагают, что эта модель вполне адекватно описывает поведение человека.

Атрибут	Значение
Двигатель	Не заводится
Стартер двигателя	Не работает
Система электропитания стартера	Неисправна

Рис. 3.4. Пары атрибут-значение

Рабочая память (база данных)

Другая важная часть системы, основанной на знаниях, - рабочая память (база данных). В этой памяти хранятся множество фактов,описывающих текущую ситуацию, и все пары атрибут-значение, которые были установлены к определенному моменту. Содержимое рабочей памяти со временем изменяется, увеличиваясь в объеме по мере срабатывания правил[1]. В приведенном выше примере до начала процесса вывода в рабочей памяти находились только следующие факты: двигатель не заводится; стартер двигателя не работает. После применения первого правила в рабочую память добавится новый факт: система электропитания стартера неисправна. В конце концов будет выведено окончательное решение, которое также заносится в рабочую память. Последняя представляет собой динамическую часть базы знаний,изменяющуюся со временем. Ее содержимое зависит и от состояния окружающей среды.

[1] Вообще говоря, объем рабочей памяти может и уменьшаться. Это происходит в том случае, если действие правила состоит в удалении каких-либо фактов. - *Прим. перев.*

Новые факты, добавляемые в рабочую память, являются результатом вывода, который состоит в применении правил к имеющимся фактам. База знаний включает в себя совокупность правил и содержимое рабочей памяти.

В системах с монотонным выводом факты, хранимые в рабочей памяти, статичны, т.е. не изменяются в процессе решения задачи. В системах с немонотонным выводом допускается изменение или удаление фактов из рабочей памяти. В качестве примера системы с немонотонным выводом можно привести экспертную систему, предназначенную для составления перспективного плана капиталовложений компании. В такой системе по вашему желанию могут быть изменены даже те данные, которые после ввода уже вызвали срабатывание каких-либо правил. Иными словами, имеется возможность модифицировать значения атрибутов в составе фактов, находящихся в рабочей памяти. Изменение фактов в свою очередь приводит к необходимости удаления из рабочей памяти заключений, полученных с помощью упомянутых правил. Тем самым вывод выполняется повторно для того, чтобы пересмотреть те решения, которые были получены на основе подвергшихся изменению фактов.

Механизм вывода (интерпретатор правил)

Механизм вывода (интерпретатор правил) выполняет две функции: во-первых, просмотр существующих фактов из рабочей памяти и правил из базы знаний и добавление(по мере возможности) в рабочую память новых фактов и, во-вторых, определение порядка просмотра и применения правил. Этот механизм управляет процессом консультации, сохраняя для пользователя информацию о полученных заключениях, и запрашивает у него информацию, когда для срабатывания очередного правила в рабочей памяти оказывается недостаточно данных.

В некоторых системах принят прямой порядок вывода - от фактов, которые находятся в рабочей памяти, к заключению. В других системах вывод осуществляется в обратном порядке: заключения просматриваются последовательно до тех пор, пока не будут обнаружены в рабочей памяти или получены от пользователя факты, подтверждающие одно из них. В подавляющем большинстве систем, основанных на знаниях, механизм вывода представляет собой небольшую по объему программу. Основную же часть памяти компьютера занимают правила.

Выше уже отмечалось, что механизм вывода включает в себя два компонента - один из них реализует собственно вывод, другой управляет этим процессом. Компонент вывода выполняет первую задачу, просматривая имеющиеся правила и факты из рабочей памяти и добавляя в последнюю новые факты при срабатывании какого-нибудь правила. Управляющий компонент определяет порядок

применения правил. Рассмотрим каждый из этих компонентов более подробно.

Компонент вывода. Его действие основано на применении правила вывода, обычно называемого модус поненс, суть которого состоит в следующем: пусть известно, что истинно утверждение А и существует правило вида "ЕСЛИ А, ТО В", тогда утверждение В также истинно. Правила срабатывают, когда находятся факты, удовлетворяющие их левой части: если истинна посылка, то должно быть истинно и заключение.

Хотя в принципе на первый взгляд кажется, что такой вывод легко может быть реализован на компьютере, тем не менее на практике человеческий мозг все равно оказывается более эффективным при решении задач. Рассмотрим, например, простое предложение:

Мэри искала ключ.

Здесь для слова "ключ" допустимы как минимум два значения: "родник" и "ключ от квартиры". В следующих же двух предложениях одно и то же слово имеет совершенно разные значения:

Мы заблудились в чаще.
Нужно чаще ходить в театры.

Понять факты становится еще сложнее, если они являются составными частями продукций, которые используют правило модус поненс для вывода заключений. Приведем такой пример:

ЕСЛИ Белый автомобиль легко заметить ночью
И Автомобиль Джека белый
ТО Автомобиль Джека легко заметить ночью

Это заключение легко выведет даже ребенок, но оно оказывается не под силу ни одной из современных экспертных систем, как, впрочем, и какому бы то ни было животному. Другой пример:

ЕСЛИ Сюзанна была в ресторане Сильвии
И Заказала там бифштекс
И Заплатила официанту за бифштекс
ТО Сюзанна съела бифштекс в ресторане Сильвии

И в этом случае утверждение, содержащееся в заключительной части приведенной продукции, очевидно для любого человека, но вывод его без дополнительного правила находится за пределами возможностей существующих систем, основанных на знаниях. Подводя итоги, можно сказать, что человек способен вывести большое число заключений с помощью очень большой базы знаний, которая хранится в его памяти; экспертные же системы могут вывести только сравнительно небольшое число заключений, используя заданное множество правил.

Компонент вывода должен обладать способностью функционировать в условиях недостатка информации. В нашем примере с поиском неисправности в автомобиле факт нарушения контактов аккумулятора мог и отсутствовать в рабочей памяти системы в начале сеанса консультации. Конечно, компонент вывода мог бы обратиться к пользователю за сведениями о состоянии контактов. Ну, а если и ему она неизвестна? Механизм вывода должен быть способен продолжить рассуждения и со временем найти решение даже при недостатке информации. Это решение может и не быть точным, однако система ни в коем случае не должна останавливаться из-за того, что отсутствует какая-либо часть входной информации (как, например, обычная программа, определяющая целесообразность предоставления займа).

Управляющий компонент. Этот компонент определяет порядок применения правил, а также устанавливает, имеются ли еще факты, которые могут быть изменены в случае продолжения консультации (вспомните обсуждение монотонного и немонотонного выводов). Управляющий компонент выполняет четыре функции:

1.Сопоставление - образец правила сопоставляется с имеющимися фактами.

2.Выбор - если в конкретной ситуации могут быть применены сразу несколько правил, то из них выбирается одно, наиболее подходящее по заданному критерию (разрешение конфликта).

3.Срабатывание - если образец правила при сопоставлении совпал с какими-либо фактами из рабочей памяти, то правило срабатывает.

4.Действие - рабочая память подвергается изменению путем добавления в нее заключения сработавшего правила. Если в правой части правила содержится указание на какое-либо действие, то оно выполняется (как, например, в системах обеспечения безопасности информации).

Интерпретатор продукций работает циклически. В каждом цикле он просматривает все правила, чтобы выявить среди них те, посылки которых совпадают с известными на данный момент фактами из рабочей памяти. Интерпретатор определяет также порядок применения правил. После выбора правило срабатывает, его заключение заносится в рабочую память, и затем цикл повторяется сначала.

В одном цикле может сработать только одно правило. Если несколько правил успешно сопоставлены с фактами, то интерпретатор производит выбор по определенному критерию единственного правила, которое и сработает в данном цикле. Цикл работы интерпретатора схематически представлен на рис. 3.5. Информация из рабочей памяти последовательно сопоставляется с посылками правил для выявления успешного сопоставления. Совокупность отобранных правил составляет так называемое конфликтное множество. Для разрешения конфликта интерпретатор имеет критерий, с

помощью которого он выбирает единственное правило, после чего оно срабатывает. Это выражается в занесении фактов, образующих заключение правила, в рабочую память или в изменении критерия выбора конфликтующих правил. Если же в заключение правила входит название какого-нибудь действия, то оно выполняется (например, подается звуковой сигнал, начинает выполняться процедура, запускается двигатель и т.д.).

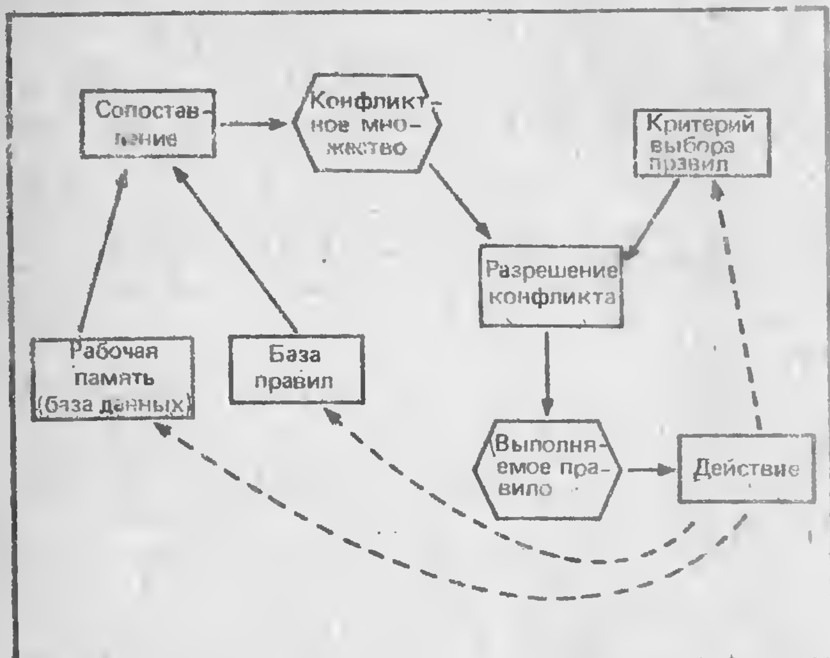


Рис. 3.5. Цикл работы интерпретатора

Новые данные, введенные в систему сработавшим правилом, в свою очередь могут изменить критерий выбора правила. В том случае, если, например, компьютерная система, предназначенная для игры в шахматы, разыгрывает партию за двух игроков, то она может принять решение придерживаться атакующей стратегии через ход, т.е. атаковать будет один из партнеров. Если вы сами играете с этой системой, то в какой-то момент она может перейти к использованию оборонительной стратегии (по крайней мере временно), а затем опять вернуться к наступательной игре. Изменение критерия основывается на заключениях, полученных после анализа положения на доске, которое представлено в рабочей памяти системы, а также правил игры (статических структурных знаний) и структурных динамических знаниях (эвристиках).

В действительности экспертные системы не располагают процедурами, которые могли бы построить в пространстве состояний сразу весь путь решения задачи. Более того, зачастую даже не удается определить, имеется ли вообще какое-нибудь решение задачи. Тем не менее поиск решения выполняется, поскольку движением в пространстве состояний управляют скрытые или виртуальные процедуры. Они получили название *демонов*, поскольку во время работы системы находятся в "засаде" и активизируются только тогда, когда их просят о помощи, т.е. на самом деле ведут себя как добрые демоны.

Свое название демоны получили от "демона Максвелла" - действующего лица одного из мысленных экспериментов, предложенного его автором для критики законов термодинамики. Другим их прообразом является Пандемониум Оливера Селфриджа - первой модели человека, в которой деятельность биологической системы представлялась как работа вызываемых по образцу демонов. Если же воспользоваться научной терминологией, то такие управляющие процедуры получили название *недетерминированных*. Это означает, что траектория поиска решения в пространстве состояний полностью определяется данными.

При разработке управляющего компонента механизма вывода необходимо решить вопрос о том, по какому критерию следует выбирать правило, которое будет применено в конкретном цикле. Этот вопрос мы рассмотрим позднее в разделе, посвященном стратегиям управления. В системах, в которых знания представлены в виде фреймов (см. гл. 4), управление осуществляется демонами, входящими в состав слотов; с принципами их работы мы познакомим вас при описании модели представления знаний, использующей фреймы.

Подсистема приобретения знаний

Подсистема приобретения знаний предназначена для добавления в базу знаний новых правил и модификации имеющихся. В ее задачу входит приведение правила к виду, позволяющему механизму вывода применять это правило в процессе работы. В простейшем случае в качестве такой подсистемы может выступать обычный редактор или текстовый процессор, который просто заносит правила в файл. В более сложных системах предусмотрены еще и средства для проверки вводимых или модифицируемых правил на непротиворечивость с имеющимися правилами.

В некоторых системах приобретение знаний осуществляется не одним, а несколькими различными методами. Примером может служить экспертная система DETEKTR, предназначенная для поиска неисправностей в электронных устройствах. Эта система разработана фирмой Tektronix в рамках программы по созданию средств инженерии знаний.

Специалисты по искусственному интеллекту фирмы Tektronix различают три типа знаний, приобретение которых соответственно требует применения трех различных методов:

* *операциональные*, необходимые для решения задачи в текущий момент;
* *вспомогательные*, или те факты, которые эксперт может и не помнить, но обязан знать, где их можно найти;
* *дополнительные*, т.е. сведения о внешних условиях, в которых будет происходить решение задачи.

В версии-прототипе системы DETEKTR операциональные знания вводятся с помощью программного средства INKA (INteractive Knowledge Acquisition - интерактивное приобретение знаний) на основе специализированной "грамматики описания знаний". Эта грамматика задаст форму представления правил поиска неисправностей для их ввода в систему экспертом, например

ЕСЛИ НАПРЯЖЕНИЕ В УЗЛЕ 4 РАВНО НАПРЯЖЕНИЮ В УЗЛЕ 5, ТО РЕЗИСТОР 2 НЕ РАБОТАЕТ.

Для поиска неполадок в электронных устройствах наиболее важными вспомогательными знаниями являются те, которые используются для проверки допустимости конкретного соединения элементов, правильности принципиальных схем и расположения плат. DETEKTR включает в себя специальную подсистему PIKA (PIctorial Knowledge Acquisition - приобретение знаний в форме изображений), которая способна воспринимать схему в графическом представлении и проверять по ней правильность соединения элементов.

Сведения о внешних условиях, в которых будет протекать процесс решения задачи (так называемые дополнительные знания), непосредственно в систему не включаются, но для того, чтобы спроектировать и реализовать эффективно работающую экспертную систему, их необходимо тщательно проанализировать. Дополнительные знания крайне необходимы, например, специалистам по ремонту, которые в случае недостатка таких знаний тратят очень много времени на поиск неисправного элемента. Дополнительные знания этого типа используются для обоснования и проверки различных проектных решений, например, при включении в электронное устройство специальных плат и схем, которые бы ускорили обнаружение неисправностей.

Средства общения на естественном языке

Поскольку системы, основанные на знаниях, реализуются на компьютерах, то и входная информация воспринимается ими в виде, понятном компьютеру, т. е. в битах и байтах. Однако для того чтобы с системой мог взаимодействовать неподготовленный пользо-

ватель, в нее требуется включить средства общения на естественном языке. Подавляющее большинство систем, основанных на знаниях, обладают достаточно примитивным интерфейсом на естественном языке - допустимые входные сообщения пользователя ограничены набором понятий, содержащихся в базе знаний. Со временем специалистам будет предоставлена возможность общаться с такими системами с помощью полных предложений, которые могут включать в себя любые части речи: существительные, глаголы, предлоги, прилагательные и наречия.

Простые системы, основанные на знаниях (такие, как продукционная система, описанная в книге), ведут с пользователем достаточно элементарный диалог, в котором он может обойтись словами "да", "нет" и иногда добавить вопрос "почему?". В более сложных системах компонент взаимодействия с пользователем способен произвести грамматический разбор входного предложения. Грамматический разбор состоит в определении связанных частей предложения. Тщательно разработанные средства общения на естественном языке, аналогичные тем, которые применяются в играх фирмы Infocom, строятся на сложном алгоритме грамматического разбора, который дает пользователю возможность вводить в систему сообщения в виде полных предложений. Алгоритм грамматического разбора определяет во введенном предложении подлежащее, сказуемое и другие члены предложения.

Подсистема объяснения

Большинство специалистов-пользователей не смогут с доверием относиться к выведенному системой заключению, пока не будут знать, как оно было получено. Если врач установил у вас наличие некоторого заболевания, то вы, конечно, захотите узнать, почему он пришел к такому выводу. Вы, вероятно, попросите его показать вам рентгеновский снимок, результаты анализа крови или что-то другое, на основе чего он сделал такое заключение, а может быть, изложить ход своих рассуждений, который позволил поставить данный диагноз. К экспертной системе предъявляются те же самые требования.

Пусть, например, в своей деятельности врач опирается на компьютерную систему, которая выполняет химический анализ крови и ставит с его помощью диагноз. И пусть на основе анализа крови пациента было установлено, что он страдает заболеванием печени и у него плохо усваиваются препараты железа. Естественно, пациент захочет узнать, почему система пришла к такому заключению.

Компонент экспертной системы, который отвечает на вопросы пользователя о том, как именно получено решение, называется подсистемой объяснения. Во время проведения консультации эта подсистема должна быть способна в любой момент привести обос-

нование принятого решения. Например, если систему, основанную на знаниях, спросить о том, почему она сделала вывод о наличии у пациента заболевания печени, то можно получить следующее объяснение.

Известны следующие факты:

Уровень содержания в крови альбумина высок, что свидетельствует об имеющемся обезвоживании организма или печеночной недостаточности.

Уровень содержания в крови билирубина низок - это указывает на нарушение функций печени или сильное утомление пациента.

Высокий уровень содержания креатинина дает основания полагать, что имеет место заболевание печени или почек, если только пациент не является спортсменом.

Пациент спортом не занимается.

Отсюда можно заключить, что пациент страдает печеночной недостаточностью.

Резюмируя изложенное выше, отметим, что продукционная система или система, основанная на правилах, работает циклически. В каждом цикле продукции (правила) из базы знаний просматриваются интерпретатором правил в определенном порядке, который устанавливается его управляющим компонентом. Если обнаруживается правило, посылка которого при сопоставлении совпала с некоторыми фактами из рабочей памяти, то правило срабатывает и его заключение добавляется в рабочую память. Затем цикл повторяется. Цикл имеет четыре фазы: *сопоставление, выбор (разрешение конфликта), срабатывание и выполнение действия (изменением состояния рабочей памяти).*

Любое правило представляет собой управляемый данными модуль (сокращенно УДМ). УДМ является автономным, не зависимым от других правил элементом базы знаний. Это дает возможность легко наращивать и отлаживать программу, состоящую из продукций, поскольку при добавлении нового правила или редактирования имеющегося не требуется вносить изменения в остальные правила. Каждому УДМ присваивается собственный номер (например, Правило-06), что позволяет весьма просто вызывать для просмотра цепочку сработавших правил. Система, основанная на правилах, содержит совокупность УДМ, состоящих из двух частей - посылки (антецедента) и заключения (консеквента). Механизм вывода может быть назван *системой вывода, использующей сопоставление по образцу.* Продукционная система - это система, основанная на правилах, механизм вывода которой включает в себя компоненты собственно вывода и управления им.

СТРАТЕГИИ УПРАВЛЕНИЯ ВЫВОДОМ

Одним из важных вопросов, возникающих при проектировании управляющей компоненты систем, основанных на знаниях, является выбор метода поиска решения, т.е. стратегии вывода. От выбранного метода поиска будет зависеть порядок применения и срабатывания правил. Процедура выбора сводится к определению направления поиска и способа его осуществления. Процедуры, реализующие поиск, обычно "зашиты" в механизм вывода, поэтому в большинстве систем инженеры знаний не имеют к ним доступа и, следовательно, не могут в них ничего изменять по своему желанию.

При разработке стратегии управления выводом необходимо ответить на два вопроса:

1. Какую точку в пространстве состояний принять в качестве исходной? Дело в том, что еще до начала поиска решения система, основанная на знаниях, должна каким-то образом выбрать исходную точку поиска. От выбора этой точки зависит и метод осуществления поиска - в прямом или в обратном направлении.

2. Как повысить эффективность поиска решения? Чтобы добиться повышения эффективности поиска решения, необходимо найти эвристики разрешения конфликтов, связанных с существованием нескольких возможных путей для продолжения поиска в пространстве состояний, поскольку требуется отбросить те из них, которые заведомо не ведут к искомому решению.

Прямой и обратный вывод

В системах с обратным выводом вначале выдвигается некоторая гипотеза, а затем механизм вывода в процессе работы как бы возвращается назад, переходя от нее к фактам, и пытается найти среди них те, которые подтверждают эту гипотезу (рис. 3.6). Если она оказалась правильной, то выбирается следующая гипотеза, детализирующая первую и являющаяся по отношению к ней подцелью. Далее отыскиваются факты, подтверждающие истинность подчиненной гипотезы. Вывод такого типа называется *управляемым целями, или управляемым консеквентами.* Обратный поиск применяется в тех случаях, когда цели известны и их сравнительно немного.

В системах с прямым выводом по известным фактам отыскивается заключение, которое из этих фактов следует (см.рис. 3.6). Если такое заключение удается найти, то оно заносится в рабочую память. Прямой вывод часто называют *выводом, управляемым данными, или выводом, управляемым антецедентами.*

В системах диагностики чаще применяется прямой вывод, в то время как в планирующих системах более эффективным оказыва-

3 Заказ № 299

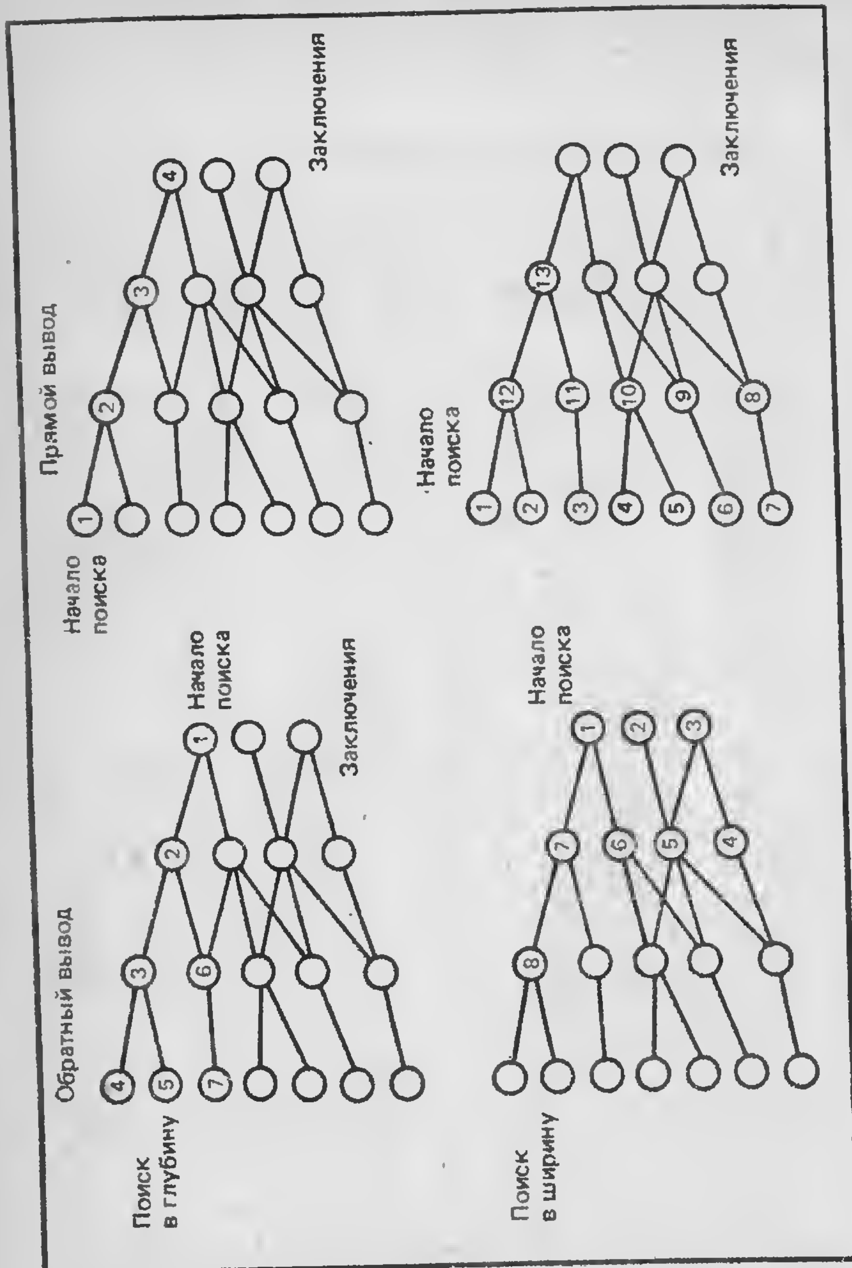


Рис. 3.6. Стратегии вывода

ется обратный вывод. В некоторых системах вывод основывается на сочетании упомянутых выше методов - обратного и ограниченного прямого. Такой комбинированный метод получил название *циклического.*

Повышение эффективности поиска

В системах, база знаний которых насчитывает сотни правил, весьма желательным является использование какой-либо стратегии управления выводом, позволяющей минимизировать время поиска решения и тем самым повысить эффективность вывода. К числу таких стратегий относятся поиск в глубину, поиск в ширину, разбиение на подзадачи и альфа-бета алгоритм.

Сопоставление методов поиска в глубину и в ширину. Суть поиска в глубину состоит в том, что при выборе очередной подцели в пространстве состояний предпочтение всегда, когда это возможно, отдается той, которая соответствует следующему, более детальному уровню описания задачи. Например, диагностирующая система, сделав на основе известных симптомов предположение о наличии определенного заболевания, будет продолжать запрашивать уточняющие признаки и симптомы этой болезни до тех пор, пока полностью не отвергнет выдвинутую гипотезу. При поиске в ширину, напротив, система вначале проанализирует все симптомы, находящиеся на одном уровне пространства состояний, даже если они относятся к разным заболеваниям, и лишь затем перейдет к симптомам следующего уровня детальности.

Специалисты в какой-либо узкой области выше оценивают поиск в глубину, поскольку он позволяет собрать воедино все признаки, связанные с выдвинутой гипотезой. Универсалы же отдают предпочтение поиску в ширину, так как в этом случае анализ не ограничивается заранее очерченным кругом признаков. Особенности пространства поиска во многом определяют целесообразность применения той или иной стратегии: например, программы для игры в шахматы строятся на основе поиска в ширину, поскольку при использовании поиска в глубину число анализируемых ходов может быть очень и очень большим.

Разбиение на подзадачи. При такой стратегии в исходной задаче выделяются подзадачи, решение которых рассматривается как достижение промежуточных целей на пути к конечной цели. Примером, прекрасно подтверждающим эффективность разбиения на подзадачи, может служить его реализация в системе XCON, о чем пойдет речь в гл. 5. Другой практической задачей, где эта стратегия себя хорошо зарекомендовала, является поиск неисправностей в автомобиле - вначале выявляется отказавшая подсистема (электропитания, охлаждения и т.д.), что значительно сужает прост-

ранство поиска. Если удастся правильно понять сущность задачи и оптимально разбить ее на систему иерархически связанных целей-подцелей, то можно добиться того, что путь к ее решению в пространстве поиска будет минимален. Однако если задача является плохо структурированной, то сделать это невозможно.

Альфа-бета алгоритм. Задача сводится к уменьшению пространства состояний путем удаления в нем ветвей, не перспективных для поиска успешного решения. Поэтому просматриваются только те вершины, в которые можно попасть в результате следующего шага, после чего неперспективные направления исключаются из дальнейшего рассмотрения. Например, если цвет предмета, который мы ищем, не красный, то его бессмысленно искать среди красных предметов. Альфа-бета алгоритм нашел широкое применение в основном в системах, ориентированных на различные игры, например в шахматных программах, однако он может использоваться и в продукционных системах для повышения эффективности поиска.

КОММЕРЧЕСКИЕ ПРОДУКЦИОННЫЕ СИСТЕМЫ

Большинство коммерческих систем, основанных на знаниях, является продукционным. Это прежде всего такие системы, как MYCIN, DENDRAL, PROSPECTOR, PUFF, INTERNIST, XCON или SACON (см. приложение Д). Их базы знаний насчитывают сотни правил. Имеются версии, которые работают как на больших, так и на мини-компьютерах. Время разработки каждой из них составило приблизительно десять человеко-лет, хотя более поздние системы, в частности PUFF и XCON, были созданы несколько быстрее, поскольку уже имелись развитые методы их построения. Области применения перечисленных выше экспертных систем приведены в табл. 3.1.

Известны несколько примеров успешной реализации экспертных систем на персональных компьютерах. Однако почти все эти системы предназначались для решения задач из относительно небольших предметных областей. Примеры такого рода были приведены в гл. 1 и 2. Видимо, наиболее плодотворными при реализации на персональных компьютерах окажутся идеи и методы искусственного интеллекта, позволяющие сделать традиционные пакеты программ более дружественными к пользователю и простыми в применении. Текстовые редакторы, электронные таблицы и СУБД будут расширены базами знаний продукционного типа, средствами вывода и интерфейсами на естественном языке, которые органично войдут в состав пакетов и повысят уровень их собственного "интеллекта".

ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА

Одно из наиболее замечательных свойств продукционных систем - возможность создания на их основе совершенно новой экспертной системы путем простой замены базы знаний. В частности, система MYCIN, предназначенная для диагностики и лечения менингита и инфекционных заболеваний, путем замены базы знаний может быть превращена в систему PUFF, ориентированную на диагностику и лечение заболеваний совершенно другого типа, а именно легочных. Та часть системы, которая остается неизменной после перепрофилирования, получила название *оболочки*. Она включает в себя такие компоненты, как механизм вывода, подсистемы объяснения и взаимодействия на естественном языке, а также рабочую память. Оболочка, полученная путем удаления базы знаний из системы MYCIN, известна как EMYCIN.

Таблица 3.1

Характеристики некоторых продукционных систем

Название системы	Организация-разработчик	Область применения
DENDRAL	Станфордский университет	Органическая масс-спектрометрия
INTERNIST	Питсбургский университет	Диагностика внутренних болезней
MYCIN	Станфордский университет	Диагностика инфекционных заболеваний и менингита
PROSPECTOR	SRI International	Геологическая разведка
PUFF	Станфордский университет	Диагностика легочных заболеваний
XCON	DEC	Определение конфигурации компьютерных систем

Оболочка часто поставляется в виде самостоятельного программного средства, которое позволяет инженеру знаний создать специализированную базу правил и после наполнения ее экспертной информацией получить полную систему, основанную на знаниях. Качество такой системы будет зависеть только от объема базы знаний и адекватности содержащихся в ней правил. Это дает возможность сократить время разработки системы, так как при на-

личии оболочки инженер знаний концентрирует свои усилия на построении одной лишь базы знаний. Следует, правда, отметить, что инженеру знаний тем самым навязывается единственная стратегия вывода, которая заложена в оболочку. Однако хорошие оболочки представляют собой достаточно гибкий инструментарий, поскольку располагают несколькими стратегиями вывода, из которых инженер может выбрать наиболее соответствующую специфике решаемой задачи.

Применение коммерческих оболочек весьма значительно сокращает время создания систем, основанных на знаниях. В настоящее время при использовании оболочек построение экспертных систем занимает около восьми человеко-часов на правило. Некоторые из инструментальных средств, ориентированных на персональные компьютеры, приведены в табл. 3.2. Эволюция коммерческих экспертных систем и инструментальных средств показана на рис. 3.7. Более полный перечень таких систем и средств содержится в приложениях Г и Д. Во второй части книги вы сможете ознакомиться с листингами программ, реализующих оболочку, а добавив конкретную базу правил, получите полную систему, основанную на знаниях.

Таблица 3.2

Примеры оболочек продукционного типа, ориентированных на персональные компьютеры типа IBM PC и TI Professional Computer

Оболочка	Фирма-изготовитель	Цена, дол.	Язык реализации
ES/PADVISR	Expert Systems International	1895	Пролог
INSIGHT	Level 5 Research	95	Паскаль
M.1	Teknowledge	12500	Пролог
PERSONAL CONSULTANT	Texas Instruments	3000	IQLISP
SERIES-PC	SRI INTERNATIONAL	5000	IQLISP
KES	Software A&E	4000	IQLISP
EXPERT-2	Mountain View Press	100	Форт

При выборе для последующего приобретения инструментального средства построения экспертных систем инженер знаний дол-

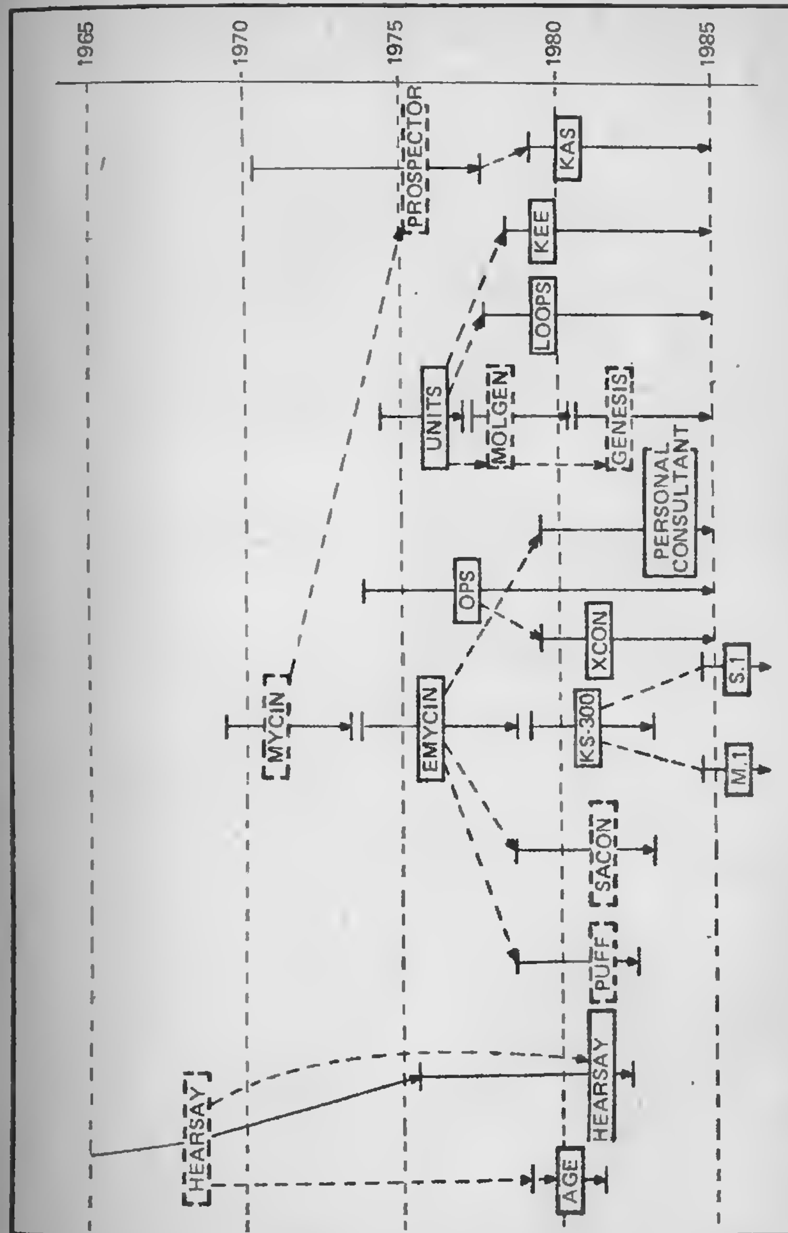


Рис. 3.7. Эволюция коммерческих экспертных систем и инструментальных средств

жен быть очень внимателен. Чтобы выбор был удачным, необходимо учесть все характеристики разрабатываемой с его помощью экспертной системы - число правил в базе знаний, особенности интерфейса с пользователем, вид решаемой задачи и стратегии вывода. Так как последняя является частью механизма вывода, тип оболочки предопределяет и набор доступных стратегий вывода. Цена оболочки - немаловажный параметр. Если вы предполагаете использовать оболочку для разработки коммерческой системы, основанной на знаниях, которая будет продана большому количеству потребителей, то необходимо учесть ее стоимость при определении цены конечного продукта. В отличие от компилятора с языка программирования, затраты на который окупаются за время эксплуатации, оболочка в этом смысле подобна интерпретатору, т.е. ее копию должен иметь каждый пользователь. К изложенному следует добавить, что лучшие образцы инструментальных средств могут эффективно применяться только после того, как будущие разработчики пройдут соответствующий курс обучения. Стоимость обучения также следует включить в смету построения системы, основанной на знаниях.

ЯЗЫКИ ПРОГРАММИРОВАНИЯ СИСТЕМ, ОСНОВАННЫХ НА ЗНАНИЯХ

Системы, основанные на знаниях, могут быть реализованы практически с помощью любого языка программирования, однако наиболее известные из них были разработаны на языках, позволяющих выполнять символьную обработку, таких, как Лисп и Форт (см. гл. 6). Последние являются языками интерпретирующего типа, что замедляет скорость выполнения программ. Поэтому в настоящее время ведутся работы по созданию компиляторов для языков символьной обработки. Ожидается, что это даст новый мощный импульс к построению экспертных систем.

Более широкими возможностями обладают языки программирования, специально предназначенные для разработки систем, основанных на знаниях (рис. 3.8). Одним из таких языков является Пролог, транслятор с которого часто пишется на Лиспе. Пролог позволяет весьма просто создавать эти системы.

И наконец, самый высокий уровень по своим возможностям для построения систем, основанных на знаниях, занимают оболочки. Они также часто разрабатываются на языке Лисп (или Пролог), и поэтому тоже работают в режиме интерпретации. Оболочки обладают меньшей гибкостью и стоят дороже по сравнению с трансляторами языков программирования, однако их применение сокращает время создания экспертных систем и позволяет разработчику полностью сосредоточиться на построении базы знаний.

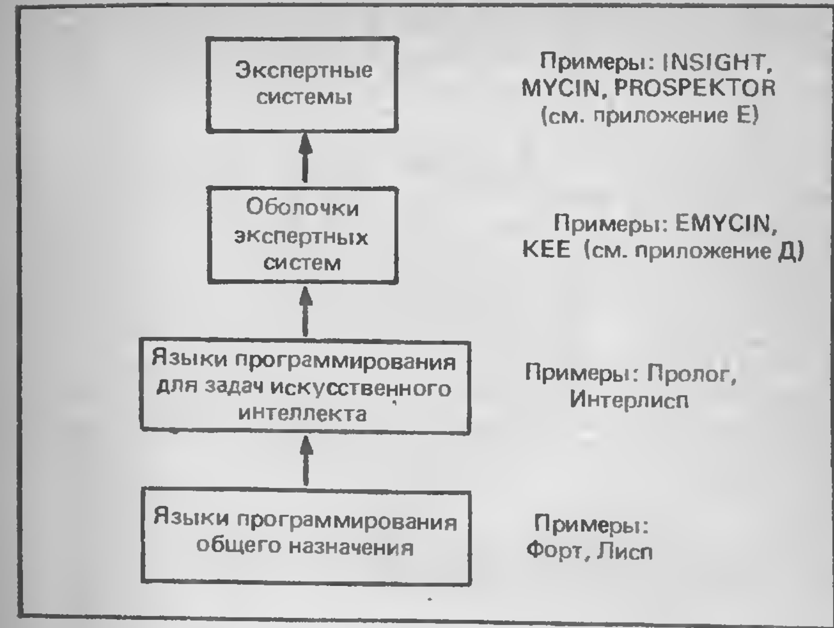


Рис. 3.8. Программные средства построения систем, основанных на знаниях

ПРОДУКЦИОННАЯ СИСТЕМА - СРЕДСТВО МОДЕЛИРОВАНИЯ ПРОЦЕССОВ МЫШЛЕНИЯ ЧЕЛОВЕКА

Одной из причин высокой популярности продукционных систем как моделей представления знаний является совпадение схемы их функционирования с формой процессов мышления человека (в том числе и эксперта) при решении задач. Долговременной памяти человека можно поставить в соответствие базу знаний, а кратковременной – рабочую память или базу данных экспертной системы. Символьные образы создаются в кратковременной памяти из понятий и связей между ними. Они используются для нахождения и выполнения продукций, хранимых в долговременной памяти. После срабатывания продукции ее заключение добавляется в кратковременную память.

Каждый набор взаимосвязанных образов (пар атрибут-значение) из рабочей памяти представляет собой единый чанк. В кратковременной памяти одновременно могут быть обработаны от че-

тырех до семи чанков. Память человека имеет еще одно свойство - способность поддерживать во время решения задачи иерархические связи между чанками. Как вы увидите в следующей главе, это может быть сделано с помощью другой модели представления знаний, основанной на *фреймах*.

Однако следует иметь в виду, что сходство между работой продукционной системы и решением задач человеком не является абсолютным. Действительно, человек-эксперт в своей деятельности не только применяет известные ему правила, но и существенно опирается на свой опыт для распознавания исключений из правил, прибегает к аналогии, использует интуицию и индуктивный вывод. Кроме того, база правил представляет собой всего лишь модель знаний эксперта. По мере приближения к границе познаний эксперта качество его рекомендаций снижается довольно плавно, в то время как система, основанная на знаниях, при выходе за пределы своей области приложений оказывается просто неработоспособной. К этому нужно добавить, что база знаний эксперта весьма динамична - в ней постоянно происходит образование новых понятий, которые эксперт сопоставляет с имеющимися. При решении задач эксперт обучается и по мере необходимости реорганизует свою базу знаний. И наконец, в незнакомой ситуации эксперт всегда может руководствоваться здравым смыслом, что недоступно даже лучшим из компьютеров.

УПРАЖНЕНИЯ

1. Приведите какие-либо общие стратегии сужения пространства поиска, применимые в шахматных компьютерных программах.

2. При разработке стратегии вывода на каком правиле вы остановите свой выбор - на том, при котором у вас наибольшая вероятность потерпеть неудачу, или на том, при котором вероятность неудачи наименьшая? Почему?

3. Сопоставьте эффективность традиционных компьютерных программ, использующих для решения задач процедуры, логику, деревья решений, и систем, основанных на знаниях, которые ориентированы на решение этого же класса задач. Перечислите достоинства и недостатки систем, основанных на знаниях. Как изменится объем базы знаний и что с ней произойдет, если некоторые фрагменты знаний, используемые для решения задач, окажутся неоднозначными?

4. За счет чего применение инструментальных средств упрощает процесс построения систем, основанных на знаниях?

5. Какие доводы вы приведете при выборе инструментальных средств для решения конкретной задачи? Какие критерии при этом будете использовать?

Глава 4

ПРЕДСТАВЛЕНИЕ ЗНАНИЙ

В предыдущей главе были рассмотрены экспертные системы, использующие конкретный способ представления знаний, в основе которого лежат *продукции*, или *правила*. Здесь обсуждается альтернативный способ представления знаний в сопоставлении с вышеупомянутым. Чтобы сопоставление было более понятным, мы проведем его на примере простой задачи, построив для нее фрагмент базы знаний с применением как того, так и другого способа. В качестве примера приведем игру, которая называется "Отгадай животное". Этот пример может показаться тривиальным, но следует иметь в виду, что путем замены правил на совокупность других или увеличением их числа можно создать систему любой степени сложности, ориентированную на решение задач из какой угодно предметной области.

Все способы представления знаний имеют свои ограничения. Поэтому важной проблемой, которая еще долго будет оставаться актуальной, является разработка общей теории или метода представления знаний произвольного вида.

ИГРА "ОТГАДАЙ ЖИВОТНОЕ"

Суть игры состоит в том, что пользователь задумывает какое-либо животное, а компьютер, задавая вопросы, пытается отгадать его. База знаний, необходимая для проведения игры, содержит сведения о признаках различных видов животных и хранится в памяти компьютера, который обращается к ней для организации диалога с пользователем. Фрагмент типичного диалога приведен на рис. 4.1. Компьютер задает пользователю серию вопросов, а пользователь может отвечать на них только словами "да" и "нет". Анализируя ответы, компьютер со временем отгадывает задуманное животное. В любой момент диалога пользователю предоставляется возможность задать вопрос "почему?", после которого компьютер вос-

производит цепочку своих рассуждений, объясняющую, почему именно был задан тот или иной вопрос.

ПРЕДСТАВЛЕНИЕ ЗНАНИЙ В ВИДЕ ПРАВИЛ

В продукционной системе (системе, использующей при выводе сопоставление по образцу), рассматривавшейся в предыдущей главе, мы могли бы представить необходимые для игры знания в виде набора правил. Как известно, правила состоят из двух частей - антецедента и консеквента. В нашем примере база знаний включает 15 правил, которые дают возможность распознавать животных семи видов. На рис. 4.2 приведены сами эти правила, а на рис. 4.3 изображена сеть, иллюстрирующая связи между ними.

В процессе игры формируется также рабочая память, куда заносятся выявленные в процессе диалога признаки животных. В начальный момент времени в рабочей памяти отсутствуют какие-либо данные о животных. Система выдвигает некоторую гипотезу о виде животного, а затем, используя обратный вывод, пытается определить те вопросы, которые необходимо задать для подтверждения этой гипотезы. Например:

[(Гипотеза) Задуман альбатрос]
(Требуемый вопрос) Может летать?

Являются ли следующие утверждения истинными?
(МОЖЕТ ЛЕТАТЬ)?
Да/Нет? - НЕТ
(ИМЕЕТ ПЕРЬЯ)
Да/Нет? - НЕТ
(ДАЕТ МОЛОКО)
Да/Нет? - ДА
(ЖУЕТ ЖВАЧКУ)
Да/Нет? - ДА
(ИМЕЕТ НА ТЕЛЕ ЧЕРНЫЕ ПОЛОСЫ)
Да/Нет? - ДА
Гипотеза (ЗАДУМАННОЕ ЖИВОТНОЕ - ЗЕБРА) подтвердилась

Рис 4.1. Фрагмент диалога из игры "Отгадай животное"

Рассматривая рис. 4.1 и 4.3, легко заметить, как система реализует обратный вывод - сначала выдвигается гипотеза, затем

ПРАВИЛО 01 ЕСЛИ животное имеет волосяной покров
ТО это животное - млекопитающее

ПРАВИЛО 02 ЕСЛИ животное дает молоко
ТО это животное - млекопитающее

ПРАВИЛО 03 ЕСЛИ животное имеет перья
ТО это животное - птица

ПРАВИЛО 04 ЕСЛИ животное может летать
И откладывает яйца
ТО это животное - птица

ПРАВИЛО 05 ЕСЛИ животное ест мясо
ТО это животное - хищник

ПРАВИЛО 06 ЕСЛИ животное имеет острые зубы
И животное имеет когти
И его глаза смотрят вперед
ТО это животное — хищник

ПРАВИЛО 07 ЕСЛИ животное является млекопитающим
И имеет копыта
ТО это животное - парнокопытное

ПРАВИЛО 08 ЕСЛИ животное является млекопитающим
И жует жвачку
ТО это животное - парнокопытное

ПРАВИЛО 09 ЕСЛИ животное является млекопитающим
И это животное - хищник
И это животное желто-коричневого цвета
И это животное имеет темные пятна
ТО МОЖНО ПРЕДПОЛОЖИТЬ, ЧТО
это животное - гепард

ПРАВИЛО 10 ЕСЛИ животное является млекопитающим
И это животное - хищник
И это животное - желто-коричневого цвета
И это животное имеет темные полосы
ТО МОЖНО ПРЕДПОЛОЖИТЬ, ЧТО
это животное - тигр

ПРАВИЛО 11 ЕСЛИ животное является парнокопытным
И имеет длинную шею
И имеет длинные ноги
И имеет черные пятна
ТО МОЖНО ПРЕДПОЛОЖИТЬ, ЧТО
это животное - жираф

ПРАВИЛО 12 ЕСЛИ животное является парнокопытным
И имеет черные полосы
ТО МОЖНО ПРЕДПОЛОЖИТЬ, ЧТО
это животное - зебра

ПРАВИЛО 13 ЕСЛИ животное является птицей
И не может летать
И имеет длинную шею
И имеет длинные ноги
И имеет черно-белую окраску
ТО МОЖНО ПРЕДПОЛОЖИТЬ, ЧТО
это животное - страус

ПРАВИЛО 14 ЕСЛИ животное является птицей
И не может летать
И может плавать
И имеет черно-белую окраску
ТО МОЖНО ПРЕДПОЛОЖИТЬ, ЧТО
это животное - пингвин

ПРАВИЛО 15 ЕСЛИ животное является птицей
И хорошо летает
ТО МОЖНО ПРЕДПОЛОЖИТЬ, ЧТО
это животное - альбатрос

Рис. 4.2. Продукции для игры "Отгадай животное"

отыскивается подтверждающий ее вопрос, который находится на самом верхнем уровне сети, после чего включается прямой вывод от вопроса к заключению. Если, например, на первый вопрос получен ответ "да", то в рабочую память заносится факт (МОЖЕТ ЛЕТАТЬ).

В рабочей памяти хранятся пары атрибут - значение, истинность которых установлена в процессе решения конкретной задачи. Поэтому для экономии памяти система выдвигает предположение о виде объекта (животного). В системах этого типа правила из базы знаний состоят из пар атрибут-значение (А - З), причем каждая часть антецедента и консеквента содержит единственный атрибут и единственное значение. Например:

	Атрибут	Значение
ЕСЛИ	животное	может летать
И	животное	откладывает яйца
ТО	животное	является птицей

В некоторых продукционных системах каждое высказывание состоит из атрибута, объекта и значения атрибута. В этом случае антецеденты и консеквенты состоят из троек атрибут - объект - значение (А - О - З). Так, например, база знаний системы MYCIN включает следующее правило:

ЕСЛИ Окраска микроорганизма по Граму является грам-отрицательной
И Организм является бактерийной палочкой
И Микроорганизм рос в анаэробных условиях
ТО Можно определенно утверждать, что микроорганизмом является baktcroides

В форме троек А - О - З это правило будет иметь следующий вид:

	Атрибут	Объект	Значение
ЕСЛИ	Окраска по Граму	Микроорганизм	Грам-отрицательная
И	Является	Микроорганизм	Бактерийная палочка
И	Условия роста	Микроорганизм	Анаэробные
ТО	Является	Микроорганизм	Bacteroides

Заметим, что в нашем случае все объекты идентичны. В системах, где все объекты идентичны, выдвинутая гипотеза также может быть представлена в виде пары А - З. В системе MYCIN использование троек А - О - З дает возможность рассматривать пациента или культуру тоже как объект. В большинстве существующих продукционных систем применяются пары А - З, что упрощает процесс формирования правил.

В системах, основанных на знаниях, существует четкая грань между следствием, или гипотезой, с одной стороны, и заключением - с другой. Следствие, или гипотеза, носит предположительный характер, поскольку оно получено на основании фактов, которые могут быть как истинными, так и ложными. Заключение представляет собой факт, полученный путем вывода на основании фактов, которые считаются истинными.

Например, если животное летает и откладывает яйца, то мы можем вывести *следствие (выдвинуть гипотезу)*, что это пингвин (см. рис. 4.3). Но наша гипотеза нуждается в дополнительной проверке. Однако на основании тех же фактов мы можем прийти к истинному *заключению*, что отгадываемое животное - птица. Предполагая истинность выдвинутой гипотезы, можно спросить у пользователя, умеет ли птица плавать. Получив утвердительный ответ, мы вправе предположить, что это пингвин. В конце концов на каком-то шаге мы придем к заключению, что рассматриваемый объект - определенно пингвин. Заключения заносятся в рабочую память или базу данных. Гипотезы никогда не включаются в базу данных, но используются при выводе.

Часто бывает необходимо включить в правила элемент неопределенности. Например, заключение упомянутого выше правила из базы знаний системы MYCIN в действительности выполняется только в 70% случаев. В зависимости от вида системы фактор определенности может назначаться как антецеденту, так и консеквенту. Способы включения элементов неопределенности в базу зна-

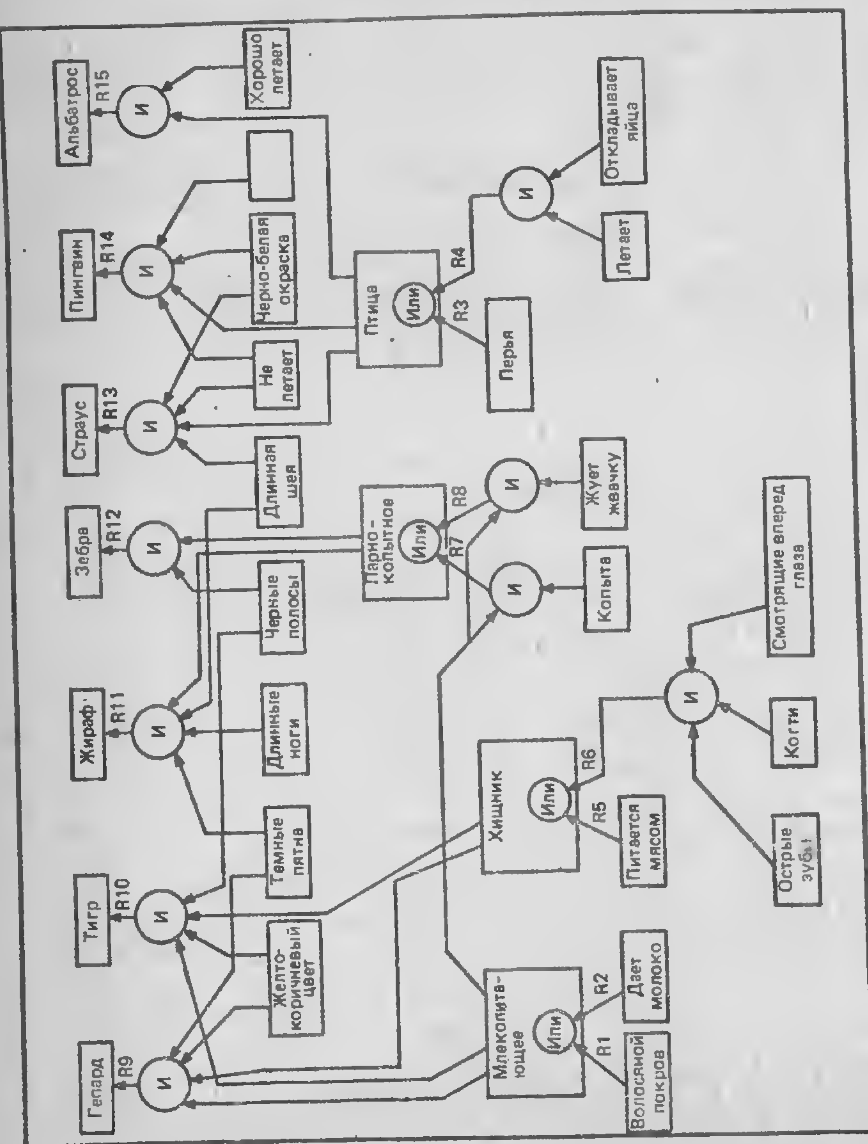


Рис. 4.3. Сеть правил игры "Отгадай животное"

ний и методы их обработки при выводе будут более подробно рассмотрены в гл. 10.

Продукции имеют по сравнению с другими классами представления знаний следующие преимущества:

* модульность;
* единообразие структуры (основные компоненты экспертной системы могут применяться для построения систем с иной проблемной ориентацией);
* естественность (вывод заключения в продукционной системе во многом аналогичен процессу рассуждений эксперта);
* гибкость родовидовой иерархии понятий, которая поддерживается только как связи между правилами (изменив правило вы тем самым внесете изменение и в иерархию).

Однако продукционные системы не свободны от недостатков:

* процесс вывода менее эффективен, чем в других системах, поскольку большая часть времени при выводе затрачивается на непроизводительную проверку применимости правил;
* этот процесс трудно поддается управлению;
* сложно представить родовидовую иерархию понятий.

Представление знаний с помощью продукций иногда называют *плоским*. Реляционную базу данных также считают плоской, поскольку в ней в отличие от систем представления знаний не отражена в явном виде иерархия понятий, т.е. родовидовые отношения между объектами из базы данных. В этом же смысле следует понимать и плоскость продукционных систем, так как в них отсутствуют средства для установления иерархии правил. Объем базы знаний продукционных систем растет линейно по мере включения в нее новых фрагментов знаний, в то время как в традиционных алгоритмических системах, использующих деревья решений, зависимость между объемом базы знаний и количеством собственно знаний является логарифмической.

Большинство существующих коммерческих систем, основанных на знаниях, - продукционные. Краткие сведения о некоторых из них были приведены в предыдущей главе (см. рис. 3.1).

ФРЕЙМЫ

Продукционные системы, базы знаний которых насчитывают сотни правил, отнюдь не считаются чем-то необычным. При такой сложности системы для инженера знаний процесс обновления состава правил и контроль связей между ними становится весьма затруднительным, поскольку добавляемые правила могут дублировать имеющиеся или вступать с ними в противоречие. Для выявления подобных фактов можно использовать программные средства, но включение их в работу системы приводит к еще более тяжелым последствиям - потере ею работоспособности, так как в этом случае инженер знаний теряет представление о том, как взаимодейст-

вуют правила. Сеть, отражающая взаимосвязи правил, в такой ситуации похожа на схему организационной структуры федерального правительства. Количество опосредованных родовидовых связей между понятиями резко возрастает, и инженеру знаний трудно их контролировать.

Представление знаний, основанное на фреймах, является альтернативным по отношению к системам продукций: оно дает возможность хранить родовидовую иерархию понятий в базе знаний в явной форме. *Фреймом* называется структура для описания стереотипной ситуации, состоящая из характеристик этой ситуации и их значений, характеристики называются *слотами*, а значения - *заполнителями слотов*. Пример типичной системы фреймов, описывающих различные классы птиц, приведен на рис. 4.4. Слот может содержать не только конкретное значение, но и имя процедуры, позволяющей вычислить его по заданному алгоритму, а также одну или несколько продукций (эвристик), с помощью которых это значение можно найти.

В слот может входить не одно, а несколько значений. Например, в слоте БРАТ фрейма, описывающего больного, допускается одновременно несколько имен. Иногда слот включает компонент, называемый фасетом, который задает диапазон или перечень его возможных значений. Фасет указывает также граничные значения заполнителя слота (например, максимально допустимое число братьев).

Как уже отмечалось, помимо конкретного значения, в слоте могут храниться процедуры и правила, которые вызываются при необходимости вычисления этого значения. Можно сказать, что такие процедуры, или правила, действуют как демоны (см. гл. 3), активизируемые только по мере надобности. Если, например, фрейм, описывающий человека, включает слоты ДАТА РОЖДЕНИЯ и ВОЗРАСТ и в первом из них находится некоторое значение, то во втором слоте может стоять имя процедуры, вычисляющей возраст по дате рождения и текущей дате. Процедуры, располагающиеся в слоте, называются *связанными процедурами* (см. гл. 10). В приведенном примере связанная процедура будет активизироваться при каждом изменении текущей даты.

Совокупность фреймов, моделирующая какую-либо предметную область, представляет собой иерархическую структуру, в которую фреймы соединяются с помощью родовидовых связей. На верхнем уровне иерархии находится фрейм, содержащий наиболее общую информацию, истинную для всех остальных фреймов. Фреймы обладают способностью наследовать значения характеристик своих родителей, находящихся на более высоком уровне иерархии. Так, фрейм АФРИКАНСКИЙ СЛОН наследует от фрейма СЛОН значение СЕРЫЙ характеристики ЦВЕТ. Значения характеристик фреймов могут передаваться по умолчанию фреймам, находящимся ниже них в иерархии, но если последние содержат со-

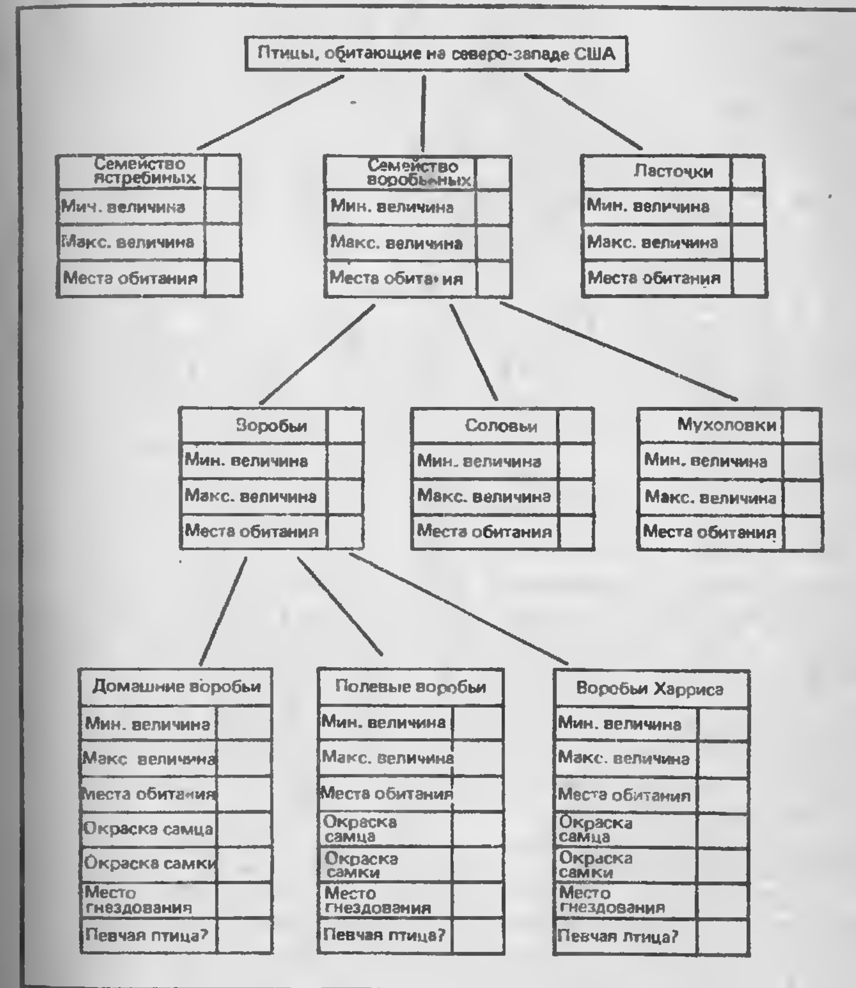


Рис. 4.4. Пример базы знаний, основанной на фреймах

бственные значения данных характеристик, то в качестве истинных принимаются именно они. Это обстоятельство позволяет легко учитывать во фреймовых системах различного рода исключения. В частности, во фрейме АЗИАТСКИЙ СЛОН значением слота ЦВЕТ будет КОРИЧНЕВЫЙ, а не СЕРЫЙ, которое могло бы в нем на-

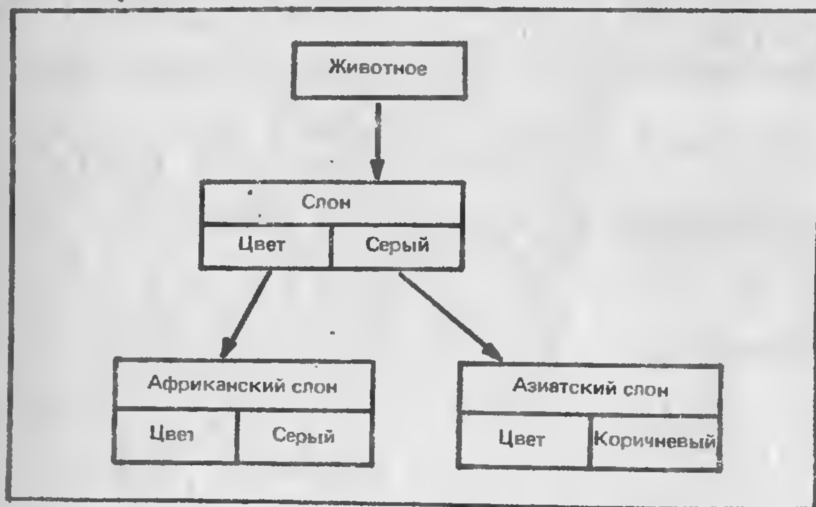


Рис. 4.5. Пример наследования свойств

ходиться, если бы предпочтение при выборе отдавалось не собственному значению, а наследуемому от фрейма СЛОН (рис. 4.5).

Различают статические и динамические системы фреймов. В системах первого типа фреймы не могут быть изменены в процессе решения задачи, в системах второго типа это допустимо.

О системах программирования, основанных на фреймах, говорят, что они являются объектно-ориентированными. Каждый фрейм соответствует некоторому объекту предметной области, а слоты содержат описывающие этот объект данные, т.е. в слотах находятся значения признаков объекта. Фрейм может быть представлен в виде списка свойств, а если использовать средства базы данных, то в виде записи.

Наиболее ярко достоинства фреймовых систем представления знаний проявляются в том случае, если родовидовые связи изменяются нечасто и предметная область насчитывает немного исключений. Во фреймовых системах данные о родовидовых связях хранятся явно, т.е. так же, как и знания всех других типов. Значения слотов представляются в системе в единственном экземпляре, поскольку включаются только в один фрейм, описывающий наиболее общее понятие из всех тех, которые содержат слот с данным именем. Такое свойство систем фреймов дает возможность уменьшить

объем памяти[1], необходимый для их размещения в компьютере. Еще одно достоинство фреймов состоит в том, что значение любого слота при необходимости может быть вычислено с помощью соответствующих процедур или найдено эвристическими методами. Для этого инженер знаний должен заранее разработать все требуемые процедуры и эвристические методы, чтобы включить их в систему на этапе ее проектирования.

Как недостаток фреймовых систем следует отметить их относительно высокую сложность, что проявляется в снижении скорости работы механизма вывода и в увеличении трудоемкости внесения изменений в родовидовую иерархию. Кроме того, во фреймовых системах затруднена обработка исключений. Например, не ясно, как представить в базе знаний игры "Отгадай животное" сведения о гепарде-альбиносе.

В настоящее время имеется лишь небольшое число экспертных систем, где для представления знаний служат фреймы. Самая популярная среди них - система KEE, созданная компанией Intellicorp. Программное обеспечение этой системы выполнено на языке Лисп и требует применения специальной аппаратуры. Стоимость самих программ равна приблизительно 60000 дол., а к ней еще нужно добавить стоимость Лисп-машины, на которой они выполняются, т.е. еще 50000 - 100000 дол. Стоимость же полного комплекта системы составляет не менее 250000 дол.

ПРЕДСТАВЛЕНИЕ ЗНАНИЙ В ВИДЕ СЕМАНТИЧЕСКОЙ СЕТИ

Семантические сети, по мнению специалистов, - наиболее общий способ представления знаний, причем они появились, по-видимому, ранее других. Семантическая сеть отображает совокупность объектов предметной области и отношений между ними, при этом объектам соответствуют *вершины* (или *узлы*) сети, а отношениям - соединяющие их дуги. Так, семантическая сеть, которая представляет факты "Альбатрос является птицей" и "Птица летает и имеет перья", приведена в верхней части рис. 4.6.

В семантическую сеть включаются только те объекты предметной области, которые необходимы для решения прикладных задач. В качестве объектов могут выступать события, действия, обобщенные понятия (например, хищники) или свойства объектов (например, "имеет перья", "может летать"). Свойства представляются в сети также в виде вершин и служат для описания классов объектов.

[1] Однако основным назначением родовидовой иерархии является все же не экономия памяти, а представление в базе знаний семантических связей, существующих между понятиями предметной области. - *Примеч. перев.*

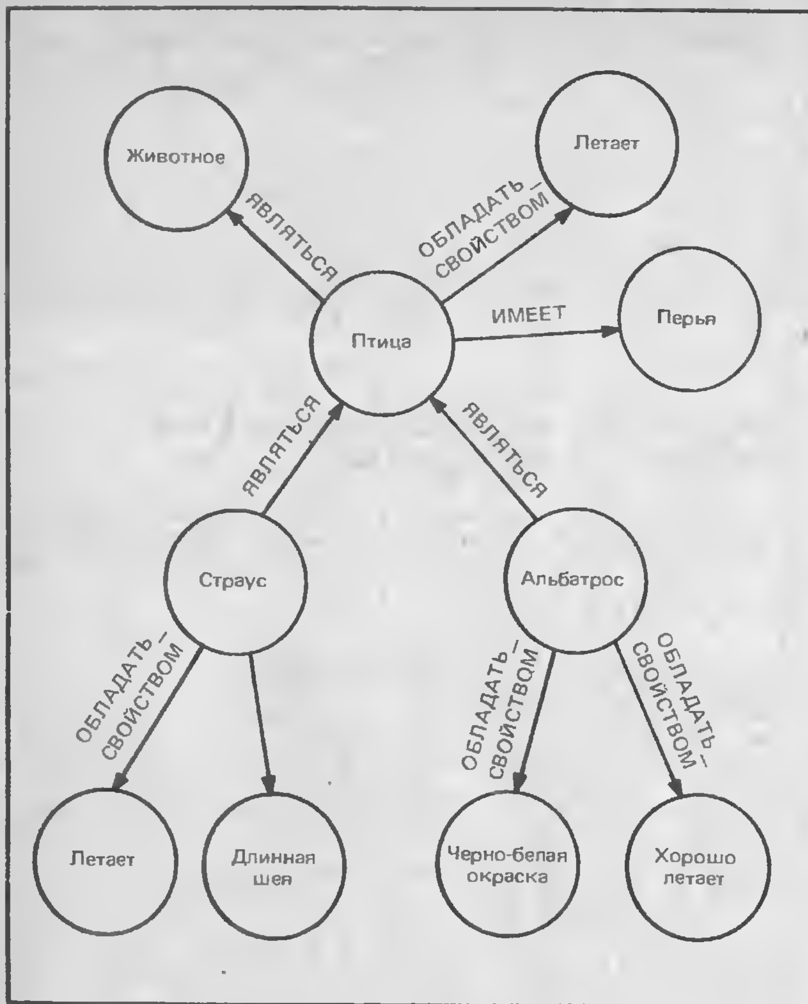


Рис. 4.6. Пример семантической сети

Вершины сети соединяются дугой, если соответствующие объекты предметной области находятся в каком-либо отношении. Наиболее распространенными являются следующие типы отношений:

* БЫТЬ_ЭЛЕМЕНТОМ_КЛАССА (ЯВЛЯТЬСЯ) - означает, что объект входит в состав данного класса, например альбатрос является птицей;

* ИМЕТЬ - позволяет задавать свойства объектов, например жираф имеет длинную шею;
* ЯВЛЯТЬСЯ__СЛЕДСТВИЕМ - отражает причино - следственные связи, например низкое содержание альбумина в крови является следствием нарушения работы печени (рис. 4.7);
* ИМЕТЬ_ЗНАЧЕНИЕ - задает значения свойств объектов, например пациент может иметь двух братьев (рис. 4.8).

Как и в системе, основанной на фреймах, в семантической сети могут быть представлены родовидовые отношения, которые позволяют реализовать наследование свойств от объектов-родителей. Это обстоятельство приводит к тому, что семантические сети приобретают все недостатки и достоинства представления знаний в виде фреймов. Основной недостаток сетей - сложность обработки исключений.

ПРЕДСТАВЛЕНИЕ ЗНАНИЙ СРЕДСТВАМИ ЛОГИКИ ПЕРВОГО ПОРЯДКА

Четвертый способ представления знаний реализуется средствами логики предикатов. Это один из наиболее важных способов, рассматриваемых в книге, поскольку он составляет базис языка Пролог, который будет обсуждаться в гл. 10.

В основе такого представления лежит язык математической логики, позволяющий формально описывать понятия предметной области и связи между ними. В естественном языке существуют грамматические правила, которые задают его *синтаксис*. Эти правила никак не связаны со значением слов, т.е. с семантикой языка. Основными компонентами естественного языка являются слова (существительные, глаголы, предлоги, наречия, прилагательные), предложения и контексты. Правила языка задают порядок следования слов в предложениях.

Точно так же язык, предназначенный для формализации знаний, должен иметь собственный синтаксис и располагать средствами для выражения связей между объектами реального мира. Указанным требованиям отвечают два языка математической логики: логика высказываний и исчисление предикатов, или логика первого порядка. Оба эти языка были разработаны задолго до появления компьютеров. Прежде чем перейти к практическому применению логики для представления знаний, введем ее некоторые понятия.

Основные понятия логики предикатов

Предикатом называется функция, принимающая только два значения - *истина* и *ложь* - и предназначенная для выражения свойств объектов или связей между ними. Выражение, в котором

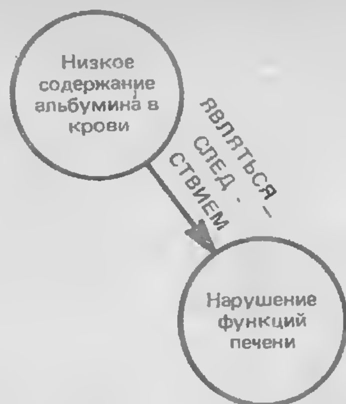


Рис. 4.7. Пример отношения ЯВЛЯТЬСЯ_СЛЕДСТВИЕМ

утверждается или отрицается наличие каких-либо свойств у объекта, называется *высказыванием*. *Константы* служат для именования объектов предметной области. Логические предложения или высказывания образуют *атомарные формулы*. *Интерпретация предиката* - это множество всех допустимых связываний переменных с константами. Связывание представляет собой подстановку констант вместо переменных. Предикат считается общезначимым, если он истинен на всех возможных интерпретациях. Говорят, что высказывание логически следует из заданных посылок, если оно истинно всегда, когда истинны посылки. Рассмотрим, например, утверждение:

ЕСЛИ Животное имеет перья
И Откладывает яйца
ТО Это животное - птица

Здесь высказывание "это животное - птица" логически следует из двух других высказываний. Заключение данного утверждения истинно всегда, когда истинны обе посылки. Правила вывода называются *непротиворечивыми*, если их заключения логически следуют из посылок. Совокупность правил вывода, позволяющих вывести все высказывания, которые логически следуют из посылок, является полной. База знаний считается полной, если любое высказывание, истинность которого вытекает из содержащихся в ней сведений, может быть получено с помощью логического вывода.

Исчисление предикатов с кванторами (логика предикатов)

Наиболее простым языком логики является исчисление высказываний, в котором отсутствуют переменные. Любому высказыванию можно приписать значение *истинно* или *ложно*. Отдельные высказывания могут соединяться связками И, ИЛИ, НЕ, которые называются *булевыми операторами*. Основу исчисления высказываний составляют правила образования сложных высказываний из

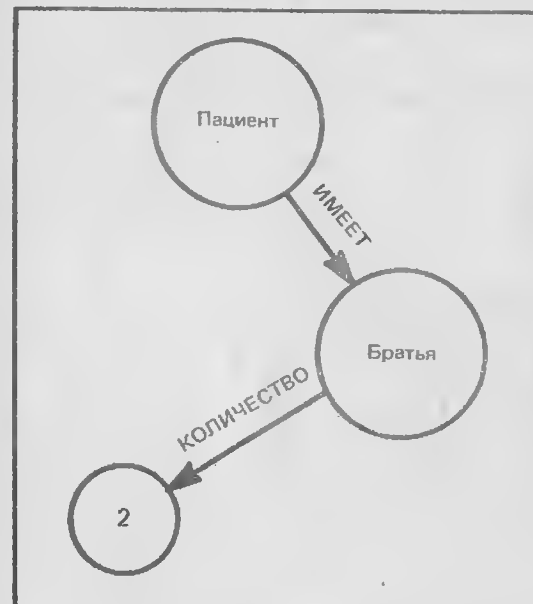


Рис. 4.8. Пример отношения ИМЕТЬ_ЗНАЧЕНИЕ

атомарных. В качестве примеров сложных (составных) высказываний можно привести следующие:

X - ИСТИННО и Y - ЛОЖНО,
X и Y ЛОЖНО,
X или Y ИСТИННО

Здесь переменные обозначают логические высказывания, т.е. высказывания, о которых можно сказать, что они истинны или ложны. Логические операторы имеются в большинстве языков программирования, в таких, например, как Бейсик, Си, Паскаль. Однако исчисление высказываний - недостаточно выразительное средство для обработки знаний, поскольку в нем не могут быть

представлены предложения, включающие переменные с кванторами.

Исчисление предикатов с *кванторами* (логика предикатов) является расширением исчисления высказываний, в котором для выражения отношений между объектами предметной области могут использоваться предложения, включающие не только константы, но и переменные.

Синтаксис языка логики предикатов

Основные символы языка логики предикатов (элементарные составляющие) называются *атомами*. Различают три типа атомов: константы, функциональные символы и предикатные символы.

Константы применяются для обозначения объектов реального мира. Примерами констант могут служить: АЛЬБАТРОС, ПТИЦА, ПАРНОКОПЫТНОЕ, ПЕРЬЯ. Константы состоят из одной или нескольких букв алфавита, цифр или других символов (за исключением скобок).

Функциональные символы обозначают операторы (функции), которые после воздействия на объект возвращают некоторое значение. К числу функциональных символов относятся, например, + , ЦВЕТ, ВОЗРАСТ.

С помощью предикатных символов задаются отношения между объектами, такие, например, как БЫТЬ_БРАТОМ, БЫТЬ_ОТЦОМ, ЯВЛЯТЬСЯ.

Высказывания образуются с помощью этих трех типов атомов, скобок и операторов, в результате чего получаются конструкции, которые называются *термами*. Например, истинное высказывание "альбатрос является птицей" может быть представлено термом:

(ЯВЛЯТЬСЯ АЛЬБАТРОС ПТИЦА)

Используя введенные выше три типа атомов, можно получить высказывания трех видов: атомарные, логические и высказывания с кванторами. Атомарные высказывания состоят из предикатного символа и одного или нескольких термов, например

(ЯВЛЯТЬСЯ КОТ АБИССИНЕЦ)
(БЫТЬ_БРАТОМ ДЖОН ДЖЕК).

Функции также могут быть выражены с помощью предикатных символов:

(= (+ 4 2) 6)

Приведенное атомарное высказывание эквивалентно следующему:

(= (SUM 4 2) 6)

Логические высказывания - это предложения, о которых можно говорить, что они являются истинными или ложными. Логические высказывания могут состоять из атомарных высказываний, соединенных связками И, ИЛИ, НЕТ и ЕСЛИ. Например:

(НЕ (ЯВЛЯТЬСЯ ПТИЦА ЗВЕРЬ))
(ЕСЛИ (И (ЛЕТАЕТ) (ОТКЛАДЫВАЕТ ЯЙЦА))
(ПТИЦА))

И наконец, высказывания с кванторами могут включать одну или несколько переменных, находящихся под действием кванторов ДЛЯ ВСЕХ и СУЩЕСТВУЕТ, например

(ДЛЯ_ВСЕХ x (ЕСЛИ (АБИССИНЕЦ x) (КОТ x)))

Формулы, построенные с помощью логических связок и кванторов (см. табл. 4.1), называются правильно построенными (ППФ). В исчислении предикатов первого порядка предусмотрено достаточно много правил формального вывода, позволяющих образовывать формулы. Как и в продукционных системах, вывод может использоваться для получения новых ППФ из числа имеющихся в результате применения правил вывода. Например, если истинны предикаты:

(ДЛЯ_ВСЕХ x (ЕСЛИ (КОТ x) (ИМЕТЬ_ЛАПЫ x)))
(ДЛЯ_ВСЕХ x (ЕСЛИ (АБИССИНЕЦ x) (КОТ x)))

то можно вывести, что кот Абиссинец тоже имеет лапы, т.е. получить следующий предикат:

(ДЛЯ_ВСЕХ x (ЕСЛИ (АБИССИНЕЦ x) (ИМЕТЬ_ЛАПЫ x)))

Для доказательства истинности ППФ необходимо располагать базой знаний и правилами вывода. Всякий раз, когда в базе данных обнаруживаются высказывания, совпадающие при сопоставлении с посылкой некоторого правила вывода, то его заключение добавляется к имеющимся утверждениям. Этот метод вывода называется *резолюцией*.

С помощью правил, задающих синтаксис языка, нельзя установить истинность того или иного высказывания, причем это распространяется абсолютно на все языки. Высказывание может быть построено синтаксически правильно, но оказаться совершенно бессмысленным.

Еще одним важным понятием исчисления предикатов является процедура *унификации*, позволяющая сравнивать выражения путем их сопоставления. Говорят, что два выражения могут быть унифицированы, если для их переменных существует такая подстановка,

называемая унификатором, в результате которой они становятся идентичными. Пусть, например, истинны следующие предложения:

(БЫТЬ_МАТЕРЬЮ Джейн Мери)
(БЫТЬ_МАТЕРЬЮ Мери Сью)

В них утверждается, что Джейн является матерью Мери, а Мери - матерью Сью. Отсюда можно вывести, что Джейн является бабушкой Сью. Теперь построим общее правило вывода, которое позволяет установить между объектами отношение БЫТЬ_БАБУШКОЙ:

(ДЛЯ_ВСЕХ x y z (ЕСЛИ (И (БЫТЬ_МАТЕРЬЮ x y) (БЫТЬ_МАТЕРЬЮ y z)) (БЫТЬ_БАБУШКОЙ x z)))

Таблица 4.1

Операторы и кванторы исчисления предикатов

ЛОГИЧЕСКИЕ СВЯЗКИ		КВАНТОРЫ	
$\wedge$	И		
$\vee$	ИЛИ	$\exists$	СУЩЕСТВУЕТ
$\neg$	НЕ	$\forall$	ДЛЯ ВСЕХ
$\rightarrow$	ЛОГИЧЕСКИ СЛЕДУЕТ		
$\equiv$	ЭКВИВАЛЕНТНО		

СИСТЕМЫ, ИСПОЛЬЗУЮЩИЕ ПРИНЦИП ДОСКИ ОБЪЯВЛЕНИЙ

В системах, построенных по принципу доски объявлений, существует несколько независимых источников знаний, которые имеют общую рабочую память. Эти источники знаний могут различаться по типам, знания в них могут храниться в виде, наиболее соответствующем их природе, причем допустимы и смешанные представления, но при едином механизме вывода и единственной рабочей памяти. Примером такой системы является система HEARSAY.

ФОРМИРОВАНИЕ БАЗЫ ЗНАНИЙ ПО ПРИМЕРАМ

У тех, кто не имеет специальной подготовки в области инженерии знаний, пользуется популярностью метод *представления знаний по примерам.* Работая с системой такого типа, пользователь задает ей несколько примеров задач вместе с решениями. На их основе система самостоятельно строит базу знаний, которая затем будет применяться для решения других задач. При создании базы знаний пользователь имеет возможность в любой момент вызвать на экран дисплея матрицу, состоящую из примеров задач и их решений, с тем, чтобы установить в ней наличие пустых мест, которые необходимо заполнить соответствующими примерами "задача - решение".

Знания в такой системе также могут храниться в виде правил или в какой-либо иной форме, но для пользователя база знаний всегда существует только в форме матрицы примеров. Эта матрица формируется пользователем, а затем преобразуется во внутреннее представление. Таким образом, база знаний может быть расширена или изменена лишь путем корректировки примеров, содержащихся в матрице, или их добавлением.

Основным достоинством рассматриваемого способа построения экспертных систем является простота его применения, поскольку пользователь может не иметь ни малейшего представления о продукционных правилах, декларативных языках, фреймах или исчислении предикатов. Как недостаток следует отметить отсутствие гибкости процесса построения систем. Пользователь оказывается отстраненным от собственно создания базы знаний и поэтому не может контролировать связи между содержащимися в ней понятиями. Этот способ прекрасно зарекомендовал себя в тех случаях, когда решаемая задача имеет соответствующий аналог в базе знаний. Что же касается задач нового типа, то, даже если они взяты из той же области приложений, система, основанная на изложенных выше принципах, "не умеет" применять свои знания для их решения.

Одной из лучших систем этого типа считается сравнительно недорогая и простая в использовании система EXPERT-EASE. Она может применяться для составления финансовых отчетов о командировках специалистов по снабжению и проверки расходов по соответствующим статьям, принятым в данной организации. База знаний создается путем ввода примеров разрешенных и запрещенных видов расходов на командировку. EXPERT-EASE формирует внутреннее представление базы знаний, с помощью которой она сможет стандартным способом проводить анализ отчетов о командировках.

ПОСТРОЕНИЕ ДЕМОНСТРАЦИОННОЙ БАЗЫ ЗНАНИЙ

Демонстрационная база знаний может быть построена применительно к игре "Отгадай животное", упоминавшейся в этой главе. Для этого следует обратиться к приложению Б. Если вы имеете дискету с системой, то можете сразу приступить к игре, задумав какое-нибудь животное. Если же такой дискеты у вас нет, то, ис-

пользуя приложения А и Б, создайте необходимые компоненты системы и загрузите их в компьютер. После чего можно инициировать игру вводом команды DIAGNOSE (рис. 4.9). Схема базы знаний приведена на рис. 4.3, правила - на рис. 4.2, а их программная реализация - на экранах 81 и 82 в приложении Б.

Система задаст серию вопросов и по вашим ответам на них отгадает задуманное вами животное. Во время игры вы сможете выявить следующие особенности работы системы:

1. Вопросы, которые задаются пользователю, зависят от имеющихся фактов или полученных признаков. Ответы запоминаются в рабочей памяти (в списке FACTS) и затем используются при выводе заключений.

2. Для получения заключений применяется обратный метод вывода. Вначале выводятся более общие свойства животного (например,"животное является млекопитающим"), затем система поэтапно приближается к более конкретным заключениям.

```
DIAGNOSE
ИМЕЕТ_ПЕРЬЯ
Истинно ли это утверждение? (ДА/НЕТ)   --ДА
ХОРОШО_ЛЕТАЕТ
Истинно ли это утверждение? (ДА/НЕТ)   --НЕТ
ЛЕТАЕТ
Истинно ли это утверждение? (ДА/НЕТ)   --НЕТ
НЕ_ЛЕТАЕТ
Истинно ли это утверждение? (ДА/НЕТ)   --ДА
ПЛАВАЕТ
Истинно ли это утверждение? (ДА/НЕТ)   --ДА
ИМЕЕТ_ЧЕРНО-БЕЛУЮ_ОКРАСКУ
Истинно ли это утверждение? (ДА/НЕТ)   --ДА
ЗАКЛЮЧЕНИЕ:
(ЯВЛЯЕТСЯ_ПИНГВИНОМ) --ПРАВИЛЬНО
```

Рис. 4.9. Пример диалога, инициированного командой DIAGNOSE в компьютерной игре "Отгадай животное"

УПРАЖНЕНИЯ

1. Какие способы представления знаний более всего подходят для решения каждой из следующих прикладных задач:

а) классификация биологических видов (птиц, цветов и т. д.);
б) поиск неисправностей в компьютере;
в) медицинская диагностика.

2. Как можно оценить сложность прикладной задачи?

3. Как способ представления знаний в системе зависит от ее типа (система диагностики, планирования или интерпретации)? Может быть, такая зависимость вообще отсутствует?

4. Как оценить мощность имеющихся знаний? Возможно ли это сделать в принципе?

5. Приведите пример метазнаний, которые могут быть использованы в экспертной системе, предназначенной для поиска неисправностей в автомобиле.

6. Представьте эквиваленты на естественном языке для следующих логических высказываний:

(ВОЗРАСТ ДЖОН 32)
(НЕ (СЕМЕЙНОЕ_ПОЛОЖЕНИЕ МЕРИ НЕ_ЗАМУЖЕМ))
(ДЛЯ_ВСЕХ x (ЕСЛИ (БЫТЬ_ЯБЛОКОМ x) (ЦВЕТ x КРАСНЫЙ))).

7. К какому типу относится каждое из приведенных выше высказываний?

8. Запишите следующие предложения на языке исчисления предикатов:

Билл - это конь.
Джон - это собака.
Существует лошадь, которая обгонит любую собаку.
Джейн - сестра Сью.

9. Постройте систему фреймов, описывающую организационную структуру небольшого предприятия. Приведите пример наследования свойств в такой системе.

Глава 5

ИНЖЕНЕРИЯ ЗНАНИЙ

Одной из наиболее сложных проблем, возникающих при создании экспертных систем, является преобразование знаний эксперта и описания применяемых им способов поиска решений в форму, позволяющую представить их в базе знаний системы, а затем эффективно использовать для решения задач в данной предметной области.

Обычно эксперт не прибегает к процедурным или количественным методам; его основные средства - аналогия, интуиция и абстрагирование. Часто эксперт даже не может объяснить, как именно им было найдено решение. В лучшем случае вы получите от него лишь описание основных приемов или эвристик, которые помогли ему успешно справиться с задачей. На инженера знаний возлагается очень сложная работа по преобразованию этих описаний в строгую, полную и непротиворечивую систему, которая позволяла бы решать прикладные задачи не хуже, чем это сделал бы сам эксперт.

В настоящей главе мы рассмотрим построение базы знаний на основе сведений, полученных от эксперта. Процесс ее построения состоит из трех этапов:

* описание предметной области;
* выбор метода и модели представления знаний;
* приобретение знаний.

У читателя может сложиться впечатление, что процесс построения базы знаний - структурированный и линейный. Однако в действительности он гораздо сложнее, поскольку плохо структурирован и по своей природе является скорее циклическим, чем линейным.

Излагаемый здесь материал довольно сложен, но одновременно он относится к числу наиболее важных в книге. Для того чтобы вам было легче уяснить принципы построения базы знаний, этот

процесс рассматривается на примере системы, используемой в медицинской практике при постановке диагноза заболевания на основе результатов химического анализа крови и некоторых симптомов, наблюдаемых у пациента. В каких-то ситуациях мы будем обращаться к примерам из других областей, чтобы проиллюстрировать ряд специфических методов, но главным объектом нашего изучения останется упомянутая выше диагностическая система.

ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ

Первый шаг при построении базы знаний заключается в выделении предметной области, на решение задач из которой ориентирована экспертная система. По сути эта работа сводится к очерчиванию инженером знаний границ области применения системы и класса решаемых ею задач. Сюда входит пять этапов:

* определение характера решаемых задач;
* выделение объектов предметной области;
* установление связей между объектами;
* выбор модели представления знаний;
* выявление специфических особенностей предметной области.

Инженер знаний должен корректно сформулировать задачу. В то же время он должен уметь распознать, что задача не структурирована, и в этом случае воздержаться от попыток ее формализовать или применить систематические методы решения. Главная цель начального этапа построения базы знаний - определить, как будет выглядеть описание предметной области на различных уровнях абстракции. Основные шаги абстрагирования при построении базы знаний показаны на рис. 5.1. Экспертная система включает базу знаний, которая создается путем формализации некоторой предметной области, а та в свою очередь является результатом абстрагирования определенных сущностей реального мира.

Рассмотрим пример из медицинской практики. Признаки и симптомы любого заболевания развиваются по некоторой схеме, которая упрощенно выглядит так. Человек рождается с определенными генетическими характеристиками. На состояние его организма оказывают влияние различные факторы внешней среды: режим питания, климатические и экологические условия, различные стрессы и т.д. Под воздействием этих факторов появляются признаки нарушения функций организма, которые в свою очередь могут быть объединены в симптомы, а симптомы свидетельствуют о возникновении заболевания. Заболевание же, если его не лечить, со временем может привести к смерти.

Все врачи в своей практике сталкиваются с одними и теми же заболеваниями человека, но рассматривают их каждый со своей точки зрения, т.е. их предметные области различаются. Врач - диагност "видит" болезни, которые уже развились в организме. Он выдвигает гипотезу о наличии у пациента некоторого заболевания

4 Заказ № 299

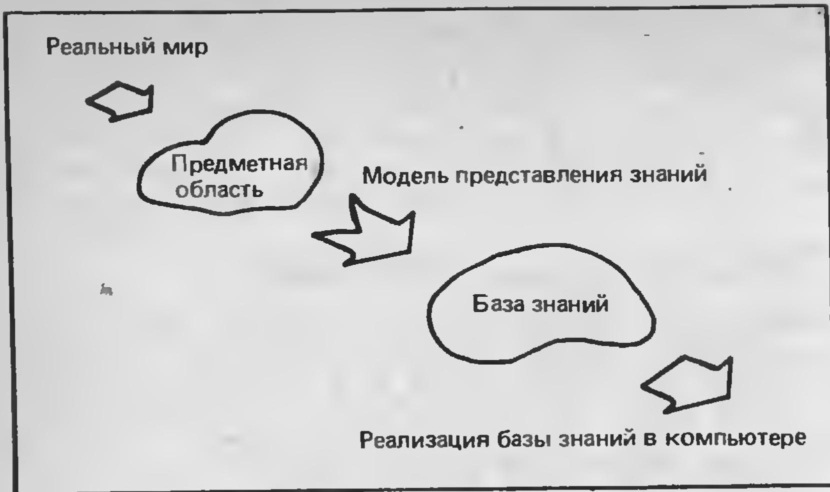


Рис. 5.1. Этапы создания базы знаний для некоторой предметной области

(основываясь при этом на жалобах пациента), а затем, отталкиваясь от своей гипотезы, пытается вылечить больного и восстановить его здоровье. Специалист по профилактике заболеваний умеет хорошо выявлять их признаки и симптомы на ранней стадии, когда болезнь легче всего поддается лечению. Но он испытывает затруднения, если болезнь находится в активной фазе. Врачи-натуропаты, специалисты по нарушениям обмена веществ, специалисты по прогнозированию развития болезней - все они имеют дело с одним и тем же организмом человека, но каждый из них видит разные совокупности симптомов, т.е. различные предметные области (рис. 5.2). Даже врачи-диагносты (такие, как дерматологи) пользуются собственными критериями для установления заболеваний, а значит, располагают специфической предметной областью. Выделение предметной области представляет собой первый шаг абстрагирования реального мира.

После того как предметная область выделена, инженер знаний должен ее формально описать. Для этого ему необходимо выбрать какой-либо способ представления знаний о ней (модель представления знаний). В настоящее время отсутствует общий способ представления знаний, который бы годился для формализации предметных областей любой природы. Инженер знаний должен воспользоваться той моделью, с помощью которой можно лучше всего отобразить специфику предметной области. Когда будет создана общая теория представления знаний (если это вообще когда-нибудь

произойдет), ее можно будет применять для формализации новых предметных областей без учета их особенностей.

Полученная после формализации предметной области база знаний представляет собой результат ее абстрагирования, а предметная область в свою очередь была выделена в результате абстрагирования реального мира. Человек обладает способностью работать с предметными областями различных типов, использовать различные модели представления знаний, рассматривать понятия реального мира с различных точек зрения, выполнять абстрагирования различных видов, проводить сопоставление знаний различной природы и прибегать к самым разнообразным методам решения задач. Компьютеру все это оказывается не под силу. Имеются отдельные примеры совместного использования баз знаний, ориентированных на различные предметные области, но большинство современных систем могут решать задачи только из одной предметной области.

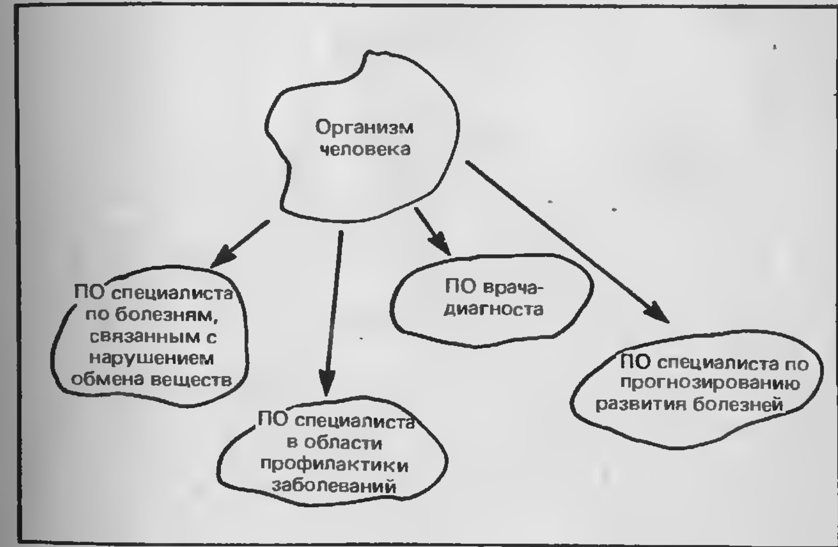


Рис. 5.2. Предметные области (ПО) в медицине

Определение характера решаемых задач

Вернемся к нашему примеру из медицинской практики. Предположим, что мы хотим построить экспертную систему, предназначенную для обработки результатов химического анализа крови, выполненного в лаборатории. Инженер знаний прежде всего обязан

провести опрос эксперта и только потом приступать к построению системы. Эксперт, безусловно, должен быть специалистом в той области, в которой будет работать система. Первым делом необходимо определить целевое назначение системы. Какие, собственно, задачи предстоит решать системе, основанной на знаниях? Цели разработки системы следует сформулировать точно, полно и непротиворечиво. Например, для диагностической системы это может быть получение ответов на такие вопросы:

1. Здоров ли пациент (исправна ли система)? Если нет, то какое именно у него заболевание? Если имеется несколько заболеваний, то какое из них наиболее опасно?

2. Какие изменения в диете и рационе питания следует рекомендовать и какие из них считаются особенно важными?

3. Какие лабораторные исследования необходимо провести дополнительно и какие из них являются первоочередными?

4. Как нужно изменить образ жизни пациента или климатические условия, в которых он находится?

5. Нужно ли направить пациента для обследования к врачам-специалистам и если да, то к каким именно? Подумайте, на какие еще вопросы должна уметь отвечать наша диагностическая система?

После того как цель разработки системы определена, инженер знаний приступает к формулированию подцелей. Это поможет ему установить иерархическую структуру системы и разбить ее на модули. Введение тех или иных подцелей обусловливается наличием связей между отдельными фрагментами знаний. Проблема сводится к разбиению задачи на две или несколько подзадач меньшей сложности и последующему поиску их решений. При необходимости полученные в результате разбиения подзадачи могут дробиться и дальше.

Рассмотрим, в частности, как происходит выделение подцелей в экспертной системе XCON, которая применяется фирмой DEC (Digital Equipment Corporation) для определения конфигурации компьютерных систем по заказу пользователей. Основная задача (нахождение нужной конфигурации компьютерных систем типа VAX) разбивается на шесть следующих подзадач:

1. При наличии серьезных сбоев в оборудовании проверить заказ на покупку.

2. Доставить требуемые компоненты системы в помещение для сборки.

3. Доставить необходимые блоки системы в помещение для сборки.

4. Доставить требуемые платы в помещение для сборки.

5. Определить последовательность соединения компонентов системы.

6. Соединить компоненты системы кабелем.

Действия, которые необходимо выполнить для решения каждой подзадачи, могут существенно различаться в зависимости от того, что именно предпринималось в рамках предыдущей задачи и типа используемых компонентов. База знаний системы XCON насчитывает около 800 правил, предназначенных для определения конфигурации компьютерных систем типа VAX.

Система осуществляет поиск решения, используя обратную стратегию вывода: вначале рассматриваются наиболее общие подзадачи, находящиеся на верхнем уровне иерархии целей, а затем - подзадачи следующего уровня детальности. Цели каждого уровня представляют собой часть общей цели или одно из состояний в пространстве поиска. Инженер знаний, описывая задачу в виде иерархии подзадач, применяет тем самым нисходящий метод: от абстрактного (т.е. от наиболее общих и нестрого сформулированных целей) к конкретному. Таким образом, система решает вначале самые трудоемкие подзадачи, после чего приступает к более простым. Переход на следующий уровень иерархии происходит лишь по завершении всех задач текущего уровня.

В нашем примере исходную задачу можно разбить на несколько подзадач, решение которых сводится к поиску ответов на следующие вопросы:

1. Имеются ли нарушения в системе кровообращения?
2. Имеются ли нарушения в системе пищеварения?
3. В порядке ли нервная система?

Какие еще подзадачи такого рода вы могли бы назвать? Заметим, что все перечисленные подзадачи могут быть увязаны с первым вопросом, находящимся в первой из пяти групп вопросов, список которых приведен выше. Другими словами, упомянутые подзадачи детализируют этот вопрос или выступают подцелями для него.

Из изложенного следует, что система выполняет поиск решения как бы в обратном направлении, все более и более детализируя подзадачи. Процесс детализации продолжается до тех пор, пока не будут обнаружены факты, содержащие непосредственно результаты измерений или лабораторных анализов, т.е. исходные данные поставленной задачи,например:

* результаты химического анализа крови;
* наблюдаемые признаки и симптомы заболеваний;
* климатические условия и наследственные факторы.

Приведите примеры других исходных данных, которые могут потребоваться при постановке диагноза.

Выявление объектов предметной области

Следующим шагом построения базы знаний является выделение объектов предметной области, или в терминах теории систем, установление границ системы. Как и формальная система, совокупность выделенных понятий должна быть точной, полной и непротиворечивой. Итак, какие конкретно лабораторные анализы необходимо провести? Следует ли обратиться к истории болезни пациента и если да, то какие данные в ней наиболее важны? Какие еще сведения о пациенте могут представлять интерес (например, отмечались ли раковые заболевания у родственников)? Нужно ли учитывать лекарства, которые больной принимал ранее, а также предыдущие назначения врачей? Играет ли какую-нибудь роль род занятий и образ жизни больного, климатические условия и режим питания? Какие симптомы у него наблюдаются (головные боли, жар и т.д.)?

Ответы на все перечисленные вопросы позволяют очертить границы исходных данных. А как построить само пространство поиска решения? Для этого необходимо определить подцели на каждом уровне иерархии целей общей задачи. В вершине иерархии следует поместить задачу, которая по своей общности отражает принципиальные возможности и назначение системы.

Установление взаимосвязей между объектами

После выявления объектов предметной области необходимо установить, какие между ними имеются связи. Например, низкое содержание тиреотропного гормона в крови может свидетельствовать о повышенной активности поджелудочной железы, но может означать и нечто другое. Следует стремиться к выявлению как можно большего количества связей, в идеале - всех, которые существуют в предметной области.

Формализация знаний

Полученное качественное описание предметной области должно быть представлено средствами какого-либо формального языка, чтобы привести это описание к виду, позволяющему поместить его в базу знаний системы. Для решения этой задачи выбирается подходящая модель представления знаний, с помощью которой сведения о предметной области можно выразить формально.

Выявление специфических особенностей предметной области

В предметной области необходимо выявить специфические особенности, которые затрудняют решение прикладных задач. Вид этих особенностей зависит от назначения системы (см. гл. 2). Так, для нашей диагностической системы они могут состоять в следующем:

* некоторые из признаков (результатов анализов или симптомов) перекрываются другими;
* отдельные признаки изменяются скачкообразно, их значения зависят от периода наблюдения или внешних условий;
* возможны сбои в лабораторном оборудовании;
* ряд признаков недоступен или их получение связано с большими затратами;
* могут быть известны не все взаимосвязи между признаками.

Следует заметить, что перечисленные особенности характерны для всех систем диагностического типа.

Выбор модели представления знаний

После завершения описанных выше действий необходимо выбрать способ организации выделенных объектов предметной области и определить связи между ними в терминах некоторой модели представления знаний. Этот этап является особенно важным для систем, в которых знания представлены с помощью фреймов и семантических сетей, но играет определенную роль и для систем, основанных на продукциях. Природа некоторых предметных областей такова, что их описание имеет классификационную структуру, т.е. все объекты связаны родовидовыми отношениями БЫТЬ_ЭЛЕМЕНТОМ_КЛАССА (ЯВЛЯТЬСЯ). Примером подобной предметной области может служить биология, в которой классификация птиц и цветов выполняется по иерархическому принципу.

Однако не все предметные области имеют классификационную структуру, и рассматриваемый нами пример из медицинской практики служит тому подтверждением. В таких случаях процесс проектирования экспертной системы может быть упрощен путем разбиения всего множества продукций на классы, что дает возможность проводить групповой ввод и редактирование правил, входящих в один класс, как мы это делали в примере поиска неисправностей в автомобиле. В примере с диагностической системой можно выделить группу правил, связанных с первой подцелью, т.е. позволяющих ответить на вопрос о том, обнаружены ли нарушения в системе кровообращения. С подцелью следующего уровня детальности будет связана другая группа, насчитывающая меньшее число правил.

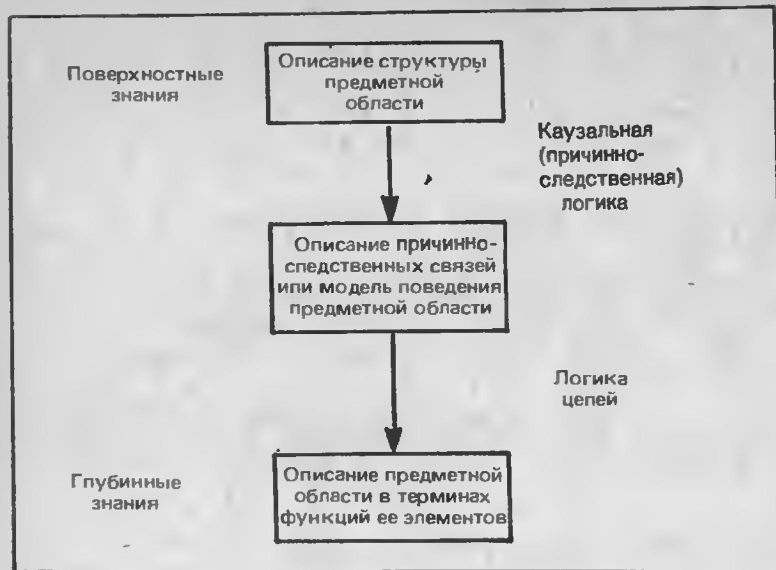


Рис. 5.3. Виды иерархий в предметных областях и связи между ними

Способ организации знаний, состав классов, вид структуры и перечень выделенных объектов в предметной области зависит от того, под каким углом зрения ее рассматривает эксперт и инженер знаний. В предметной области могут быть выделены иерархии трех видов: функциональная, причинно-следственная (она отражает поведение системы) и структурная (рис. 5.3). Давайте на некоторое время отвлечемся от медицинской тематики и рассмотрим виды иерархий, существующие в предметных областях, на примере компьютера. Но прежде отметим следующее:

1. В данном примере каждый вид иерархии соответствует определенной точке зрения на предметную область.
2. Природа предметной области может быть такова, что окажется возможным представить ее как некоторую иерархическую структуру. Мы будем рассматривать два типа иерархий: объективно существующие в предметной области и отраженные в некотором ее представлении. Далее мы поясним эту мысль на примере.
3. Не во всех предметных областях может быть установлена иерархия какого бы то ни было вида.
4. Деление знаний на поверхностные и глубинные, проведенное в данном примере, применимо к любой предметной области. Более того, оно носит общий характер и будет рассмотрено более подробно позднее.

Структурная иерархия

При описании структуры компьютера он рассматривается как совокупность иерархически организованных физических компонентов. В вершине иерархии находится собственно компьютер в целом. На следующем уровне он представляется в виде набора скомпонованных блоков. Третий уровень образован платами, из которых состоит каждый блок. На четвертом уровне его описание детализируется до таких составляющих, как микросхемы.

При отказе компьютера (самый верхний уровень иерархии) вначале проверяется работоспособность каждого блока. После обнаружения неисправного блока проверяются платы, из которых он состоит, а по выявлении неисправной платы в ней проводится поиск дефектной микросхемы. Заметим, что этот метод локализации неисправности не требует наличия у проверяющего никаких знаний об устройстве компьютера или сведений из электроники. Не являются необходимыми и данные о функциях блоков. Те знания, на основе которых осуществляется рассмотренный поиск неисправности, специалисты по экспертным системам называют *поверхностными.*

Причинно-следственная иерархия, или модель поведения системы

Иерархия такого вида позволяет построить модель поведения системы. Функционирование компьютера в этом случае описывается в терминах "причина-следствие". Например, на самом верхнем уровне иерархии нажатие клавиш на клавиатуре должно приводить к высвечиванию соответствующих букв на экране дисплея или выводу на печать. Если ничего подобного не происходит, необходимо спуститься на следующий уровень иерархии и проверить работоспособность принтера. Его действия в терминах "причина-следствие" можно описать так: поступление через линию связи данных от компьютера приводит к их выводу на печать. Если принтер не работает, то нужно спуститься еще на один уровень иерархии и проверить исправность вольтметра или осциллографа. Здесь уже могут иметь место и другие причины отказа принтера, рассмотрение которых даст ключ к локализации неисправности. Так, прежде всего надо посмотреть, горят ли лампочки на его лицевой панели. Если нет, то можно предположить, что в электросети нет напряжения, поврежден кабель электропитания или перегорел предохранитель.

Для поиска неисправности можно использовать программы диагностики. Эти программы сами генерируют возможные причины неполадок, а вы, анализируя результаты, можете определить, какой именно компонент данного уровня вышел из строя. Затем программы диагностики начинают тестировать элементы, находя-

щисся на следующем иерархическом уровне описания компьютера, вследствие чего неисправность может быть выявлена. Например, в компьютере типа IBM PC XT программа диагностики оперативной памяти может выдать на экран дисплея изображение соответствующей платы и указать в ней неисправную микросхему. Таким образом, рассматривая причинно-следственную иерархию, мы применяем знания более глубокого уровня. Однако и их может оказаться недостаточно для восстановления работоспособности компьютера. Конечно, знания о поведении системы являются более содержательными, чем знания о ее структуре, но и они все-таки довольно поверхностны в приложении к такой сложной системе, как современный компьютер.

Функциональная иерархия

На третьем (и самом высоком) уровне абстракции компьютер рассматривается как иерархия модулей или подсистем, выполняющих определенные функции. Так, функция компьютера в целом, находящаяся на вершине этой иерархии, заключается в преобразовании поступивших входных данных в требуемые выходные с помощью заданных алгоритмов или эвристик (если компьютер оснащен системой, основанной на знаниях). На следующем уровне рассматриваемой иерархии мы имеем целый спектр функций, таких, как ввод данных, обработка, запись их в оперативную память, запись и считывание их из внешней памяти, вывод. Если не выполняется основная функция компьютера (не выводятся результаты вычислений или выведенные данные оказываются неверными), то нужно спуститься на следующий уровень иерархии и попытаться выяснить, какая из его функций "дала сбой". Предположим, например, что не удается записать данные во внешнюю память, т.е. неисправен дисковод. Далее осуществляется локализация невыполняющейся функции на том уровне иерархии, где содержится описание дисковода. Вы можете, например, осмотреть дисковод и установить, что находящийся на нем диск не приводится в движение в результате отказа мотора.

Теперь мы можем построить иерархии всех перечисленных выше видов в такой предметной области, как медицина, для чего вновь обратимся к примеру из предыдущего раздела. Так, структурная иерархия может быть введена при рассмотрении физического строения человеческого тела. В нее будут входить следующие уровни: организм в целом, отдельные органы, ткани, клетки, молекулы и гены. В функциональной иерархии можно выделить уровень организма в целом, затем уровень подсистем организма (мышечная и нервная системы, система кровообращения и т.д.). Дробление проводится до получения описания требуемой степени детальности. Таким образом, содержание базы знаний определяется тем, какие объекты, взаимосвязи между ними и виды

иерархий выделены в предметной области. В нашем примере мы воспользуемся причинно-следственной иерархией, поскольку большинство фигурирующих в нем пар атрибут-значение включают в себя названия химических элементов и симптомов заболеваний. Связь между двумя событиями или проявлениями признаков называется *причинной*, если одно из них с необходимостью влечет за собой другое, т.е. мы представляем причинную связь в терминах модели поведения предметной области.

При переходе в описании любой предметной области с уровня поверхностных на уровень глубинных знаний обязательно увеличивается количество имеющейся информации о ней, а также о том, каким образом ее отдельные составляющие объединяются в единое целое. Для перехода от структурной иерархии к описанию поведения системы требуется привлечение соответствующего теоретического аппарата, в качестве которого выступает каузальная (причинно-следственная) логика. Аналогично для получения функциональной иерархии на основе модели поведения системы необходимо использовать такой теоретический аппарат, как логика целей[1] (см. рис. 5.3).

Глубинные и поверхностные знания

Поскольку назначение компьютера наиболее полно выражается в терминах функций, которые он выполняет, функциональная иерархия должна соответствовать наиболее глубокому уровню представления знаний о предметной области. Глубинные знания - это совокупность основных закономерностей, аксиом и фактов о конкретной предметной области. В представлении же с помощью поверхностных знаний структура предметной области выглядит как иерархия составляющих ее физических компонентов. Содержание как глубинных, так и поверхностных знаний полностью обусловливается природой предметной области и видом решаемых задач. Глубинные знания отражают наиболее общие принципы, в соответствии с которыми развиваются все процессы в предметной области, и свойства этих процессов. Поверхностные знания представляют собой эвристики и некоторые закономерности, устанавливаемые опытным путем и используемые при отсутствии общих теорий и методов.

Заметим, что человек в процессе мышления не различает упомянутые выше виды иерархий и имеет равные возможности применять как поверхностные, так и глубинные знания при решении

[1] Упоминаемые здесь логики получили название псевдофизических. С их применением для представления знаний и вывода решений можно ознакомиться по кн.: Поспелов Д.А. Ситуационное управление. - М.: Наука, 1986. - *Прим. перев.*

задач. Эксперт опирается в своей работе на все имеющиеся у него знания, не отдавая себе отчета в том, к какому типу они относятся. При построении же базы знаний, напротив, нужно очень четко представлять, какого типа знания и какого вида иерархию вы в нее включаете.

Для того чтобы лучше уяснить различия между глубинными и поверхностными знаниями, представьте себя в роли инженера знаний, который работает вместе с врачом над созданием экспертной системы, предназначенной для оказания помощи в постановке медицинского диагноза. Присутствуя при беседах врача с пациентами (и просматривая затем видеозаписи таких бесед), вы проследили связь между результатами лабораторных анализов, жалобами пациентов и различными видами заболеваний. Допустим, что вас не интересует путь, которым врач пришел к тому или иному выводу, вы лишь механически включаете в продукционные правила исходные данные (посылки) и вытекающие из них заключения. Это обеспечивает возможность системе, основанной на знаниях, принимать в одинаковых ситуациях идентичные решения. Однако применять такую систему в медицинской практике будет весьма опасно, да и пациенты вряд ли будут ей доверять. Это объясняется тем, что в ней содержатся только поверхностные знания, поскольку при построении системы вы не включили в нее никаких теоретических знаний, полученных врачом в институте, и не отразили его практический опыт.

Аналогичный эффект вы получите, если при построении экспертой системы для игры в покер будете, наблюдая за действиями опытного игрока, запоминать только решения, которые он принимает при наличии той или иной комбинации карт с тем, чтобы воспроизвести их в компьютере. Эти знания также являются поверхностными. Глубинные же знания, позволяющие игроку принимать правильные решения, вновь остались "за кадром".

Для получения глубинных знаний эксперт должен понять внутренние механизмы, действующие в предметной области, и прежде всего основные закономерности, которые обусловливают возможность принятия правильных решений. Существуют предметные области, где они до сих пор не выявлены. В таких случаях следует прибегать к помощи эвристик, т.е. ограничиваться применением поверхностных знаний.

Сопоставление методов представления и реализации знаний

Инженер знаний очерчивает границы предметной области и формализует знания о ней. Модель представления знаний необходима для реализации перехода от содержательного описания предметной области к форме, допускающей его включение в базу знаний экспертной системы (см. рис. 5.1). База знаний игры "Отгадай животное", упоминавшейся в гл. 4, построена на основе продукций. Для реализации данной базы знаний средствами языка Пролог потребуется другой тип представления - логика предикатов. Это означает, что одно представление знаний может быть преобразовано в другое. Запись содержания базы знаний средствами языка Пролог называется ее *реализацией*.

Если вы будете применять какое-либо инструментальное средство для построения экспертной системы, то сможете уловить различие между содержанием знаний (т.е. представлением знаний) и формой хранения и использования знаний (т.е. реализацией). При работе с системой Expert-Ease вы описываете знания в виде матрицы, содержащей примеры исходных ситуаций (условий) и соответствующих им решений. Затем соответствующая программа системы компилирует эти данные во внутреннее представление и загружает их в базу знаний, после чего систему можно использовать для решения новых задач. Способ реализации базы знаний в компьютере часто называют *архитектурой* системы, чтобы отличать его от модели представления знаний.

ПРИОБРЕТЕНИЕ ЗНАНИЙ

Заключительным этапом построения экспертной системы является собственно приобретение знаний и загрузка их в базу знаний. Этот этап начинается с разбиения задачи на максимально возможное число подзадач. Инженер знаний должен выяснить у эксперта, каким образом лучше всего произвести такое разбиение, а также выявить основные приемы, которые эксперт применяет при анализе и решении задач.

В своей деятельности эксперт обычно использует профессиональный язык, лексика которого определяется спецификой предметной области. Этот язык часто оказывается непригодным для описания способа решения задачи в терминах, позволяющих его реализовать в экспертной системе. Но тем не менее не следует заставлять эксперта изучать модели представления знаний, принципы построения экспертных систем или какие-нибудь другие элементы искусственного интеллекта, что входит в компетенцию инженера знаний. От эксперта же требуется лишь умение решать проблемные задачи из соответствующей предметной области.

Процесс приобретения знаний носит скорее циклический, чем линейный характер и структурирован хуже, чем любая другая деятельность, выполняемая при построении экспертной системы. Вначале инженер должен получить от эксперта как можно больше знаний о каком-либо фрагменте или свойстве предметной области. Затем знания подвергаются анализу, в ходе которого в них может выявиться много несогласованностей. В таком случае знания модифицируются и расширяются, после чего опять проверяется их не-

противоречивость. Таким образом поэтапно наращиваются возможности экспертной системы в решении проблемных задач.

Разработку системы, основанной на знаниях, следует проводить в следующей последовательности:

1. Выберите задачу, характер которой дает возможность применить для ее решения технологию экспертных систем.
2. Определите точно цель решения задачи.
3. Вникните как можно глубже в существо задачи.
4. Установите подцели, разбив задачу на подзадачи.
5. Выявите специфические особенности предметной области.
6. Найдите эксперта, специализирующегося в выбранной предметной области, и заручитесь его согласием оказать вам помощь в разработке системы, основанной на знаниях.
7. Участвуя вместе с экспертом в решении нескольких прикладных задач, выявите приемы, которые он применяет. Подробно их опишите.
8. Выберите программные средства, необходимые вам для создания системы. Этот выбор будет зависеть от типа решаемой задачи, ваших финансовых возможностей и сложности предметной области. Выберите технические средства, на которых будет работать ваш программный инструментарий.
9. Постройте лабораторный прототип экспертной системы, позволяющий успешно справиться с примерами тех задач, которые вы решили совместно с экспертом.
10. Приступите к созданию базы знаний. Выявите объекты предметной области, взаимосвязи между ними, виды иерархий, разбейте объекты на классы. Структурируйте базу знаний в соответствии с представлением эксперта о строении предметной области.
11. Выполните необходимое число циклов по наращиванию базы знаний, каждый из которых включает добавление знаний, проверку их непротиворечивости и модификацию с целью устранения обнаруженных несогласованностей.
12. Определите, какие инструментальные средства вам следует использовать для создания системы, и приобретите их.
13. Разработайте документацию на систему.
14. С первых шагов реализации проекта стремитесь к тому, чтобы построить хотя и ограниченную по своим возможностям, но правильно работающую модель, что позволит вам завоевать доверие эксперта, для чего используйте при разработке модульный подход.

При опросе эксперта инженер знаний должен исключить из диалога такие обороты, как ЕСЛИ - ТО, поскольку их применение предполагает, что инженер строит примеры задач, а затем запоминает выданные экспертом решения. Однако такой способ дает возможность получить только поверхностные знания. Более продуктивным считается подход, основанный на построении модели предметной области. Если это ваш первый опыт разработки экспертной системы, то вам следует выбрать задачу, сущность которой абсолютно ясна, и вы легко определите, являются ли достаточными имеющиеся у вас технические и финансовые средства. Лучше недооценить возможности эксперта, чем обмануться в своих ожиданиях. Если вы разработали предложения по созданию экспертной системы, то воспользуйтесь решенными ранее задачами для оценки вероятной стоимости и времени, необходимых для реализации ваших предложений. Постройте прототип системы, который абсолютно правильно решает задачи в небольшой предметной области, - это расположит к вам эксперта, а затем постепенно раздвигайте ее границы.

УПРАЖНЕНИЯ

1. Предположим, что вы решаете прикладную задачу поиска неисправностей в автомобиле.

а) Какую стратегию следует применять эксперту для приобретения знаний, необходимых для решения этой задачи?

б) Какие способы следует применять инженеру знаний для приобретения необходимых сведений и разработки экспертной системы?

2. В чем способ построения экспертной системы совпадает со способом разработки обычных программ? Чем они различаются?

3. Как можно развивать возможности экспертной системы? Определите пути развития экспертной системы, предназначенной для поиска неисправностей в автомобиле. Что в данной задаче поддается измерению, а что - нет?

4. Можете ли вы привести пример прикладной задачи, для решения которой в экспертную систему достаточно заложить лишь поверхностные знания? Как вопросы соотношения глубинных и поверхностных знаний связаны с проблемой обучения системы? Может ли обучение системы начинаться с уровня поверхностных знаний? Для обоснования ответа приведите конкретный пример.

ЧАСТЬ 2

построение систем, основанных на знаниях

Создание систем, основанных на знаниях, является в большей степени искусством, чем наукой, поэтому их качество во многом зависит от уровня практических навыков разработчика. Во второй части книги основное внимание сосредоточено на описании деталей программной реализации экспертных систем. С этой целью мы воспользовались языком Форт - достаточно распространенным и легким в освоении программным средством. Гл. 6 знакомит читателя с языком Форт, на базе которого в дальнейшем ведется изложение методов программирования экспертных систем. Гл. 7 посвящена обработке списков - основному элементу программ, предназначенных для решения задач искусственного интеллекта. В гл. 8 рассматриваются некоторые из более сложных методов, применяемых при разработке таких программ. Затем с помощью введенных понятий описывается процесс построения ядра системы, основанной на знаниях, - интерпретатора продукций, аналогичного интерпретатору языка Пролог, и приводится подробное объяснение всех ключевых моментов данного процесса. В заключение (гл. 10 и 11) обсуждаются некоторые новые результаты в области искусственного интеллекта, не связанные с построением Пролог-системы. Упражнения, которые приведены в конце каждой главы, помогут читателю приобрести необходимые навыки в составлении программ для решения задач искусственного интеллекта.

Деннис Фохт

Глава 6

ЯЗЫК ФОРТ - МОЩНОЕ СРЕДСТВО ПОСТРОЕНИЯ ЭКСПЕРТНЫХ СИСТЕМ

Программы, реализующие экспертную систему, могут быть составлены практически на любом языке программирования. Интерпретатор продукций и базу знаний можно построить даже с помощью таких языков-"ветеранов", как Бейсик и Фортран. Современные языки, в частности Си, Паскаль и Модула-2, обладают в этом отношении большими возможностями, однако и они не совершенны с точки зрения задач искусственного интеллекта. Дело в том, что ни один из них не располагает структурами данных, позволяющими эффективно представлять знания, а также поддерживать процессы доступа, обновления и логического вывода на знаниях.

Средствами специализированных языков искусственного интеллекта факты и правила представляются более легко и естественно. При работе с ними программист получает возможность рассуждать в терминах решаемой задачи, а не конструкций и операторов языка программирования. Среди таких языков по наилучшему представлению предметной сущности задачи следует выделить Пролог и Смоллток. Один из старейших языков - Лисп - до сих пор не утратил своей популярности, несмотря на то, что по возрасту он не уступает Фортрану.

Язык Лисп более универсален, чем Пролог или Смоллток, поскольку его функции обеспечивают выполнение обработки данных на более низком уровне. Основным назначением Лиспа считается обработка списков. Поскольку список - это достаточно общая и естественная структура для выражения знаний, средствами Лиспа могут быть построены самые разнообразные модели представления знаний и механизмы вывода. Именно благодаря своей гибкости и богатой истории Лисп продолжает оставаться в США основным языком построения систем искусственного интеллекта.

Основное внимание на протяжении нескольких последующих глав будет уделено Форту, который был разработан в начале семидесятых годов как язык для управления процессами в реальном масштабе времени. Однако популяризироваться специальной Рабочей группой по языку Форт (Fort Interest Group - FIG) южнее Сан-Франциско он начал только в конце семидесятых, когда его сторонники создали версии трансляторов, работающих на универсальных компьютерах. Компанией General Electric на Форте разработана экспертная диагностическая система, предназначенная для поиска неисправностей в локомотивах.

Форт был выбран в качестве средства, с помощью которого объясняются детали программной реализации систем, основанных на знаниях, по следующим причинам: во-первых, транслятор с этого языка имеется практически на всех типах микрокомпьютеров, во-вторых, он достаточно дешевый, и, наконец, имеет много общего с языками искусственного интеллекта, в частности с Лиспом. В гл. 7 описывается расширение языка Форт, приближающее его по своим возможностям к Лиспу, а в приложении А приведены листинги Форт-программ.

В последующих главах мы рассмотрим элементы Лиспа, Пролога и Смоллтока, причем основное внимание сосредоточим на особенностях этих языков, позволяющих применять их для решения задач искусственного интеллекта. Язык Форт будет описан в деталях, поскольку с его помощью объясняется процесс программирования основных элементов экспертной системы, и в дальнейшем нам придется достаточно часто обращаться к нему. Существует ряд изданий, которые содержат более полное изложение Форта и могут быть рекомендованы для углубленного изучения этого языка. Их список приведен в конце нашей книги[1].

Главными достоинствами языка Форт как средства разработки систем, основанных на знаниях, являются его широкие выразительные возможности, интерактивное исполнение программ, наличие интерпретирующей среды и концептуальная ясность. К недостаткам Форта следует отнести необходимость уделять большее, чем в других языках, внимание деталям реализации программ, особенно при работе с переменными, передаваемыми через стек.

СЛОВА И СЛОВАРЬ ЯЗЫКА ФОРТ

Форт - высокоинтерактивный язык интерпретирующего типа. Его *словарь* содержит *слова*, которые соответствуют либо процедурам, либо данным, причем и те, и другие могут выполняться. Инициирование слова приводит к вызову связанной с ним процедуры. Процедуру, реализующую слово, можно рассматривать как его интерпретацию, поскольку в ней заданы предписываемые этим словом действия.

Форт включает в себя базовый набор слов, причем некоторые из них могут использоваться для создания новых слов. Такие слова называются *определяющими*. В ходе разработки Форт-программы для задания новых слов обычно применяются несколько определяющих слов. Наиболее распространенным из них считается слово

[1]Для изучения Форта можно также порекомендовать следующие книги, вышедшие на русском языке: Баранов С.Н., Ноздрунов Н.Р. Язык Форт и его реализации. - Л.: Машиностроение, 1988; Броуди Л. Начальный курс программирования на языке Форт. - М.: Финансы и статистика, 1989.

":" (двоеточие), которое служит для определения процедуры. Синтаксис задания нового слова таков:

: имя_слова определение ;

Здесь имя определяемого слова следует сразу за двоеточием.

В Форте пробел используется в качестве разделителя слов. Поскольку двоеточие является словом, оно отделяется от имени определяемого слова пробелом. После имени находится последовательность уже определенных слов, выполнение которых в заданном порядке и составляет действие, связываемое с новым словом. Заканчивается определение словом ";" (точка с запятой), служащим признаком завершения данного определения через двоеточие.

Двоеточие - не единственное определяющее слово в языке Форт, к этому же типу относятся слова CODE (код), VARIABLE (переменная) и CONSTANT (константа). Слово CODE применяется для включения в Форт-программу текста на языке ассемблера. В большинстве реализаций для этого применяется ассемблер Форт-системы. Синтаксис слова CODE:

CODE имя_слова ассемблерный_код END_CODE

(Иногда после ассемблерного кода может стоять символ C - синоним слова END_CODE). Возможность включения ассемблерного кода очень важна для решения задач искусственного интеллекта, поскольку позволяет заменять часто выполняемые части программы (такие, как интерпретатор продукций) непосредственно ассемблерным кодом, что повышает эффективность программ. После того как построение алгоритма нового слова завершено точкой с запятой (т.е. высокоуровневыми средствами определения слова), оно может быть переведено на язык ассемблера и включено в программу словом CODE. Таким образом, в Форте имеется весьма широкий спектр возможностей для представления алгоритма на разных уровнях абстракции - от языка ассемблера до проблемно-ориентированных языков, построенных путем определения необходимых слов.

Два других стандартных определяющих слова служат для создания переменных и констант и аналогичны операторам описания типов данных в традиционных языках программирования. Слово VARIABLE (переменная) создает новую переменную, его синтаксис таков:

VARIABLE имя_переменной

т.е. имя переменной следует за словом VARIABLE. При выполнении этого слова в памяти резервируется область (объемом обычно в 16 бит) для хранения значения переменной с указанным именем. При обращении к переменной исполняющая процедура возвращает

в качестве результата адрес зарезервированной области, который затем может быть использован для считывания или записи. В Форте эти операции обозначаются соответственно символами @ (произносится "разыменовать") и ! (произносится "присвоить").

Аналогично константы определяются с помощью слова CONSTANT. Их значения относятся к классу "только для считывания" и задаются в виде

значение CONSTANT имя

где значение, предшествующее слову CONSTANT, является числом. При использовании уже заданной константы реализующая процедура возвращает ее значение. Таким образом, даже типы данных при обращении к ним инициируют некоторые действия, превращая тем самым Форт в полностью процедуральный язык - все его слова следует рассматривать как процедуры.

ПЕРЕДАЧА ДАННЫХ ЧЕРЕЗ СТЕК

Константы и переменные при их инициировании возвращают в качестве результата соответственно числа и адреса областей. Но куда при этом записывается результат? В языке Форт имеется стек данных (или стек параметров), через который слова пересылают друг другу данные. Сам по себе стек - довольно распространенная структура данных. Он широко используется в программных системах фирмы Хьюлетт-Паккард. Данные можно "втолкнуть" в стек, а затем "вытолкнуть" из него, соблюдая определенный порядок: последним вошел, первым вышел. Например, если в стек данных Форт-системы втолкнуть числа 1, 2, 3, а затем произвести три выталкивания, то извлеченные из стека числа будут находиться в таком порядке: 3, 2, 1. Для того чтобы убедиться в этом на практике, выполните слово-команду, которое выталкивает число, находящееся на вершине стека, для чисел натурального ряда. Тогда после ввода с дисплея чисел

```
1 2 3 . . .
```

вы получите их обратно в следующем порядке:

```
. . . 3 2 1 ok
```

При записи результатов операций над стеком мы будем придерживаться определенного правила, позволяющего проследить, как выполнение слова изменяет содержимое стека (так называемая стековая нотация). В этой системе обозначений числа, находящиеся в стеке, располагаются в строчку слева направо, причем крайнее справа - число с вершины стека. Тогда результат действий,

инициированных словом, может быть представлен в виде состояний стека до и после их выполнения. Два состояния разделяются либо с помощью знака "-" (тире), либо направленной вправо стрелкой. Например, слово . (точка) удаляет элемент, находящийся на вершине стека, и печатает его в виде числа со стрелкой:

```
( n ==> )
```

где n - число, а отсутствие символа справа от стрелки означает, что вершина стека пуста (число оттуда удалено). Введенная форма записи может быть также проиллюстрирована на примере слова + , выполняющего сложение двух чисел n1 и n2, которые находятся на вершине стека. Результатом является число n3, помещаемое на стек:

```
( n1 n2 ==> n3 )
```

При помощи стековой нотации можно проследить последовательность изменений стека в процессе выполнения слов, составляющих Форт-программу.

ИЕРАРХИЧЕСКАЯ ДЕКОМПОЗИЦИЯ И РАЗБИЕНИЕ НА МОДУЛИ

Необходимость помнить последовательность состояний стека представляется малопривлекательной для программиста. С одной стороны, Форт позволяет увидеть, что происходит в программе на уровне ее ассемблерной реализации, но с другой стороны, программирование не в содержательных терминах, а в трудно интерпретируемых кодах, близких к машинным, может показаться чрезвычайно сложным. Этот недостаток Форта в значительной степени компенсируется наличием средств для определения новых слов, позволяющих освободиться от необходимости программировать в кодах путем создания высокоуровневых команд требуемой проблемной ориентации, и тем самым перейти к решению задачи на более высоком уровне абстракции. Однако операции низкого уровня хотя и неявно, но все же будут присутствовать в программах, реализующих ваш алгоритм. Языки, в которых вместо стека используются локальные переменные, конечно, свободны от такого недостатка, но в свою очередь не позволяют программисту вносить коррективы в машинную версию программы. В Форте же, напротив, существует возможность опуститься на самый низкий уровень выполнения программы, для чего и предназначено слово CODE. Иными словами, команды уровня реализации в языке Форт являются доступными для программиста. (Локальные переменные тоже могут применяться в Форте, но здесь они не рассматриваются). Явно задавая команды, выполняющие операции над стеком, вы мо-

жете непосредственно следить за ходом работы программы. Особое значение это свойство приобретает для рекурсивных программ, которые вызывают сами себя и передают через стек аргументы самим себе.

Форт поддерживает *функциональный* стиль программирования, т.е. каждое слово выполняется как функция: получает аргументы из стека, выполняет над ними соответствующие действия и засылает результаты обратно в стек. Локализация обработки данных на стеке предотвращает возможность возникновения побочного эффекта, состоящего в модификации глобальных переменных. Каждое слово представляется в виде корректно построенного, замкнутого модуля, взаимодействие которого с другими программами легко проследить.

Модульность значительно упрощает поиск ошибок в программах (т.е. процесс их отладки), поскольку в этом случае программа разбивается на несколько автономных модулей, которые могут тестироваться параллельно. Декомпозиция программы приводит к образованию модулей различного уровня абстрактности, причем при движении вниз от уровня к уровню сложность модулей уменьшается. Множество уровней абстракции образует иерархию. На самом нижнем уровне Форта находится его ядро - небольшой набор программ, написанных в машинном коде, с помощью которых строятся все слова более высоких уровней. Затем на основе стандартных слов, которые и определяют язык Форт, строятся слова, отражающие специфику конкретного приложения. Тем самым осуществляется переход с нижнего уровня, где слова реализованы в машинном коде, на более высокий уровень. Все множество слов Форта можно разбить на несколько уровней абстракции. Характеристики слов, относящихся к каждому из них, приведены на рис. 6.1.

Слова всех уровней кроме первого составляют собственно язык Форт. Нижний уровень занимают машинно-зависимые слова, образующие ядро. К следующему уровню относятся слова для управления периферийными устройствами. Они реализуют обращение из Форт-системы к клавиатуре, дисплею и дисководам компьютера. Третий уровень - уровень интерпретатора. Он включает в себя слова, с помощью которых пользователь управляет интерпретатором Форта. Верхним является уровень компилятора. К этому уровню относятся управляющие конструкции языка Форт и определяющие слова, предназначенные для создания новых слов и включения их в словарь Форт-системы. Перечень слов, входящих в стандарт языка Форт-83, приведен на рис. 6.2. Наиболее часто употребляемые из них будут далее описаны. Функциональные определения слов можно найти в приложении Б.

ВЫПОЛНЕНИЕ АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ

Как вы видели, в Форте аргументы передаются через стек, и слова "+" и "." выполняют действия над элементами стека. 16-битовое представление целого числа является стандартным для

Уровень	Описание
Уровень приложений языка	Скомпилированные слова Форта, хранящиеся в словаре системы. Предназначены для программирования прикладных задач пользователя
Уровень компилятора	Слова, предназначенные для создания новых слов (определяющие слова), их компиляции и записи в словарь
Уровень интерпретатора	Слова, поддерживающие процесс интерпретации программ, введенных с клавиатуры или считанных с диска
Уровень управления периферийными устройствами	Слова, реализующие доступ к устройствам ввода-вывода: * клавиатуре и дисплею * дисководам
Уровень машинной реализации (ядро системы)	Слова, написанные в машинном коде, которые управляют выполнением основных операций виртуальной Форт-машины. Сюда входят: * слова периода выполнения (т.е. реализующие процедуры) для определяющих и компилирующих слов * внутренний (адресный) интерпретатор

Рис. 6.1. Уровни слов Форта

реализаций Форта, ориентированных на 8-разрядный компьютер; однако в компьютере с 32-разрядным процессором область памяти для хранения числа (так называемая ячейка) занимает 32 бита. Несмотря на то что при реализации Форта всегда ориентируются на некоторый постоянный размер ячейки, сама по себе ее величина не влияет на определения базовых функций языка. Какой бы ни была величина ячейки памяти, именно она определяет количество

разрядов, отводимых для хранения чисел одинарной точности и машинных адресов.

Уровень машинной реализации

!	2+	CMOVE>	MOD
*	2-	COUNT	NEGATE
*/	2/	D+	NOT
*/MOD	<	D<	OR
+	=	DEPTH	OVER
+!	>	DNEGATE	PICK
-	>R	DROP	R>
/	?DUP	DEXECUTE	R@
/MOD	@	EXIT	ROLL
0<	ABS	FILL	ROT
0=	AND	I	SWAP
0>	C!	J	U<
1+	C@	MAX	UM*
1-	CMOVE	MIN	UM/MOD
XOR			

Уровень управления периферийными устройствами

BLOCK	KEY	BUFFER	SAVE-BUFFER
CR	SPACE	EMIT	SPACES
EXPECT	TYPE	FLUSH	UPDATE

Уровень интерпретатора

#	<#	FIND	PAD
#>	>BODY	FORGET	QUIT
#S	>IN	FORTH	SIGN
#TIB	ABORT	FORTH-83	SPAN
'	BASE	HERE	TIB
(	BLK	HOLD	U.
-TRAILING	CONVERT	LOAD	WORD
.	DECIMAL	.(	DEFINITION

Уровень компилятора

+LOOP	,	."	:
;	DO	LOOP	VOCABULARY
ABORT"	DOES>	REPEAT	WHILE
ALLOT	ELSE	STATE	[
BEGIN	IF	THEN	[']
COMPILE	IMMEDIATE	UNTIL	[COMPILE]
CONSTANT	LEAVE	VARIABLE	]
CREATE	LITERAL		

Рис. 6.2. Слова, предусмотренные стандартом Форт-83

В Форте допустимы числа не только одинарной, но и двойной точности. Арифметические операции над числами с двойной точностью не применяются при обработке символьной информации и поэтому здесь не рассматриваются; их подробное описание можно найти в литературе, ссылки на которую имеются в библиографии.

Форт включает в себя четыре основных арифметических операции и несколько их вариаций. Операция вычитания имеет вид:

- (n1 n2 ==> n3)	Вычитание n2 из n1 и занесение результата n3 на стек.

Умножение и деление представляются так:

* (n1 n2 ==> n3)	Умножение n1 на n2 и занесение результата n3 в стек.
/ (n1 n2 ==> n3)	Деление n1 на n2 и занесение результата n3 в стек.

Если нужно получить остаток от деления, то выполняется следующая операция:

/MOD (n1 n2 ==> rq)	Деление n1 на n2 с получением остатка r и частного q.

Быстрое умножение или деление на 2 позволяют осуществить такие команды:

2* (n1 ==> n2)	Значением n2 является n1, умноженное на 2.
2/ (n1 ==> n2)	Значением n2 является n1, деленное на 2.

Результат не округляется - последние цифры просто отбрасываются.

Часто также применяются слова, которые увеличивают или уменьшают число на единицу или на два:

1+ (n1 ==> n2)	Увеличение n1 на единицу.
2+ (n1 ==> n2)	Увеличение n1 на два.
1- (n1 ==> n2)	Уменьшение n1 на единицу.

2- (n1 ==> n2) Уменьшение n2 на два.

Например, в результате деления 60 на 10 и уменьшения частного на единицу, т.е. выполнения слов

60 10 / 1-

получим число 5. Поскольку в качестве чисел в Форте используются целые со знаком, имеется слово для получения значения с противоположным знаком:

NEGATE (n1 ==> n2) Результатом является n2 = -n1.

Кроме приведенных выше основных арифметических операций, в Форте предусмотрены слова для нахождения наибольшего и наименьшего из двух чисел:

MIN (n1 n2 ==> n3). Значением n3 является наименьшее из чисел n1 и n2.

MAX (n1 n2 ==> n3) Значением n3 является наибольшее из чисел n1 и n2.

Здесь по умолчанию принято, что числа n1 и n2 имеют знак. Для получения абсолютного значения числа служит слово:

ABS (n1 ==> n2) Значением n2 является |n1| - абсолютное значение числа n1.

Форт располагает и другими словами, выполняющими арифметические операции, но и перечисленных здесь вполне достаточно для понимания программ, приведенных далее в книге.

На уровне машинной реализации арифметические операции Форта выполняются над числами в двоичном представлении (с основанием 2), однако имеется возможность вводить и выводить числа, представленные в других системах счисления, отличных от двоичной и десятичной. Основание (или модуль) для представления числа задается значением системной переменной BASE. Если ее значение равно 10 (что имеет место при инициализации Форт-системы), то арифметические операции выполняются в десятичной системе счисления. Для изменения основания на 16, т.е. перехода в шестнадцатиричную систему счисления , предусмотрено слово HEX, а для обратного перехода в десятичную систему - слово DECIMAL. Например, перевод числа 20 из десятичной системы в шестнадцатиричную записывается так:

20 HEX . 14

где 14 - шестнадцатиричное представление десятичного числа 20. Число, записанное в шестнадцатиричной системе счисления, помечается предшествующим знаком доллара ($). Например, число 20 в шестнадцатиричной системе счисления будет выглядеть как $14.

МАНИПУЛИРОВАНИЕ ЭЛЕМЕНТАМИ СТЕКА

Если вы начнете составлять пограмму на Форте, то очень скоро почувствуете, что вам требуются слова, позволяющие производить различные действия с элементами стека. Представьте, например, что при делении 60 на 11 вам понадобилось сохранить остаток, а не частное. Слово /MOD засылает остаток в стек вторым элементом, а брать из стека можно только верхний элемент. Для удаления с вершины стека частного и перемещения туда остатка в Форте имеется слово DROP:

DROP (n1 ==>) Удаление верхнего элемента стека.

Удалить с вершины стека два элемента можно с помощью слова

2DROP (n1 ==>) Удаление двух верхних элементов стека.

Иногда требуется скопировать верхний элемент стека. Это позволяет сделать слово:

DUP (n1 ==> n1 n1) Создание дубля верхнего элемента стека.

Дублирование двух верхних элементов стека выполняется словом:

2DUP (n1 n2 ==> n1 n2 n1 n2) Создание дублей двух верхних элементов стека.

Обратите внимание, что слово 2DROP приводит именно к такому результату, а не к (n1 ==> n1 n1 n1), т.е. эта команда дублирует два элемента стека,а не записывает один элемент дважды.

Иногда может потребоваться создать копию второго элемента стека и поместить его на вершину. Это можно сделать с помощью слова:

OVER (n1 n2 ==> n1 n2 n1) Создание копии второго элемента и помещение его на вершину стека.

Для того чтобы получить доступ ко второму элементу, можно не создавать его копию, а просто поменять два верхних элемента стека местами, это действие выполняет слово SWAP:

SWAP (n1 n2 ==> n2 n1)	Меняет местами два верхних элемента стека.

Теперь покажем, как с помощью введенных слов манипулирования элементами стека получить доступ к остатку от деления, выполненного словом /MOD. Если полученный остаток уже находится на вершине стека, то для его снятия нужно обратиться к слову DROP, а затем переместить остаток на вершину словом SWAP, т.е. нужно выполнить такую последовательность команд:

```
60 11 /MOD SWAP DROP
```

Выполнение комбинации слов SWAP DROP приводит к выталкиванию из стека его второго элемента, поэтому данную комбинацию целесообразно оформить как новое слово Форта:

```
: NIP SWAP DROP ;
```

В дальнейшем мы будем использовать это новое слово, хотя оно и не относится к числу стандартных. Определив слово NIP, мы тем самым расширили множество слов, которые можно применять при составлении программ.

До сих пор мы имели дело со словами, которые обращаются к двум верхним элементам стека. Рассмотрим теперь слова, которые выполняют действия с тремя верхними элементами (в словах PICK и ROLL количество аргументов не ограничивается). Слово ROT передвигает третий элемент на вершину стека, т.е. производит ротацию или циклическую перестановку трех верхних элементов:

ROT (n1 n2 n3 ==> n2 n3 n1)	Перестановка n1 на вершину стека.

Действие, обратное этому, выполняется словом -ROT:

-ROT (n1 n2 n3 ==> n3 n1 n2)	Перемещение верхнего элемента на третье место в стеке.

Наконец, иногда может оказаться полезной условная операция над стеком:

?DUP (n1 ==> |n1| n1)	Создание копии верхнего элемента стека, если им является не ноль.

С помощью этих слов можно выполнять со стеком все операции, которые потребуются для программирования задач искусственного интеллекта.

ДОСТУП К ДАННЫМ

Для чтения и записи (т.е. выборки и запоминания) данных в памяти компьютера необходимы слова, реализующие операции доступа к ней. Форт предоставляет возможность обращаться к памяти, используя непосредственно внутренние адреса, аналогично операторам PEEK и POKE в Бейсике. Для доступа к данным, в частности к переменным, используются слова, которые считывают и запоминают их значения. Первое из них имеет такой синтаксис:

@ (addr ==> n)	Заносит на стек число, находящееся в памяти по адресу addr (произносится "разыменовать").

Синтаксис второго слова:

! (n addr ==>)	Записывает число n в область памяти, начинающуюся с адреса addr (произносится "присвоить").

В 8-разрядном компьютере числа Форта занимают 16 бит и под каждое из них отводится два машинных слова. Порядок расположения байтов (т.е. где находятся старшие разряды числа - по адресу addr или addr+1) зависит от типа компьютера.

Введя слова доступа к памяти, можно выполнять действия с переменными. Например, для перевода чисел в шестнадцатиричную систему счисления необходимо переменной BASE присвоить новое значение, равное 16. Это можно сделать следующим образом:

```
16 BASE !
```

Однако если мы теперь разыменуем переменную BASE командой

```
BASE @
```

и напечатаем полученное значение, то все равно получим число 10. (Почему?).

Иногда бывает необходимо выбрать значение области памяти величиной в один байт, а не считывать все 16 бит, отведенных под число. Это бывает нужно, например, для извлечения из памяти символа, который в коде ASCII занимает один байт. Доступ к символу осуществляется словами:

C@ (addr ==> n) — Замена адреса его содержимым (произносится "с-разыменовать"). Значение старшего разряда числа, находящегося на вершине стека, равно 0 .

C! (n addr ==>) — Запись младшего разряда числа по адресу addr (произносится "с-присвоить").

Кроме указанных слов общего назначения в Форте имеются слова для выполнения специализированных функций. Изменение значения адреса производится словом

+! (n addr ==>) — Добавление значения n к адресу addr (произносится "плюс-присвоить").

Слово +! увеличивает значение адреса и может использоваться при реализации цикла для увеличения на очередном шаге его счетчика. Два других слова не входят в число стандартных слов Форта, но тем не менее весьма полезны. Первое из них

ON (addr ==>) — Запись значения флага "*истина*" по адресу addr.

Вторым является слово

OFF (addr ==>) — Запись значения флага *ложь* по адресу addr.

ФЛАГИ, ЛОГИЧЕСКИЕ ОПЕРАТОРЫ И СРАВНЕНИЕ ЧИСЕЛ

В Форте-83 флаги (признаки) используются для фиксации результатов проверок и принимают значения *да/нет* или *истина/ложь*. Значение флага *истина* представляется числом -1, а значение *ложь* - числом 0. Выбор для кодирования значения *истина* числа -1, а не 1 объясняется тем, что в машинной реализации числа -1 во всех битах стоят единицы. (В Форте арифметические операции выполняются над числами, представленными в двоичной системе счисления). В машинной реализации значения *ложь* во всех битах находятся нули. В связи с этим в Форте оказалось легко реализовать слово NOT, которое изменяет значение *истина* на значение *ложь* и обратно. NOT - это булевская операция, кото-

рая выполняет логическое отрицание путем изменения значений всех битов числа на противоположные:

NOT (n1 ==> n2) — Значением n2 является логическое дополнение значения n1.

Для любого бита в представлении значения n1 верно следующее: если в нем находится 1, то станет 0; если же в этом бите стоит 0, то он будет заменен на 1. Точно так же изменяются машинные представления значений флагов и при выполнении следующих логических операций:

AND (n1 n2 ==> n3) — Значением n3 является логическая конъюнкция n1 и n2. В каждом бите машинного представления n3 будет находиться 1 только в том случае, если в обоих соответствующих битах n1 и n2 также находятся единицы.

OR (n1 n2 ==> n3) — Значением n3 является логическая дизъюнкция n1 и n2. В каждом бите машинного представления n3 будет находиться 0 только в том случае, если в обоих соответствующих битах n1 и n2 также находится 0.

XOR (n1 n2 ==> n3) — Значением n3 является результат логической операции "исключающее или". В каждом бите машинного представления n3 будет находиться 1 только в том случае, если в соответствующих битах n1 и n2 находятся противоположные значения.

Введенные логические операции являются полезными при составлении комбинаций признаков. Например, если потребуется флаг, который истинен только при условии, что истинны оба флага flag1 и flag2, то его можно образовать, соединив последние логической связкой AND.

При составлении программ необходимы средства для помещения значений флагов в стек. Вместо чисел 0 и -1, выражающих значения флагов, лучше ввести средствами Форта логические

константы TRUE (истина) и FALSE (ложь), которые соответствуют этим числам, - тогда программа будет легче читаться.

Слова, которые возвращают значения флагов, называются *булевскими операциями*, поскольку они принимают только два значения - *истина* и *ложь*. В Форте имеется несколько булевских операций, позволяющих проводить сравнение чисел. Наиболее распространенными среди них являются следующие:

= (n1 n2 ==> f)	Если n1 = n2, то f - истинно (произносится "равно").
< (n1 n2 ==> f)	Если n1 < n2, то f - истинно (произносится "меньше, чем").
> (n1 n2 ==> f)	Если n1 > n2, то f -истинно (произносится "больше, чем").
<> (n1 n2 ==> f)	Если n1 не равно n2, то f -истинно (произносится "не равно").

Операции, которые сравнивают число с нулем, реализуются следующими словами Форта:

0 = (n1 ==> f)	Значением f является *истина*, если n1 равно 0.
0 < (n1 ==> f)	Значением f является *истина*, если n1 меньше 0.
0 > (n1 ==> f)	Значением f является *истина*, если n1 больше 0.

Флаги, вырабатываемые булевскими операциями, используются управляющими словами для того, чтобы направлять вычисления в программе по той или иной ветви.

УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ

Итак, вы познакомились с арифметическими операциями Форта, с действиями над элементами стека, доступом к данным, находящимся в памяти, а также определением логических значений флагов и их комбинаций. С помощью слов, выполняющих эти функции, можно составлять процедуры в виде определения через двоеточие. Однако для написания серьезных программ необходимы

управляющие слова, которые позволяют принимать решения о том, какой оператор будет выполняться следующим. Форт базируется на управляющих конструкциях *структурного программирования*. Слова Форта состоят из фрагментов, оформленных в виде модулей, каждый из которых получает и передает управление через свои точки входа и выхода.

На рис. 6.3 приведен пример неструктурированной программы, а на рис. 6.4 для сравнения показаны управляющие конструкции языка Форт, представленные в виде отдельных модулей. Обратите внимание на то, что каждая конструкция имеет единственный вход и единственный выход. (Из этого правила имеются только два исключения - оператор LEAVE (выйти) в цикле DO, который мы не будем использовать, и прерывание программы из-за ошибки. Оператора GOTO в Форте нет.)

На рис. 6.4A и 6.4B проиллюстрированы основные способы ветвления программ. Конструкция ELSE может быть опущена. Приведенные структуры в обозначениях языка Форт записываются в виде следующих выражений:

IF A THEN или IF A ELSE B THEN

Оператор IF выполняет ветвление программы и производит над стеком действие (f ==>). Если снятый с вершины стека признак (флаг) является истинным, то после оператора IF выполняется группа команд A, в противном случае начинает выполняться группа команд B, стоящая за оператором ELSE (или следующий за IF оператор, если ветвь ELSE отсутствует). После завершения выполнения группы команд B управление передается оператору, стоящему за оператором THEN.

В конструкции цикла BEGIN-UNTIL (рис. 6.4C) значение флага проверяется после каждого выполнения группы команд A. Слово UNTIL, как и слово IF, снимает значение флага с вершины стека и производит проверку его истинности. Если этим значением является *ложь*, то группа команд A повторяется. Другими словами, выполняются те же действия, что и в конструкции ELSE оператора ветвления IF при ложном значении флага.

На рис. 6.4D представлена схема цикла WHILE. Повторное выполнение группы команд, находящихся внутри него, также происходит при значении флага *ложь*. Цикл BEGIN-WHILE начинается с выполнения группы команд A, следующих за оператором BEGIN. Затем управление получает оператор WHILE, который снимает значение флага со стека; если флаг истинен, то выполняется группа команд B, которая завершается оператором REPEAT (повторить). Этот оператор передает управление обратно в начало цикла, т.е. оператору BEGIN. В терминах языка Форт эта конструкция записывается следующим образом:

BEGIN A WHILE B REPEAT

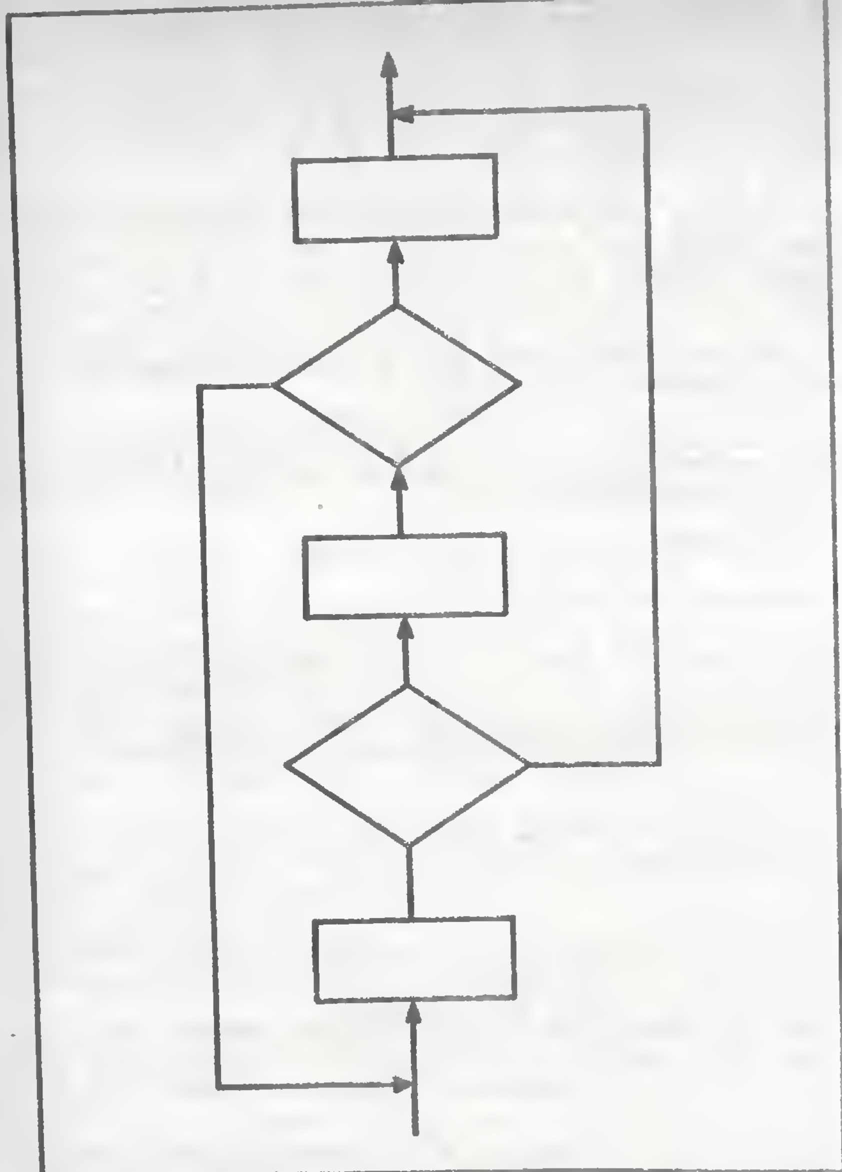


Рис. 6.3. Пример неструктурированной программы. Циклы образуют перекрывающиеся куски программы, т. е. каждый цикл расположен частично внутри, частично снаружи другого цикла

Флаг
Нет
Значение флага — ИСТИНА?
Да
IF
A
THEN
(A)

Флаг
Нет
Значение флага — ИСТИНА?
Да
ELSE
IF
B
A
THEN
(B)

BEGIN
A
Флаг
Нет
Значение флага — ИСТИНА?
Да
UNTIL
(C)

BEGIN
A
Флаг
Значение флага — ИСТИНА?
Нет
Да
WHILE
B
REPEAT
(D)

DO
Запомнить начальное и конечное значения счетчика цикла в стеке возвратов
A
Увеличить счетчик цикла и сравнить с коначным значением
Да
Счетчик цикла < конечного значения ?
Нет
LOOP
+LOOP
(E)

Рис. 6.4. Управляющие конструкции языка Форт. (A) IF-THEN; (B) IF - ELSE - THEN; (C) BEGIN - UNTIL; (D) BEGIN - WHILE; (E) DO - LOOP или DO - +LOOP

В управляющих конструкциях IF, UNTIL, WHILE условием выполнения цикла является ложное значение флага. Поскольку действия этих операторов похожи друг на друга, то в их машинной реализации имеются общие подпрограммы периода выполнения. Если управляющее слово встречается в определении через двоеточие (в период компиляции), то это приводит к компиляции фрагмента, реализующего передачу управления в объектный код, который впоследствии будет выполняться при каждом обращении к этому слову. Детали реализации управляющих слов будут рассмотрены позже при описании компилирующих слов.

В тех случаях, когда количество повторных шагов цикла известно еще до начала его выполнения, полезной может оказаться такая конструкция, как цикл со счетчиком. В Бейсике такой оператор имеет вид FOR - NEXT, а в Форте к этому типу относятся два оператора: DO - LOOP и DO - +LOOP. Их схемы приведены на рис. 6.4E. Слово DO берет со стека два аргумента. Затем командой

```
DO ( граничное_значение счетчик_цикла ==> )
```

счетчик цикла и граничное значение помещаются в стек возврата и удаляются из стека параметров. *Счетчик цикла* получает свое начальное значение, а *граничное значение* становится на единицу больше конечного значения счетчика цикла. В описании этого слова упоминается название еще одного стека языка Форт - стека возвратов. Он так называется потому, что используется в Форте в основном для запоминания того места в программе, куда нужно вернуться после завершения встретившегося слова. Это действие выполняется виртуальной Форт-машиной, которая будет рассмотрена позже. Циклы типа DO также обращаются к этой машине, поскольку все конструкции цикла должны находиться в границах одного (произвольного) слова Форта. DO-цикл использует стек возвратов только до тех пор, пока он находится внутри последовательности команд, составляющих определение нового слова (т.е. пока не встретится слово ;).

Слово DO передает значения двух счетчиков из стека параметров в стек возвратов. Значением счетчика является начальная величина, передаваемая счетчику цикла, а граничное значение превосходит на единицу конечное значение. Это объясняется тем, что в цикле LOOP увеличение счетчика предшествует его сравнению с граничным значением. Если они оказываются равны, то выполнение цикла завершается. На каждом шаге цикла типа LOOP счетчик увеличивается на единицу.

Если требуется использовать шаг, отличный от единицы (причем это могут быть и отрицательные числовые значения), то следует обратиться к циклу иного типа, реализуемому словом +LOOP. В отличие от цикла LOOP, который не взаимодействует со стеком параметров, +LOOP берет размер шага (т.е. величину, на которую увеличивается параметр цикла при каждом очередном вы-

полнении тела цикла) из этого стека. Чтобы иметь доступ к значению счетчика в теле цикла, необходимо командой I предварительно занести его в стек параметров.

С помощью управляющих слов, которые мы ввели, можно построить примеры определения новых слов языка Форт. Так, для вывода на печать чисел от 1 до n следует прибегнуть к помощи цикла DO:

```
( n ==> )
: .#S 1+ 1 DO I . LOOP ;
```

Считается хорошим стилем программирования помещать перед определением слова его стековую нотацию. Это особенно уместно, если исходные тексты Форт-программы должны быть сохранены на дискете. Имя определяемого слова (в данном случае это #S) располагается непосредственно за двоеточием через пробел. Затем идет определение слова, состоящее из слов, с помощью которых числа будут отпечатаны.

Так как мы предполагаем, что числа передаются через стек, то слово +1 устанавливает граничное значение, превышающее последнее выводимое на принтер число на единицу (поскольку последнее число тоже должно быть отпечатано). Первым числом является единица, поэтому начальное значение счетчика цикла полагается равным единице. Это необходимо сделать до начала цикла DO, после чего стек будет содержать оба значения счетчика, которые необходимы для выполнения цикла. Слово I, находящееся в теле цикла, помещает значение счетчика на стек, а команда . (точка) печатает его, одновременно удаляя со стека. Слово LOOP завершает цикл. Если ваш компьютер оснащен Форт-системой, попробуйте выполнить эту программу для n=10. При наборе команды

```
10 . #S
```

вы получите в качестве ответа следующую последовательность:

```
1 2 3 4 5 6 7 8 9 10 ok
```

Тот же самый результат может быть получен с помощью другой управляющей конструкции, отличной от цикла DO. Так, в терминах цикла BEGIN-UNTIL упомянутая программа будет выглядеть следующим образом:

```
( n ==> )
: .#S 1+ 1 BEGIN DUP . 1+ 2DUP = UNTIL 2DROP ;
```

Определение слова, более длинное по сравнению с предыдущим, свидетельствует о том, что в нашем случае лучше использовать цикл DO. В теле цикла на стек помещаются два числа. Ко-

манда 2DUP делает копии значения счетчика и граничного значения для выполнения команды = , которая их сравнивает, вырабатывает значение флага и передает его команде UNTIL. При выходе из цикла эти значения все еще находятся на стеке, откуда их удаляет слово 2DROP. Чтобы представить последовательность действий, произведенных над стеком, следует аккуратно выписать его состояния под каждым словом - так будет виден результат их выполнения. Другие примеры использования управляющих слов можно увидеть на экранах с 40 по 45 и с 60 по 65, приведенных в гл. 7 и 9 соответственно, а также в приложении А.

Нестандартной, но весьма популярной управляющей конструкцией является слово CASE, включающее в свою очередь слова OF, ENDOF и ENDCASE. Это слово служит обобщением конструкции IF-THEN, поскольку допускает неограниченное количество ветвлений. Каждая ветвь заключается в символьные скобки OF и ENDOF. Приведем пример конструкции CASE:

```
( n ==> )


CASE 1
OF A ENDOF2
OF B ENDOF3
OF C ENDOF
D
ENDCASE
```

Слово CASE берет элемент с вершины стека (часто таким элементом является символьное значение) и сравнивает его с тем элементом, который был помещен на стек до обращения к слову OF. В данном примере n сравнивается с единицей. Если они оказываются равными, то n удаляется из стека и выполняется последовательность команд A. Затем слово ENDOF передаст управление команде, размещенной за словом ENDCASE. При n, не равном единице, следующее слово OF сравнивает его с числом 2, и т.д. Если не удовлетворено ни одно из этих условий, то выполняется последовательность команд D, а n остается на стеке. Наконец, после завершения группы команд D слово ENDCASE удаляет n со стека. Особенность применения этого оператора состоит в том, что n часто включается в последовательность команд D. В таком случае фрагмент D должен оставить на стеке какое-либо число, чтобы слову ENDCASE было что удалить с него. Конструкция CASE используется в определении слова ЧТСП, исходный текст которого приведен на экране 46 (см. гл. 7 и приложение A).

СТЕК ВОЗВРАТОВ

Как вы видели, на стеке возвратов находятся счетчики циклов. Однако в некоторых случаях стек возвратов может выполнять и другие функции. Передача данных через стек параметров иногда бывает сопряжена с определенными трудностями, и возможность временно поместить данные в какое-либо другое место позволяет существенно упростить процесс составления программы. Конечно, для этой цели можно было бы объявить соответствующие переменные и поместить данные в них, но в Форт-программах не рекомендуется вводить переменные только для временного хранения каких-либо значений.

В Форте имеются три слова, с помощью которых осуществляется доступ к стеку возвратов из стека параметров (помимо команды I, обращающейся к счетчику цикла):

>R (n ==>)	Снимает число с вершины стека параметров и помещает его на стек возвратов (произносится "на R").
R> (==> n)	Снимает число с вершины стека возвратов и помещает его на стек параметров (произносится "с R").
R@ (==> n)	Помещает копию элемента, находящегося на вершине стека возвратов, на стек параметров (произносится "разыменовать R").

Над стеком возвратов выполняются те же действия, что и над стеком параметров. Например, ввод последовательности слов

```
1 2 3 >R >R >R R@ . R> . R> . R> .
```

приведет к следующему результату: 1 1 2 3 ok
Число 3 первым было помещено на стек и последним снято с него.

При использовании указанных слов в циклах DO на них накладываются ограничения. Поскольку слова LOOP и +LOOP обращаются к стеку возвратов для выполнения действий со счетчиком цикла, любое обращение к этим словам в пределах DO-цикла должно возвращать стек в его исходное состояние, т.е. в то, которое он имел прежде, чем в программе встретилось слово начала цикла. Иллюстрацией изложенного может служить такой пример:

```
... >R ... DO ... R> ...
```

Здесь некоторый элемент помещается на стек возвратов для того, чтобы в последующем быть использованным в цикле DO. Однако слово R> вместо этого извлекает значение счетчика цикла, поскольку DO поместило на стек возвратов два элемента. Таким образом, передача данных в цикл DO через стек возвратов оказывается не всегда возможной. В подобных случаях требуемый результат может быть получен путем выбора из стека возвратов третьего элемента, но лучше все-таки применять другие средства - более простые и надежные.

ОБРАБОТКА СТРОК

Одной из основных функций компьютера является обработка текстовой информации. Компьютерные программы, написанные на одном из языков программирования, представляют собой тексты. Программы в исходном виде, обрабатываемые интерпретатором или компилятором Форта, - по существу, тоже тексты. Текст состоит из символов (в Форте они кодируются с помощью кода ASCII), которые объединяются в последовательности, называемые *символьными строками.*

В Форте строки выполняют различные функции. Так, имена слов из словаря имеют вид строк. Строка - это структурированный тип данных, представляющий собой последовательность однобайтовых символов. В начале строки находится счетчик, занимающий также один байт, в котором располагается число, указывающее количество символов в строке. Значение счетчика называется *длиной строки.* Например, строка "ABC" в памяти компьютера будет представлена последовательностью чисел: 3 65 66 67, где числа с 65 по 67 - десятичные ASCII-коды букв A, B и C соответственно.

Строки могут быть представлены на стеке двумя способами: с использованием указателя на строку (адреса строки) или (более общий способ) указателя на первый элемент строки, включая ее счетчик, что в стековой системе обозначений записывается как:

(a u ==>)

где a - адрес, a u - счетчик. Мы и в дальнейшем в стековой нотации будем пользоваться этими обозначениями. Если применяется указатель на строку, то перед ним ставится знак ^. Таким образом, с помощью ^a обозначается указатель на байт счетчика строки. Для перехода от строки со счетчиком к строке с явно заданной длиной служит слово COUNT:

COUNT (^a ==> a u)

Оно преобразует указатель строки в адрес ее первой литеры и значение счетчика. Слово COUNT выбирает значение счетчика и увеличивает ^a на единицу. Имея на стеке строку, ее можно вывести на экран дисплея с помощью слова TYPE:

TYPE(a u ==>)

которое передаст строку на устройство вывода. Иногда может понадобиться вывести на дисплей строку литералов. Например, нам требуется получить на экране дисплея сообщение "ЦЕЛИ:", приведенное на экране 65 в слове TRACE. Применим слово ." :

." (==>)

и оно выдаст на дисплей символы, расположенные между ним и закрывающими кавычками ("), в виде символьной строки. Поскольку ." является словом Форта, то после него должен следовать пробел. За пробелом начинается последовательность символов, завершающаяся двойной кавычкой ("). Кавычки означают конец строки.

В некоторых реализациях Форта строки можно создавать с помощью слова " . В таких случаях строка тоже располагается после пробела и заканчивается двойной кавычкой. Слово " помещает созданную строку во временную память, называемую PAD, и заносит на стек указатель строки со счетчиком.

Ранее уже упоминались слова для задания комментариев -это левая скобка, открывающая комментарий, и правая скобка, закрывающая его. Любой текст, расположенный между левой и правой скобками, транслятором не обрабатывается. Слово \, не относящееся к числу стандартных, обычно используется для переноса без каких-либо изменений строки комментариев из исходного текста программы на диск. Примеры применения слова \ можно найти в исходных текстах многих программ, приведенных во второй части книги.

Некоторые символы используются так часто, что получили статус слов Форта. К их числу относится слово CR - возврат каретки и BL - пропуск, или пробел. Слово CR выполняет возврат назад, а слово BL помещает на стек символ пробела. Слово SPACE передает на устройства вывода пробел.

ПОТОКИ ТЕКСТОВ

Выше мы предполагали, что текст выдается на экран дисплея и вводится в компьютер с клавиатуры или дискеты. Форт-система воспринимает входные и выходные тексты как потоки символов бесконечной длины, называемые соответственно *входным* и *выходным потоками.* Как уже упоминалось ранее, ядро Форта составляют слова для управления аппаратной частью компьютера. К этому же уровню относятся и слова, управляющие устройствами ввода-вывода. Вывод символа в выходной поток осуществляется словом EMIT,

а чтение его из входного потока - словом KEY, причем слово KEY считывает символ из входного потока независимо от того, поступает ли он с клавиатуры или с диска:

KEY (==> c) — Вводит символ из входного потока и помещает его на стек.

Аналогично все символы из выходного потока можно вывести словом EMIT:

EMIT (c ==>) — Передает на выходное устройство символы со стека.

Эти слова задаются определяющим словом DEFER, которое нам пока не встречалось. Оно служит для определения слов, осуществляющих векторные (косвенные) вычисления. Такое слово может быть настроено на выполнение других слов Форта. Например, если выходной поток выводится на экран дисплея, то слово EMIT следует настроить на выполнение слова (EMIT) - драйвер дисплея. Если же выходной поток направляется на устройство печати, то EMIT заставит выполняться слово, управляющее драйвером принтера (PREMIT). Детали реализации слова DEFER мы обсудим после описания структуры слов Форта.

СТРУКТУРА СЛОВ ФОРТА

До сих пор при описании Форта мы не рассматривали структуру представления скомпилированных слов в памяти компьютера. Реализация Форта не стандартизована, поскольку ее эффективность зависит от используемого процессора и типа вычислительной системы. Однако многие реализации похожи друг на друга, и, в частности, реализацию, описанную в книге, можно считать достаточно типичной. На рис. 6.5 приведена структура слов Форта.

Элемент словаря состоит из четырех полей. Их названия расположены в правой части рисунка. Базовые адреса каждого из этих полей также имеют свои наименования, которые показаны на рисунке слева. Первое поле называется *полем связи*. Оно располагается по *адресу поля связи* (lfa). Поле связи содержит указатель на предшествующее слово в словаре. С помощью таких указателей слова из словаря увязываются в единую списковую структуру. В первом слове словаря значением поля связи является ноль, что означает конец списка.

Обратите внимание на то, что на рисунке слово СЛОВО2 связано в обратном направлении с другим полем предшествующего ему слова СЛОВО1. Это поле называется *полем имени* и располагается по *адресу поля имени* (nfa). Поле имени по своей структуре представляется наиболее сложным из полей. Его первый байт тоже

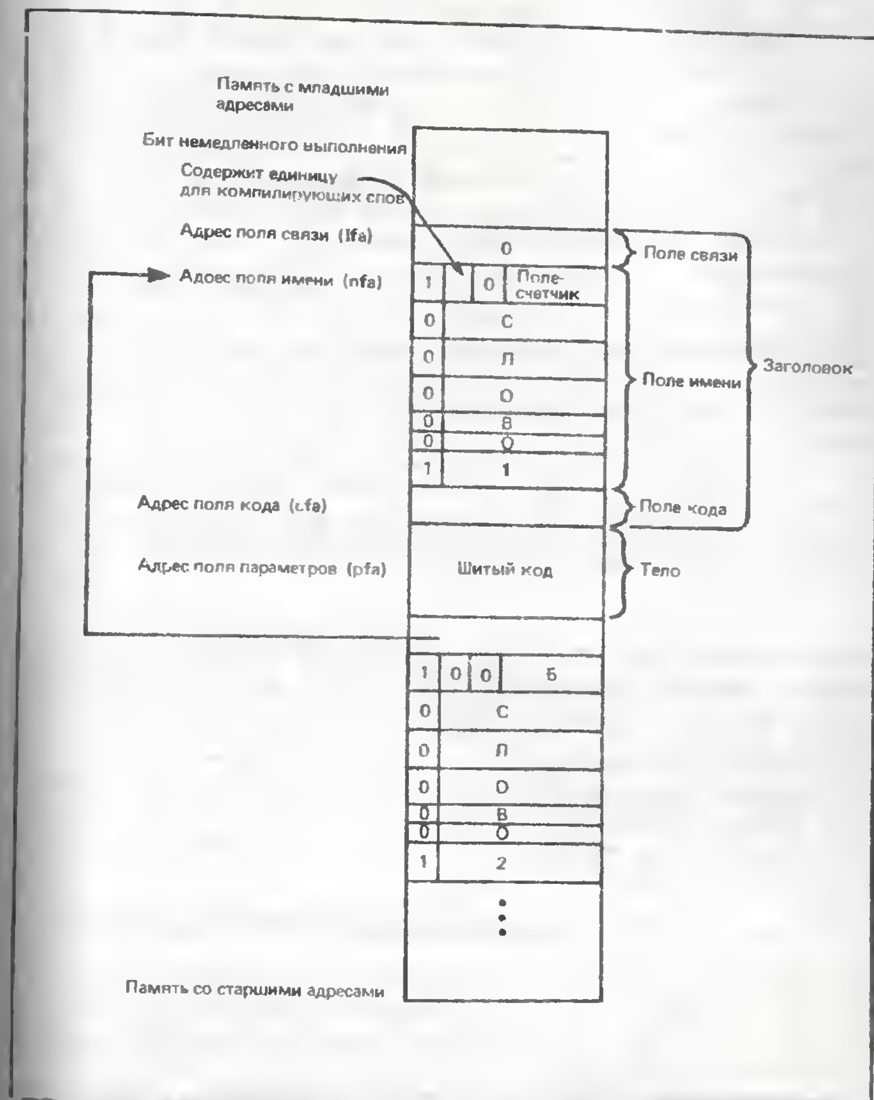


Рис. 6.5. Структура словаря и слов Форта

разбит на поля. Старший бит поля всегда равен единице, что указывает на начало поля имени. Старший бит последнего символа в поле имени также равен единице. Старшие биты остальных символов равны нулю. Эти биты служат признаком продолжения поля имени. Следующий бит первого байта поля имени (бит 6) равен единице, если слово - компилирующее. Более подробные объяснения будут даны позднее. Младшие пять бит первого байта составляют поле счетчика. В нем указывается количество символов в имени слова, т.е. он выполняет ту же функцию, что и счетчик для строки. Таким образом, поле имени - это строка специального вида.
За полем имени следует *поле кода*, расположенное по *адресу поля кода* (cfa). В нем содержится указатель на слово, которое будет выполняться после завершения слова СЛОВО1. Данная процедура периода выполнения действует как интерпретатор слова (детально мы рассмотрим ее ниже). Наконец, последнее поле - *поле параметров*, или *тело* СЛОВА1. Оно находится по *адресу поля параметров* (pfa). Тело содержит данные или программу, определяющую это слово в виде, получившем название "шитого" кода; более подробно шитый код объясняется при описании виртуальной Форт-машины.

Поскольку каждое поле имеет собственное назначение, иногда может потребоваться адрес одного из них. По умолчанию поле кода принято считать определяющим для слова. Для получения cfa служит слово '(одинарная кавычка). Например, команда

' СЛОВО1

вернет cfa СЛОВА1 и поместит его на стек. Приведенные ниже слова позволяют выбрать адрес любого поля. Их назначение поясняет следующая стековая нотация:

```
>BODY  ( cfa ==> pfa)
>NAME  ( cfa ==> nfa)
BODY>  ( pfa ==> cfa)
NAME>  ( nfa ==> cfa)
L>NAME ( lfa ==> nfa)
```

Слово .ID дает возможность выдать на дисплей в виде строки поле имени:

.ID (nfa ==>)	Передает на устройство вывода поле имени в символьном виде.

Компилятором Форта используется переменная LAST, где хранится nfa последнего слова, введенного в словарь, т.е. в ней находится указатель на первый элемент списковой структуры словаря.

УПРАВЛЕНИЕ СЛОВАРЕМ

В Форте имеется несколько слов для управления самим словарем. Для перечисления слов из словаря, начиная с последнего введенного в него слова (указатель на которое находится в переменной LAST), предназначено слово WORDS (или VLIST в некоторых версиях Форта). Это слово, не относящееся к стандартным, дает возможность просмотреть порядок слов в словаре и узнать, сколько слов определено компилятором.

В процессе разработки программ может возникнуть необходимость в удалении некоторых слов из словаря. Заданное слово, а также все слова, находящиеся в списке за ним, могут быть удалены словом FORGET. Для исключения из словаря СЛОВА1 и слов, определенных после него, следует выполнить команду:

```
FORGET СЛОВО1
```

Слово FORGET проверяет содержимое переменной FENCE, в которой хранится некоторое граничное значение адреса. Слова, находящиеся за этим адресом, не могут быть уничтожены. Если же такая попытка предпринимается, то выдается сообщение:

```
СЛОВО1 PROTECTED
```

Для снятия защиты служит команда FENCE OFF. В переменной FENCE обычно находится указатель на последнее защищенное слово.

В некоторых Форт-системах имеется слово SAVE, которое позволяет сохранить текущую версию словаря, а потом загрузить его с диска. Оно может оказаться полезным при необходимости хранения словарей, содержащих уже скомпилированные слова для различных прикладных задач. Наконец, в некоторых системах имеется так называемый декомпилятор SEE, который восстанавливает определения слов по коду, находящемуся в их телах. Этот декомпилятор располагается за именем слова, определение которого требуется восстановить.

ВИРТУАЛЬНАЯ ФОРТ-МАШИНА

Ранее уже вскользь упоминалось, что слова Форта делятся на исполняемые и скомпилированные в виде шитого кода. Теперь мы более подробно объясним, что это означает. Поскольку Форт в полном объеме не был описан, то многие особенности вычислительной модели, лежащей в его основе, остались для вас скрыты. Например, вы знаете, что Форт имеет два стека. А какие еще есть у него компоненты? Хотя Форт может компилировать программу в чистый машинный код, так же как и трансляторы с других языков

программирования, он еще содержит в себе и встроенный компилятор, который активизируется словом : и завершает работу по символу ; в конце определения через двоеточие. Большинство Форт-компиляторов генерируют *шитый код*. Для иллюстрации этого обратимся к такому языку программирования, как Ассемблер. Опытные программисты, разрабатывающие свои программы на Ассемблере, обычно выделяют в программе несколько подпрограмм, которые выполняют некоторые элементарные функции и при необходимости могут составить базис других программ. В результате основная программа представляет собой совокупность подпрограмм и обращений к ним. Форт устроен аналогично, но роль элементарных подпрограмм выполняют слова из его ядра. При создании слов более высокого уровня (определяемых через двоеточие) вместо компиляции обращений к словам ядра Форт фактически компилирует их адреса. Таким образом, шитый код представляет собой список адресов подпрограмм, подлежащих выполнению.

В подобном виде шитый код, конечно, не может быть выполнен, так как в нем отсутствуют операторы перехода. Для исполнения кода применяется небольшая специальная программа, называющаяся *адресным интерпретатором*, или *внутренним интерпретатором* NEXT. Упомянутый интерпретатор весьма прост. На рис. 6.6 приведена схема его работы. Во-первых, хотя это и может показаться вам очевидным, единственным кодом, который способен воспринять компьютер, является машинный код, т.е. слова типа CODE и собственно внутренний интерпретатор. Операторы перехода, находящиеся во входном потоке команд, в конце концов обязательно должны достигать подпрограмм, представленных в машинном коде. В большинстве восьмиразрядных компьютеров применяется косвенный шитый код (рис. 6.6А). На компьютерах, оснащенных более эффективным микропроцессором 68000, может быть выполнен прямой шитый код (рис. 6.6В). Поскольку различия между ними незначительны, мы рассмотрим только один из них - косвенный шитый код.

Виртуальная Форт-машина имеет два регистра, используемые внутренним интерпретатором. Под них обычно резервируется две области памяти. В первом регистре находится указатель IP на команды шитого кода, выполняющий в программе функцию счетчика команд. Второй регистр W является рабочим, в нем хранятся временные данные. Внутренний интерпретатор NEXT обрабатывает шитый код следующим образом. Сначала он выбирает содержимое области памяти, на которую ссылается указатель IP. В ней находится cfa выполняемого слова. Затем cfa помещается в W и осуществляется косвенный переход на содержимое W. Записав в W cfa выполняемого слова, мы тем самым запомнили точку продолжения вычислений (она имеет имя FOO). Далее происходит переход от cfa к его содержимому. В cfa находится указатель на процедуру периода выполнения, которая будет интерпретировать данное

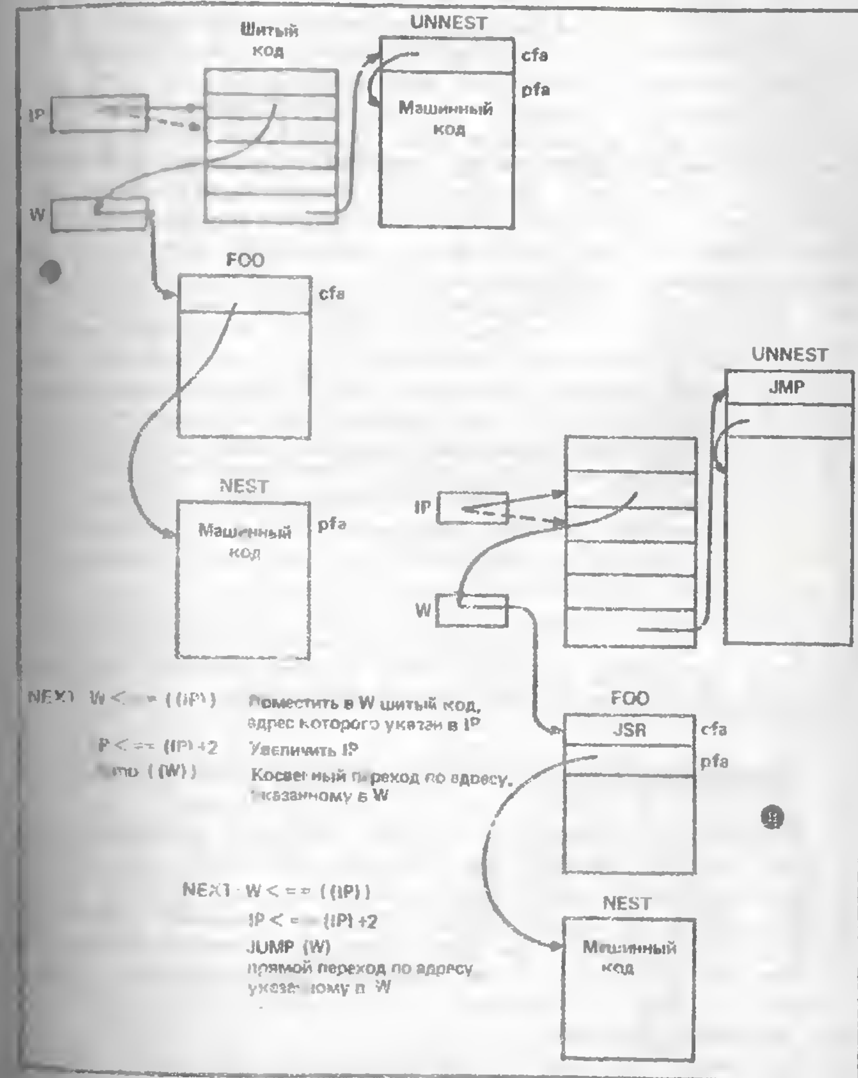


Рис. 6.6 Внутренний интерпретатор Форта: косвенного шитого кода (А) прямого шитого кода (В)

слово. Для слов, определенных с помощью слова ; , упомянутая процедура имеет имя NEST. Иногда она также называется DOCOLON или (:) - в последних версиях Форта.

Процессор сможет продолжить вычисления с данного слова, поскольку оно всегда должно находиться в машинном коде. Последнее условие является настолько важным, что заслуживает быть сформулированным в виде общего правила: *cfa слова должно содержать указатель только на машинный код.*

Процедура NEST поместит содержимое IP на стек возвратов, а содержимое W - в IP, затем увеличит значение IP таким образом, чтобы оно указывало на pfa слова FOO. Поскольку FOO определено через двоеточие, IP будет уже указывать на первый адрес шитого кода в FOO, после чего NEST перейдет на слово NEXT, а слово FOO начнет интерпретироваться. Последним адресом шитого кода, интерпретируемого словом NEXT, является слово UNNEST типа CODE. Оно помещает верхний элемент стека возвратов в IP и вызывает NEXT. Это приводит к возврату внутреннего интерпретатора на слово, которое вызвало только что завершившееся слово. В случае прямого шитого кода NEXT перейдет непосредственно на cfa слова, которое будет выполняться, поэтому по адресу поля кода (cfa) должен находиться переход на NEST, представленный обязательно в машинном коде. Отсюда следует, что прямой шитый код ближе к машинному, чем косвенный, но и работать с ним труднее, так как это фактически объектный код. Программы в их исходном виде для интерпретатора как косвенного, так и шитого кода в Форте практически одинаковы.

Помимо двух стеков и регистров внутреннего интерпретатора, виртуальная Форт-машина содержит еще область памяти с именем N. Такая временная рабочая память используется в основном словами, выполняющими арифметические операции.

ОПРЕДЕЛЯЮЩИЕ И КОМПИЛИРУЮЩИЕ СЛОВА

Форт-система может работать как в режиме интерпретации, так и в режиме компиляции. Функции интерпретатора обычно выполняет слово INTERPRET, а компилятора - слово]. Выбор имени] для компилятора может показаться несколько странным. Однако в Форте имеется еще одно слово, [, которое переводит его в режим интерпретации. Таким образом, пара квадратных скобок позволяет внутри определения через двоеточие перейти к интерпретации программы.

Состояние Форт-системы (интерпретация или компиляция) зависит от значения системной переменной STATE. Если она равна нулю, то Форт интерпретирует поступившую на его вход программу. Следовательно, слово [определяется так:

```
: [ STATE OFF ; IMMEDIATE
```


Определение тривиально, однако завершающее его слово IMMEDIATE является новым.

Если IMMEDIATE находится за каким-либо словом, то последнее является *компилирующим словом.* В приведенном выше описании первого байта из поля имени бит 6 последнего определенного слова устанавливается словом IMMEDIATE. При считывании очередного слова компилятор прежде всего проверяет значение этого бита. Если в нем находится единица, то компилятор сразу приступает к исполнению прочитанного слова, в противном случае компилирует его cfa. Так как слово выполняется в период компиляции, оно должно использоваться для поддержки самого процесса компиляции. Слово [- компилирующее, поскольку переводит Форт-систему в режим интерпретации. Такой переход может быть сделан только в период выполнения. Если же это слово начнет компилироваться как часть данного определения, то оно сможет "сработать" лишь при выполнении определяемого слова (в его период выполнения). Но нужный момент будет уже упущен, поскольку назначение слова [- переключать режимы в период компиляции. Таким образом, слово [является *компилирующим.* Чтобы продемонстрировать использование слов [и] в определении через двоеточие, необходимо объяснить назначение еще одного компилирующего слова - LITERAL. Когда компилятору встречается в тексте определения какое-либо число, он обращается к слову LITERAL. Оно компилирует слово периода выполнения с именем (LIT), за которым следует само число. В период выполнения подпрограмма (LIT) выбирает число из находящейся за ней области памяти, и переставляет указатель IP на область, расположенную за той ячейкой, в которой до этого находилось число.

Пусть нам необходимо скомпилировать внутри определения через двоеточие cfa слова FOO. Введем в определение выражение:

```
[ ' FOO ] LITERAL
```

Слово [переключит Форт-систему в режим интерпретации, а ' FOO возвратит cfa слова FOO. Затем] вернет систему обратно в режим компиляции и компилирующее слово LITERAL выполнит компиляцию cfa как числа. Поскольку такая последовательность операций встречается довольно часто, в Форте-83 предусмотрено специальное слово, результат действия которого эквивалентен упомянутой последовательности:

```
[ ' ] FOO
```

Заметим, что скобки в выражении [’] как раз и обеспечивают эту эквивалентность.

А что случится, если слово ' встретится внутри определения через двоеточие? Поскольку слово ' не относится к числу компилирующих (в отличие от [’]), оно скомпилируется в некоторое слово

и будет выполняться вместе с ним. При этом ' будет искать стоящее за ним слово во входном потоке, а его cfa - в словаре. Рассмотрим, например, слово

: FOO ' ;

Оно выполняет те же действия, что и слово ' , поэтому должно предшествовать имени некоторого другого слова. При выполнении выражения FOO СЛОВО1 вместе с ним выполняется и ' , которое считывает СЛОВО1 из входного потока (находящееся за FOO) и возвращает его cfa.

К классу компилирующих слов относятся также и управляющие конструкции. Как вы видели, слова IF, WHILE и UNTIL производят ветвление, если значение флага, снятого с вершины стека, ложно, в противном случае (т.е. когда значение флага отлично от нуля) они продолжают выполнять находящиеся за ними слова. При выполнении в период компиляции эти слова компилируют примитив перехода ?BRANCH и адрес перехода, находящийся за ним. Во время выполнения ?BRANCH снимает число (или флаг) с вершины стека. При его значении, равном нулю, производится переход. Если это не так, то указатель IP переставляется на адрес области шитого кода, находящийся за адресом перехода. Описанная схема приведена на рис. 6.7. При выполнении управляющих слов ELSE и REPEAT вместо слова ?BRANCH компилируется примитив BRANCH. Он реализует безусловный переход по адресу, находящемуся за ним в шитом коде.

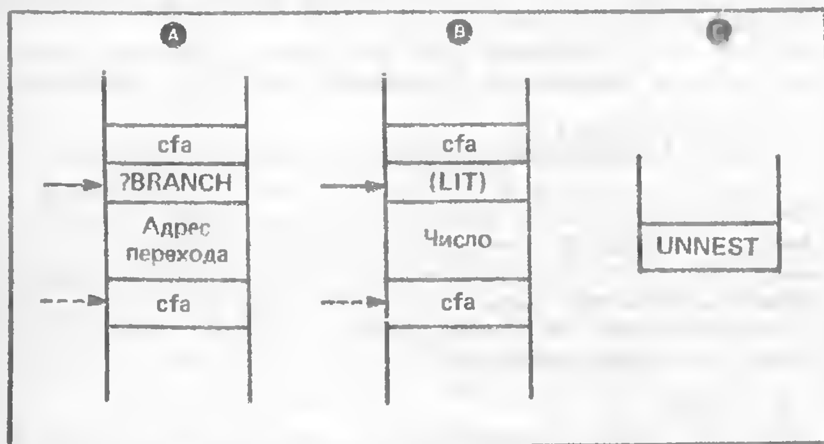


Рис. 6.7. Схема компиляции шитого кода для слов IF, WHILE (A), слова LITERAL (B), слова ; (C). Переменная IP переводится словом периода выполнения в положение сброшенного указателя

Компилирующие слова расширяют возможности компиляции. С помощью определяющих слов создаются новые слова. Все слова, введенные конкретным определяющим словом, используют одну и ту же подпрограмму периода выполнения. Например, все слова, определенные через двоеточие, в процессе исполнения вызывают подпрограмму NEST. Переменные, определенные словом VARIABLE, будут обращаться к подпрограмме (VAR), а константы - к (CON). Действия этих переменных в период выполнения объяснены выше, но мы пока еще не касались вопросов их компиляции.

При компиляции широко применяется слово CREATE. Оно создает в словаре поле имени, а также cfa, указывающий на подпрограмму (VAR). Если у создаваемого слова процедура периода исполнения иная, нежели для переменной, то cfa должен быть позднее изменен. Прежде всего слово CREATE строит поле связи для того, чтобы включить заголовок создаваемого им слова в некоторый список словаря. CREATE почти полностью завершает построение переменной, за исключением выделения памяти под ее значение, т.е. оно создает не тело, а только заголовок. Под переменную необходимо выделить еще одну ячейку памяти Форт-системы для хранения ее числового значения.

Выделение памяти осуществляется словом ALLOT, резервирующим область памяти заданного размера:

ALLOT (n ==>)	Выделение в словаре n байт.

ALLOT - простое слово, его определение выглядит так:

: ALLOT DP +! ;

Здесь DP - системная переменная Форта, которая содержит указатель на свободную область памяти, находящуюся непосредственно за последним байтом словаря. Конечно, этот адрес можно также получить с помощью команды DP@, но она применяется настолько часто, что ее целесообразно включить в число стандартных средств Форта в виде слова:

HERE (==> addr)	Возвращает адрес свободной области памяти, расположенной непосредственно за словарем.

Наконец, еще одно элементарное слово, применяемое при построении словаря, определяется следующим образом:

: , HERE ! 2 ALLOT ;

Данное определение справедливо для версий Форта, реализованных на 16-разрядных процессорах. Таким образом, чтобы завершить формирование определяющего слова для переменной, нам

необходимо только выделить два байта под ее значение. Для этой цели следует воспользоваться командой 2 ALLOT. Затем переменной необходимо присвоить начальное значение, равное нулю, что легче всего сделать с помощью слова "запятая", расположив его в упомянутом выражении после нуля:

```
: VARIABLE CREATE 0 , ;
```

Более широкие по сравнению с другими языками возможности Форта заключаются не в наличии средств для определения слов, не входящих в стандартный набор, а в том, что вы можете вводить новые определяющие слова и их исполняющие процедуры. Впоследствии эти определяющие слова применяются для задания новых слов, таких, как : (двоеточие) и VARIABLE.

Помимо периода компиляции и периода выполнения в Форте еще выделяется этап третьего типа - период компиляции определяющего слова. Оно называется *периодом определения*. Процедура DOES> позволяет вводить подпрограммы периода выполнения для слов, заданных с помощью новых определяющих слов. В качестве иллюстрации рассмотрим определение слова ARRAY, создающего массивы. Во время компиляции выполняются следующие действия. Слово

```
ARRAY (n ==> )
```

считывает число n, которое указывает размерность массива, задаваемую количеством выделенных ячеек памяти. Затем во время исполнения происходит обращение к слову

```
ARRAY ( n ==> addr )
```

При этом со стека снимается индекс созданного словом ARRAY массива и возвращает указатель на соответствующее значение. Определение слова ARRAY имеет вид:

```
: ARRAY CREATE 2 * ALLOT DOES> SWAP 2 * + ;
```

Пусть слово ARRAY встретилось в выражении:

```
10 ARRAY A1
```

Здесь слово ARRAY сначала создаст заголовок A1, затем извлечет из стека число 10, удвоит его и выделит 20 байт памяти под тело A1. На этом закончится период компиляции для A1.

Слово DOES> отмечает, что далее следует период выполнения для массивов. Вначале в cfa массива A1 возвращается значение, указывающее на начало последовательности действий. Затем компилируется машинный код перехода на слово (DOES) - исполняющую процедуру для DOES. Она создает шитый код, находящийся

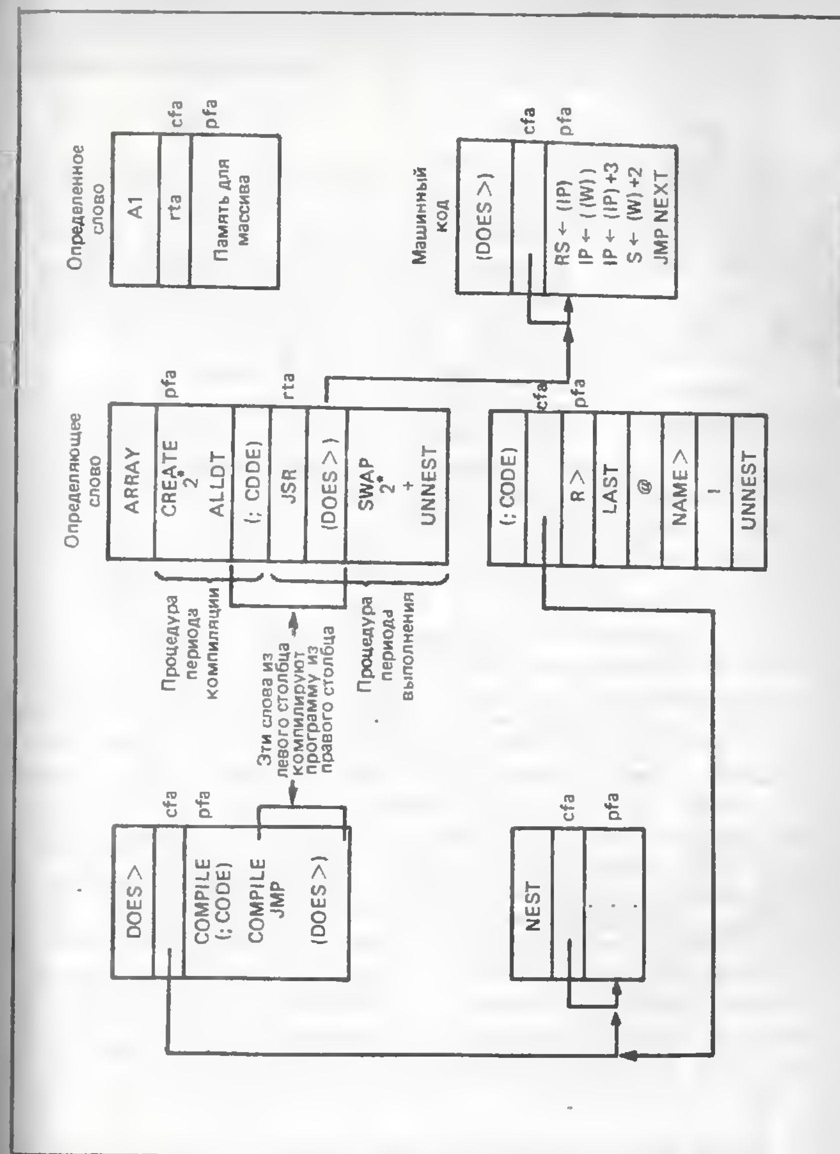


Рис. 6.8. Слово Форта DOES>, определенное через двоеточие, компилирует в слове ARRAY последовательность: (;CODE), JSR и (DOES>). Во время определения словом ARRAY слова A1 команда (;CODE) помещает rta в cfa слова A1. Команда (DOES>) инициирует выполнение внутренним интерпретатором процедуры периода выполнения слова ARRAY, которое находится за (DOES>)

за JSR (DOES), который и выполняется. Перечисленные действия слова DOES показаны на рис. 6.8. Их суть лучше всего пояснить в терминах трех типов периодов, выделяемых в процессе обработки слова.

В период определения слово ARRAY создается (компилируется). Во время компиляции производятся стандартные действия до тех пор, пока не встретится слово DOES>. Оно является компилирующим и будет выполнено немедленно. Слово DOES>, определенное через двоеточие, последовательно компилирует cfa выражения (; DOES) в ARRAY, затем машинную команду перехода JSR и, наконец, cfa слова (DOES), имеющего тип CODE. Следующие за DOES> слова - исполняющие подпрограммы ARRAY компилируются как элементы высокого уровня (шитый код).

Во время компиляции теперь, когда слово ARRAY уже определено, происходит собственно создание массива A1. Таким образом, для слова ARRAY начинается период выполнения, а для массива A1 - период компиляции. Подпрограмма периода выполнения слова ARRAY создает заголовок A1 и выделяет память для хранения массива. Затем выполняется слово (;CODE). Оно определено через двоеточие и поэтому первым действием снимает верхний элемент со стека возвратов. В этот момент в вершине стека возвратов находится указатель на шитый код слова ARRAY. Если указатель был снят во время выполнения подпрограммы UNNEST слова (; CODE), то внутренний интерпретатор вернется для продолжения работы на то слово, которое обратилось к слову ARRAY. Это один из возможных способов выйти из слова ARRAY без выполнения подпрограммы UNNEST. Значение, извлеченное из стека возвратов, служит указателем на область памяти, находящуюся непосредственно за словом (; CODE) в слове ARRAY, т.е. адресом подпрограммы периода выполнения для ARRAY. Остальные слова из (; CODE), следующие за словом R>, запоминают rta (адрес периода выполнения) в cfa A1, так что при обращении к A1 эта подпрограмма будет выполняться, начиная с rta. В период выполнения слова A1 подпрограмма NEXT перейдет с cfa A1 на rta, после чего инициализирует слово (DOES>). Оно аналогично слову NEST, поскольку также запоминает содержимое IP на стеке возвратов, пересылает значение, на которое указывает W (являющееся не чем иным, как rta) в IP, увеличивает его на три для обхода JSR (DOES>), помещает pfa A1 на стек параметров и переходит к слову NEXT.

Рассмотренная процедура - едва ли не самая сложная из всех, которые могут встретиться в Форте. Если столь подробное описание кажется вам не совсем ясным, можно ограничиться следующими сведениями, которых будет достаточно для практической работы. В период выполнения вновь определенных слов процедура DOES> заносит pfa слова, которое подлежит выполнению, на стек.

Таким образом, стековая нотация слова DOES> в период выполнения имеет следующий вид:

```
DOES> ( ==> pfa )
```

Слово ARRAY использует pfa A1 как базовый адрес. Оно помещает индекс массива на стек, предварительно умножив его на 2, чтобы выделить необходимую для хранения массива память, а затем прибавляет его к значению pfa. Результатом является указатель на значение массива с данным индексом.

В заключение приведем еще одно определяющее слово - DEFER. Оно нам уже встречалось ранее в качестве определителя векторов исполнения. Его определение выглядит так:

```
: DEFER CREATE [ ' ] NOOP , DOES> @ EXECUTE ;
```

DEFER сначала создает заголовок, а затем компилирует словом , cfa слова NOOP, которое не производит никаких действий:

```
: NOOP ;
```

После этого управление передается в исполняющую часть слова DEFER. Процедура DOES> возвращает pfa определенного с помощью DEFER слова, в котором было скомпилировано NOOP. Оно считывает cfa слова NOOP из заданной области памяти и выполняет NOOP подпрограммой EXECUTE. EXECUTE - слово Форта, которое снимает со стека cfa некоторого слова и выполняет его. Поскольку не имеет смысла исполнять одно лишь слово NOOP, pfa определяемых с помощью DEFER слов следует изменить таким образом, чтобы они указывали на нужное слово. Как уже отмечалось, слово EMIT обычно указывает на подпрограмму (EMIT), управляющую дисплеем. Для реализации этого нужно выполнить следующую последовательность слов:

```
' (EMIT) ' EMIT >BODY !
```

или использовать более подходящее нестандартное слово IS:

```
' (EMIT) IS EMIT
```

Действия слова IS зависят от текущего режима Форт-системы, при компиляции и интерпретации они различны. Слово COMPILE берет cfa следующей непосредственно за ними ячейки шитого кода и компилирует cfa в словарь. Находящееся в ячейке слово не относится к числу компилирующих, поскольку работает в период выполнения другого слова, находясь внутри него.

```
( cfa ==> )
: IS STATE @
IF COMPILE (IS)
ELSE ' >BODY !
THEN
; IMMEDIATE
```

Слово IS является компилирующим, так как обладает компилирующими возможностями. Его определение:

```
: (IS) R@ @ >BODY ! 2+ >R
```

СЛОВА ДЛЯ УПРАВЛЕНИЯ ВНЕШНЕЙ ПАМЯТЬЮ

Форт - больше чем язык, это программная среда, обладающая собственными средствами для взаимодействия с внешней памятью. На логическом уровне внешняя память представляется в виде совокупности блоков по 1024 байта (1K) в каждом, пронумерованных начиная с нуля. Внешняя память в Форте построена на основе принципа виртуализации. В памяти Форт-системы 1K блоков резервируется под буферный пул. При включении нового блока под него выделяется свободный буфер. Если все буферы оказались занятыми, то самый "старый" блок (тот, который обновлялся позже остальных) будет переписан обратно во внешнюю память, чтобы освободить место для нового блока. Каждый буфер включает в себя флаг (признак), по значению которого можно судить, обновлялся блок или нет. Если во время нахождения блока в буфере он не изменялся, то его не нужно переписывать обратно во внешнюю память.

Все буферы могут быть очищены с уничтожением содержащихся в них данных с помощью слова EMPTY-BUFFERS. Для сохранения во внешней памяти всех блоков, подвергавшихся изменениям, служит слово SAVE-BUFFERS. Перед очисткой внешней памяти целесообразно выполнить слово FLUSH, определяемое следующим образом:

```
: FLUSH SAVE-BUFFERS EMPTY-BUFFERS ;
```

Важную роль играют слова LOAD и THRU, управляющие входным потоком данных, передаваемых из внешней памяти. Слово LOAD загружает один экран, а слово THRU - последовательность экранов:

LOAD (n ==>)	Переписывает блок n.
THRU (b e ==>)	Переписывает блоки с b по e.

Для работы с исходными текстами программ, находящимися во внешней памяти, требуется текстовый редактор. В Форте не предусмотрен какой-либо стандарт на средства редактирования, поэтому за их описанием следует обращаться к документации по конкретой системе. Блоки в редакторе называются *экранами*. В конце книги помещен глоссарий слов Форта. Там же приведена библиография по языку Форт.

УПРАЖНЕНИЯ

1. Определите следующие слова Форта, используя стандартные слова:
   а) ON и OFF б) 2DUP в) -ROT
   г) +! д) ?DUP ж) COUNT
2. Каково назначение слов:
   а) (IS)
   б) ?CREATE

(оно приведено на экране 45 исходных текстов программ в приложении А)?

Глава 7

ОБРАБОТКА СПИСКОВ

Формы представления данных, или *структуры данных*, в компьютере весьма разнообразны. В предыдущей главе при программировании на Форте использовались следующие структуры данных: связный список для хранения слов в словаре, стеки для передачи параметров и хранения указателей возврата, а также строки, например, символьные строки в полях имени слов. Из всех этих структур данных списковая наиболее универсальна. С помощью списков могут строиться другие структуры данных, такие, как деревья, строки или стеки. Подобная универсальность полезна при представлении сложных форм знаний в программах ИИ, многие из которых построены на базе списков. Обработка списков характерна для символьных вычислений вообще, а для ИИ в особенности. Списки являются одним из инструментов создания систем, основанных на знаниях.

Слова Форта для работы со списками, рассматриваемые здесь, заимствованы из языка Лисп (LISP - List processor). Лисп был разработан Дж. Маккарти в 50-х годах для исследований в области искусственного интеллекта. Хотя по "компьютерным меркам" этот язык считается довольно старым, он не приобрел той популярности, какую завоевали впоследствии Фортран или Бейсик, причем не в силу своей концептуальной сложности, а из-за того, что он в большей степени подходил для символьных преобразований, чем для численных. Большинство же компьютерных прикладных программ продолжало в основном оставаться расчетными.

Численные методы используются главным образом для интегрирования функций или решения уравнений. На Лиспе написана программа MACSYMA, которая не только решает подобные задачи, но и воспринимает информацию в привычных математических обозначениях. Анализируя формы алгебраических выражений, она устанавливает правила интегрирования, применимые к этим выражениям. Символьные вычисления нашли коммерческое применение

сравнительно недавно. Лисп-машины в настоящее время продаются несколькими компаниями и снабжены программами типа MACSYMA. В гл. 9 рассматривается другое направление, вынесенное в название данной книги. Разработанная в рамках этого направления система, основанная на знаниях, реализована посредством слов управления списками.

ДЛЯ ЧЕГО НУЖНО ЭМУЛИРОВАТЬ ЛИСП?

Лисп представляет собой простой элегантный язык обработки списков, и поэтому естественно, что выбор пал на него. Однако Форт по сравнению с Лиспом обладает определенными преимуществами, и нашу разработку следует рассматривать как попытку объединить лучшие свойства этих языков путем расширения Форта лиспоподобными средствами. Предлагаемые слова обработки списков *не являются* транслятором с Лиспа, написанным на Форте: интерпретатор Форта не был заменен циклом EVAL-APPLY (так называется интерпретатор Лиспа). Тем не менее Форт и Лисп во многом схожи:

* Оба языка расширяемы в том смысле, что можно определять новые функции. Однако в Лиспе нет понятия словаря; он не позволяет, в отличие от Форта, использовать имя одной и той же функции в различных контекстах. В Лиспе также отсутствует возможность расширения средств компиляции.

* Как Форт, так и Лисп ориентированы на вызовы функций. При передаче параметров в Форте применяется постфиксная (или обратная польская) запись, в то время как в Лиспе - префиксная (имя функции должно предшествовать аргументам). Чтобы выдержать единообразие, мы приняли порядок следования аргументов, соответствующий Лиспу. Для передачи параметров между словами-функциями в Форте имеется стек. Работа со стеком в Форте осуществляется явно. Лисп также имеет аналогичный стек, но он скрыт от пользователя.

* Идентификаторами в Форте служат переменные, расположенные в словаре - статически распределенном списке. OBLIST в Лиспе несет ту же функциональную нагрузку.

* И в Форте, и в Лиспе широко используется рекурсия - прием программирования (см. гл.8), в котором разрешается производить вызов функции из тела этой же функции с запоминанием состояния исходного экземпляра функции. Стековый механизм такую возможность допускает, так как прежние параметры при новом обращении к той же функции не затираются новыми параметрами, а проталкиваются в глубь стека.

* И Форт, и Лисп имеют простой синтаксис, что частично вытекает из реализации вычислений вызовами функций.

• Оба языка компактны. Транслятор с чистого Лиспа занимает приблизительно такой же объем памяти, что и Форт-система без расширений.

• И Форт, и Лисп являются не просто языками, а *системами программирования*. Каждый из них наделен функциями операционной системы и может использоваться автономно как полноправная вычислительная среда. Они снабжены механизмами расширения динамического окружения и возможностей прикладного уровня. Некоторые расширения Лиспа послужили основанием для того, чтобы ошибочно считать его громоздким языком, занимающим память объемом в мегабайты.

• Эти языки обеспечивают удобную интерактивную среду интерпретирующего типа, что хорошо согласуется с восходящим стилем программирования, принятым в области ИИ, упрощает экспериментирование с программами и их "расчленение". Поскольку восходящий стиль программирования не рекомендуется для больших проектов, в которых занято много людей и требуется строгая дисциплина программирования, программы, выдержанные в стиле ИИ, сравнительно малы. И в Форте, и в Лиспе очень многое можно сделать с помощью программы небольшого размера, но эта программа должна быть тщательно продумана. Стиль ИИ подходит для тех случаев, когда вырабатываются новые понятия, так как проработка общей структуры (сверху вниз) невозможна без достаточных знаний о программах нижнего уровня, или когда область применения программы уточняется в ходе ее создания и эксплуатации. Поскольку лиспоподобное расширение Форта, рассматриваемое в книге, очень похоже на сам Лисп, при программировании рекомендуется пользоваться пособиями по Лиспу. Ссылки на соответствующую литературу приводятся в приложении.

СТАТИЧЕСКОЕ И ДИНАМИЧЕСКОЕ УПРАВЛЕНИЕ ПАМЯТЬЮ

Слова Форта состоят из последовательности адресов шитого кода, ссылающихся на другие слова. Будучи однажды скомпилированными, эти слова имеют постоянное местоположение. Следовательно, при определении таких структур данных, как константы, переменные и массивы, за ними закрепляется фиксированная область памяти. Для многих прикладных программ подобное *статическое* распределение памяти приемлемо. Однако существует ряд приложений, где требуется *динамическое* распределение памяти, т.е. распределение, при котором память, выделенная под заданную структуру данных, не фиксируется во время компиляции и ее местоположение может меняться во время выполнения программы.

Если мы рассмотрим возможности распределения памяти под такие структуры Форта, как массивы, то сразу же столкнемся со сложными схемами управления памятью. Элементы массива занимают фиксированное место в области памяти и доступ к ним осуществляется по значению индекса. Переопределить порядок элементов массива или сделать число элементов в массиве неограниченным довольно трудно. Чтобы освободить память для добавляемых к массиву элементов, требуется сдвинуть все, что расположено в памяти за ним, и обновить указатели на перемещенные объекты. Очевидно, что это не выполнимо. Списки делают динамическое управление памятью концептуально проще, а с точки зрения вычислений - и эффективней.

ЧТО ТАКОЕ СПИСОК?

В Лиспе список строится из множества соединенных между собой так называемых *ячеек связи* (CONS-cells), или *точечных пар*. Эти ячейки представляют собой элементарные структуры, из которых составляются списки. Ячейка связи представляет собой тип данных, состоящий из двух основных компонент. Возможное графическое представление такой ячейки показано на рис. 7.1.

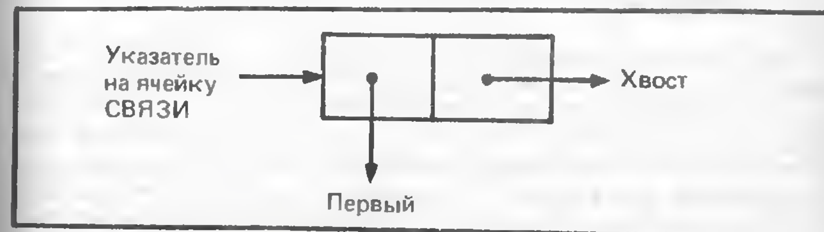


Рис. 7.1. Ячейка связи, элементарная структура данных, из которых строятся списки

Левый компонент - это *голова*, или *первый* (CAR), а правый - *хвост*, или *остаток* (CDR). Ничего не значащие имена CAR и CDR сложились исторически. В свое время так обозначались машинные регистры, используемые в реализациях Лиспа. Как голова, так и хвост служат указателями. Голова указывает на элемент списка, а хвост - на следующую ячейку связи. Пример построения списка из трех элементов показан на рис. 7.2.

Указатель на список - это всего лишь адрес. Список может быть представлен в стеке данных Форта в виде указателя на первую ячейку связи (или голову) списка. (Список в стеке представлен проще, чем строка Форта, поскольку она требует наличия в стеке двух элементов: адреса начала строки и ее длины.) Список, показанный на рис. 7.2, состоит из трех элементов, на которые ссылаются соответствующие головы ячеек связи, - списки А, В и С (по порядку обхода списка). Хвост последней ячейки связи указы-

вает на НУЛЬ, что означает конец списка. У нас НУЛЬ является переменной Форта.

Для списков существует и другая (даже более подходящая) так называемая *скобочная запись*, согласно которой список просто заключается в круглые скобки. В скобочной записи список, показанный на рис. 7.2, будет выглядеть следующим образом:

(A B C)

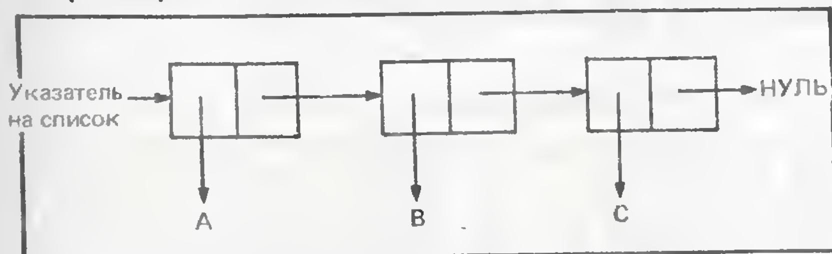


Рис. 7.2. Список из трех элементов: (A, B, C)

Реализация лиспоподобных слов на Форте, обсуждаемая в настоящей главе, показана на экранах с 40 по 49, а также в приложении А, где приводятся листинги исходных текстов. В такой реализации списки создаются несколько иным способом, отличным от рассмотренного выше. Здесь применяется более общая структура использования ячеек связи - с динамическим управлением списками. Во избежание необходимости динамического управления памятью списки создаются в виде переменных Форта, для которых память выделяется статически. Список идентифицируется именем переменной, а собственно список находится в теле этой переменной. Ее pfa содержит указатель на выделенный участок памяти под голову списка.

Слово НОВСПИСОК (NEWLIST) является определяющим словом, создающим переменную с именем, которое следует за НОВСПИСОК во входном потоке. Кроме того, оно берет из стека число, задающее максимальное количество элементов для данного списка. Это число используется для выделения памяти следом за переменной. Полученное в результате выполнения слово действует как переменная периода выполнения, поскольку CREATE в НОВСПИСОК построит cfa, указывающий на процедуру периода выполнения, соответствующую переменной. НОВСПИСОК инициирует вновь созданный список как пустой. Переменная Форта с выделенной дополнительной памятью называется *расширенной переменной*. Под списками мы подразумеваем расширенные переменные.

ПРОСТЕЙШИЕ ОПЕРАЦИИ НАД СПИСКАМИ

Чтобы получить указатель на первый элемент списка (голову списка), активируется слово ПЕРВЫЙ (FIRST). Оно помещает в стек голову первой ячейки связи списка, при этом указателем на список служит @СПИСОК (он же и будет находиться в стеке). Например, головой списка, изображенного на рис. 7.2, является А. Для занесения в стек хвоста первой ячейки связи списка активируют слово ХВОСТ (TAIL). (В Лиспе вместо слов ПЕРВЫЙ и

```
   40
 0 \ СЛОВА ПОСТРОЕНИЯ СПИСКОВ ЛИСПА НА ФОРТЕ-83
 1
 2 VARIABLE НУЛЬ НУЛЬ НУЛЬ ! \ ПУСТОЙ СПИСОК
 3
 4 ( #ЭЛЕМЕНТОВ -> ) \ #ЭЛЕМЕНТОВ = МАКСИМ. ЧИСЛУ
 5                 \ ЭЛЕМЕНТОВ СПИСКА
 6 : НОВСПИСОК CREATE HERE 2+ , НУЛЬ , 2* ALLOT ;
 7 ( @СПИСОК -> @ПЕРВЫЙ) \ @ПЕРВЫЙ-УКАЗАТ. НА ПЕРВЫЙ
 8                  \ ЭЛЕМ. СПИСКА
 9 : ПЕРВЫЙ @ ;
10 ( @СПИСОК : НУЛЬ -> ФЛАГ) \ ФЛАГ = ИСТИНА, ЕСЛИ
11                    \ СПИСОК ПУСТОЙ
12 : НОЛЬ @ НУЛЬ = ;
13 ( @СПИСОК -> @ХВОСТ) \ @ХВОСТ - УКАЗАТЕЛЬ НА ОСТАТОК
14 СПИСКА
15 : ХВОСТ DUP НОЛЬ IF @ ELSE 2- THEN ;
```

Экран 40. Слова построения списков: НОВСПИСОК, ПЕРВЫЙ и ХВОСТ

ХВОСТ все еще по традиции используются слова CAR и CDR, соответственно.) Слово ХВОСТ заносит в стек указатель на оставшуюся часть списка:

(B C)

Если мы снова активируем слово ХВОСТ, то получим в результате (C), а при повторной активации - НУЛЬ. Таким образом, применив к исходному списку выражение

ХВОСТ ХВОСТ ХВОСТ

мы получим в вершине стека НУЛЬ. Слова ПЕРВЫЙ и ХВОСТ позволяют нам перемещаться по спискам. Всякий раз, применяя

слово ХВОСТ, мы укорачиваем текущий список на один элемент, а с помощью слова ПЕРВЫЙ получаем указатель на первый элемент текущего списка. Например, если нужно установить указатель на второй элемент

ХВОСТ ПЕРВЫЙ

то будет получен указатель на В.

В Лиспе некоторые комбинации операций CAR и CDR объединены в одну операцию:

Операция	*Последовательность1*	*Последовательность2*
CAAR	CAR CAR	ПЕРВЫЙ ПЕРВЫЙ
CDDR	CDR CDR	ХВОСТ ХВОСТ
CADR	CDR CAR	ХВОСТ ПЕРВЫЙ
CDAR	CAR CDR	ПЕРВЫЙ ХВОСТ

CADR помещает на вершину стека второй элемент списка, в то время как CDAR - указатель на второй элемент головы исходного списка. Например, CADR применительно к ((A B)C D) даст C, а если применить к тому же выражению CDAR, то получим (B).

Конструирование списков осуществляется с помощью слова СВЯЗЬ (CONS). В Лиспе для выполнения этого слова требуются два аргумента: @ЭЛЕМЕНТ и I - идентификатор списка. СВЯЗЬ получает свободную ячейку и устанавливает ее голову на @ЭЛЕМЕНТ, а хвост - на I. В данной реализации ячейки связи не используются. Они заменены ячейками списка, головы которых заносятся в некоторый массив в обратном порядке, как показано на рис. 7.3. Адрес pfa переменной списка указывает на голову списка, которая расположена по самому старшему адресу. Указатели на элементы списка (головы его ячеек) располагаются в порядке уменьшения адресов до последней ячейки (она же начальная), если считать от pfa. Завершает список указатель на НУЛЬ. На рис. 7.3 приведен список (A B C).

Поскольку не используется аппарат ячеек связи, слово СВЯЗЬ должно всегда воспринимать список в памяти как переменную. В отличие от Лиспа здесь нельзя получать свободные ячейки (которые могут быть разбросаны в памяти произвольно). В Лиспе ячейка связи помещает на стек указатель на голову результирующего списка. А слово СВЯЗЬ, определенное на экране 41, указатель в стек не заносит, так как оперирует непосредственно областью размещения списка в памяти. Следовательно, имя списка I в стеке является переменной-списком в словаре Форта.

Под такие списки (определяемые через НОВСПИСОК) выделяется фиксированный объем памяти, поэтому их максимальная длина, т.е. максимальное число членов списка, заранее определена.

```
   41
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2  ( I -> ) \ СПИСОК ДЕЛАЕТСЯ НУЛЕВЫМ (ПУСТОЙ СПИСОК)
 3 : ПУСТОЙ DUP 2+ DUP ROT ! НУЛЬ SWAP ! ;
 4
 5 ( I @СПИСОК -> ) \ УСТАНОВКА ИМЕНИ СПИСКА I НА @СПИСОК
 6 : УСТАНОВИТЬ DUP НУЛЬ = IF DROP ПУСТОЙ ELSE SWAP ! THEN ;
 7
 8 ( @ЭЛЕМЕНТ I -> ) \ ДОБАВЛ. @ЭЛЕМЕНТ К ГОЛОВЕ СПИСКА I
 9 : СВЯЗЬ 2 OVER +! @ ! ;
10
11 \ СЛОВО, ОСУЩЕСТВЛЯЮЩЕЕ РЕКУРСИЮ
12 : РЕКУРСИЯ LAST @ NAME> , ; IMMEDIATE
13
14
15
```

Экран 41. Слова построения списков: УСТАНОВИТЬ, СВЯЗЬ и РЕКУРСИЯ

СВЯЗЬ присоединяет новый элемент к списку, продвигая указатель списка (в pfa) вперед на один элемент и помещая на выделенное место указатель на новый объект - @ЭЛЕМЕНТ. Для удаления элемента из головы списка нужно всего лишь переместить указатель списка (pfa) в противоположном направлении. Если вы хотите присоединить список (A B C) к списку с именем I1, чтобы тот предварял список I1, достаточно ввести следующее:

C I1 СВЯЗЬ B I1 СВЯЗЬ A I1 СВЯЗЬ

ИДЕНТИФИКАТОР И УКАЗАТЕЛЬ СПИСКА

Списки идентифицируются своими именами в словаре Форта. При активации имен они выполняются как переменные, помещая в вершину стека указатель на pfa слова-списка, т.е. *идентификатор списка*, или *имя списка*. Указатель, помещенный в pfa имени списка, указывает на голову списка, поименованного данным именем. Такой указатель называется *указателем списка*. Так как расположение pfa слова-списка относительно составляющих его полей, известно, оно может быть использовано для представления

6 Заказ № 299

списка в качестве слова Форта. С другой стороны, при заданном указателе списка невозможно узнать, где находится pfa соответствующего имени списка. Для работы же со списком требуется указатель списка, который должен быть передан словам ПЕРВЫЙ и ХВОСТ. Этот указатель, помещенный в стек, и будет называться списком.

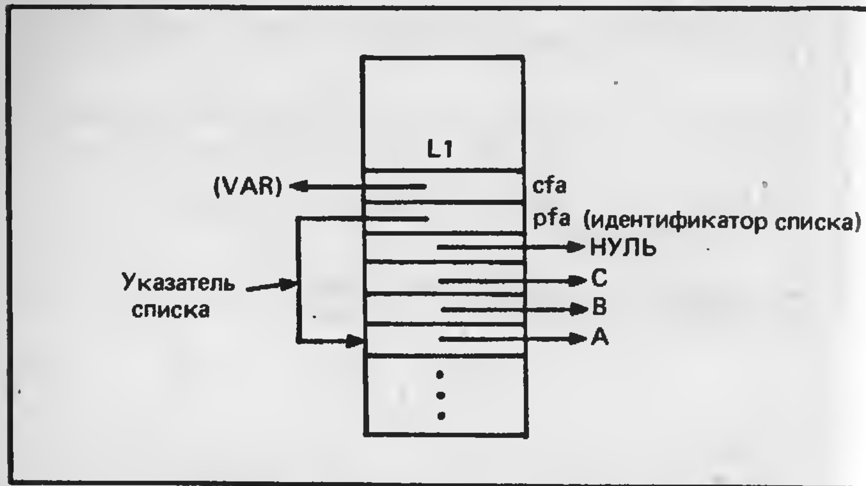


Рис. 7.3. Словарная структура списка, созданного словом НОВСПИСОК. Указатель на голову списка помещен в pfa, а по адресу pfa+2 находится указатель на НУЛЬ, отмечающий конец списка (A B C)

При заданном имени списка указатель списка может (а часто и должен) быть получен с помощью операции выборки @. В частности, выделить указатель списка из имени списка l1 можно так:

```
l1 @
```

Для того чтобы имя списка указывало на список, нужно воспользоваться словом УСТАНОВИТЬ (SET), имеющим следующую стековую нотацию:

```
( l1 @СПИСОК -> )
```

где l1 - имя списка (его pfa), а @СПИСОК - указатель списка. Если, например, ввести команды

```
l1 НУЛЬ УСТАНОВИТЬ
```

то список с именем l1 станет пустым (или нулевым).


```
     42
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2 ( НУЛЬ S1 S2 . . . SN l -> ) \ ПОСТРОЕНИЕ СПИСКА С ИМЕНЕМ l
 3 : СПИСОК >R
 4   BEGIN DUP НОЛЬ NOT
 5   WHILE R@ СВЯЗЬ
 6   REPEATE R> 2DROP
 7 ;
 8
 9  ( @СПИСОК l -> ) \ РЕКУРСИВНОЕ ОПРЕДЕЛЕНИЕ 2СОЕД
10 : 2СОЕД OVER НОЛЬ
11    IF 2DROP
12    ELSE OVER ХВОСТ OVER РЕКУРСИЯ
13       SWAP ПЕРВЫЙ SWAP СВЯЗЬ
14    THEN
15 ;
     43
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2 ( @ -> ФЛАГ) \ ФЛАГ = ИСТИНА,
 3              \ ЕСЛИ @ ДАЕТ PFA ПЕРЕМЕННОЙ
 4 : АТОМ? BODY> @ ['] НУЛЬ @ = ;
 5 ( @СПИСОК -> )
 6 : ВЫДАТЬСП CR ." ("
 7   BEGIN DUP ПЕРВЫЙ DUP АТОМ?
 8     IF DUP НОЛЬ NOT
 9        IF BODY> >NAME .ID ELSE DROP THEN
10     ELSE РЕКУРСИЯ
11    THEN ХВОСТ DUP НОЛЬ
12   UNTIL 8 ( ЗАБОЙ) EMIT ." ) " DROP
13 ;
14
15
     44
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2 ( @СПИСОК -> )
 3 : ВЫДАТЬ DUP @ НОЛЬ
 4    IF DROP CR ." НУЛЬ"
 5    ELSE DUP АТОМ?
 6       IF BODY> >NAME .ID
 7       ELSE ВЫДАТЬСП
 8       THEN
 9    THEN
10 ;
11
12
13
14
15
```

Экраны 42, 43, 44. Слова построения списков: СПИСОК, 2СОЕД, АТОМ?, ВЫДАТЬСП и ВЫДАТЬ

ВЫВОД СПИСКОВ НА ПЕЧАТЬ

Вывод списков на печать производится словом ВЫДАТЬ (PRINT). Оно берет из стека указатель на список и выводит список в скобочной записи. Для удобства восприятия длинных списков каждая левая скобка начинается с новой строки. Чтобы вывести список с именем I1, нужно ввести:

```
I1 @ ВЫДАТЬ
```

Операция @ помещает на стек указатель списка с именем I1.

ВВОД СПИСКОВ

Слово СВЯЗЬ представляет собой примитивную операцию построения списков, однако такая операция не совсем удобна с точки зрения пользователя. Слово СПИСОК (LIST) в этом отношении несколько лучше. Оно берет элементы из стека и включает их в список. Нуль в стеке отмечает начало списка. Стековая нотация слова СПИСОК:

```
( НУЛЬ S1 S2 . . . SN I1 -> )
```

где S1 . . . SN - элементы, а I1 -имя списка. Результирующий список с именем I1 следующий:

```
(S1 S2 . . . SN)
```

Введя

```
I1 @ ВЫДАТЬ
```

вы увидите этот список.

Элементы списка служат указателями на pfa переменных Форта. При активации слова СПИСОК слова с S1 по SN уже должны существовать. Но самый простой путь ввода списка - с помощью ЧТСП (READL). Это слово представляет собой небольшой символьный интерпретатор, создающий списки из следующих объектов: (,), @, атомов и имени списка. ЧТСП обращается к ЧТСИМ (READCH), которое выдает очередной символ из входного потока. Оно воспринимает список только в скобочной записи. Чтобы ввести список, нужно набрать:

```
I ЧТСП
```

где I - имя списка, а затем набрать список в скобочной записи. Левая круглая скобка (начинает список, а правая) заканчивает его.

Непосредственно можно ввести список только одного уровня. Такой, например, список ввести сразу нельзя:

```
(A (B C) D)
```

поскольку в нем содержится внутренний список. В данной реализации каждый список должен быть определен через НОВСПИСОК.

```
 45
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2 ( -> C) \ ЧТЕНИЕ ОЧЕРЕДНОГО СИМВОЛА ИЗ ВХОДНОГО ПОТОКА
 3 : ЧТСИМ
 4   BEGIN ИСТОЧНИК >IN @ /STRING
 5     IF C@ 1 >IN +! TRUE
 6     ELSE DROP ['] ИСТОЧНИК >BODY @ ['] (ИСТОЧНИК) =
 7       IF QUERY ELSE 1 BLK +! 0 >IN ! THEN FALSE
 8     THEN
 9   UNTIL
10 ;
11 ( -> CFA) \ ЕСЛИ СЛОВА НЕТ В СЛОВАРЕ,
12            СОЗДАЕТСЯ ПЕРЕМЕННАЯ
13 : ?CREATE >IN @ DEFINED
14 IF NIP ELSE DROP >IN ! HERE VARIABLE 4 + NAME> THEN
15 ;
 46
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2 ( I -> )
 3 : ЧТСП >R
 4   BEGIN ЧТСИМ
 5     CASE BL
 6       OF FALSE ENDOF ASCII (
 7       OF НУЛЬ FALSE ENDOF ASCII )
 8       OF R@ СПИСОК TRUE ENDOF ASCII @
 9       OF @ FALSE ENDOF
10      -1 >IN +! ?CREATE EXECUTE FALSE ROT
11     ENDCASE
12   UNTIL R> DROP
13 ;
14
15
```

Экраны 45, 46. Слова построения списков: ЧТСИМ, ?CREATE и ЧТСП.

Если определить (B C) как I1, то приведенный выше список можно ввести как I2:

I2 ЧТСП (A I1 @ D)

Обратите внимание на символ @, следующий за I1, и знак пробела после каждого имени элемента, включая последний, за D. Для сканирования слов применяется сканер Форта, который в качестве ограничителя требует символа пробела. Знак @ используется для получения указателя списка I1, а не его идентификатора. При выводе I2 результат будет таким же, как показано выше, но если мы вместо прежнего запишем

I2 ЧТСП (A I1 D)

а затем

I2 @ ВЫДАТЬ,

то получим

(A I1 D)

```
    50
 0 \ ОТДЕЛЬНЫЕ СПИСКИ
 1
 2 20 НОВСПИСОК I1
 3 20 НОВСПИСОК I2
 4 20 НОВСПИСОК I3
 5
 6 VARIABLE S1
 7 VARIABLE S2
 8 VARIABLE S3
 9
10 I1 ЧТСП (S1 S2 S3 )
11 I2 ЧТСП (S1 I1 @ S2 S3 )
12 I3 ЧТСП (S3 )
13
14
15
```

Экран 50. Отдельные списки

ТИПЫ ДАННЫХ ПРИ РАБОТЕ СО СПИСКАМИ

В приводимых выше примерах головы списков ссылались на другие списки или элементы, в частности на A, B, C или НУЛЬ. Такие элементы называются *атомами*. В лиспоподобном расширении Форта атом представлен в стеке как pfa некоторой переменной. Идентификатор списка - тоже вид атома. Атомами считаются либо имя списка, либо переменные, на которые указывают головы списков. В Лиспе атомами являются типы данных, во многом схожие с константами и переменными Форта. Это объекты, на которые ссылаются списки, но не сами списки.

Слово Форта АТОМ? на экране 41 идентифицирует указатели как атомы, если они служат адресами pfa переменных. Поскольку идентификаторы списков представляют собой расширенные переменные Форта, то и они - атомы. Указатели же списков атомами назвать нельзя: они отображают списки в большей степени, чем атомы, потому что указывают на фактическую списковую структуру. Так называемое С-ВЫРАЖЕНИЕ (символьное выражение) - либо атом, либо список и в стековой нотации обозначается символом С. Иерархия списковых типов данных представлена на рис.7.4.

Рис. 7.4. Иерархия типов данных Лиспа, используемых в книге. Идентификатор списка представляет собой pfa переменной Форта, именующей список

При использовании переменной Форта в стеке оказывается указатель на ее значение, а не само значение. Этот указатель в стековой нотации обозначен символом I в том случае, если переменная является идентификаторм списка. Указатели на списки обозначаются сокращенно СП или @СПИСОК, а на них самих

ссылается идентификатор списка. Чтобы получить указатель на список, требуется применить операцию @.

ЧТО ТАКОЕ НУЛЬ?

Нуль - единственный в своем роде атом, который в то же время представляет собой и список (пустой список). НУЛЬ в стековой нотации аналогичен (). Кроме того, НУЛЬ ссылается сам на себя. Выражение НУЛЬ @ помещает на стек НУЛЬ. По определению заголовок НУЛЯ и хвост НУЛЯ - НУЛИ.

Список свойств содержит в последней ячейке хвост НУЛЬ, что используется многими словами обработки списков, поскольку НУЛЬ обрывает просмотр, осуществляемый этими словами. Могут быть созданы списки, не заканчивающиеся НУЛЕМ (например, замкнутые списки), но их применение может привести к процедурам без останова. Слово НОВСПИСОК помещает НУЛЬ во вновь созданные (пустые) списки.

СПИСКИ СВОЙСТВ

Атомы могут ссылаться на уже упоминавшиеся выше *списки свойств*. Список свойств отличается от обычного списка наличием в голове символа -1 (или другого специального символа). Он создается таким образом, что имена свойств и их значения в нем чередуются. Для списка СП заголовок СП равен -1, второй элемент отображает первое свойство, а третий - его значение (рис. 7.5).

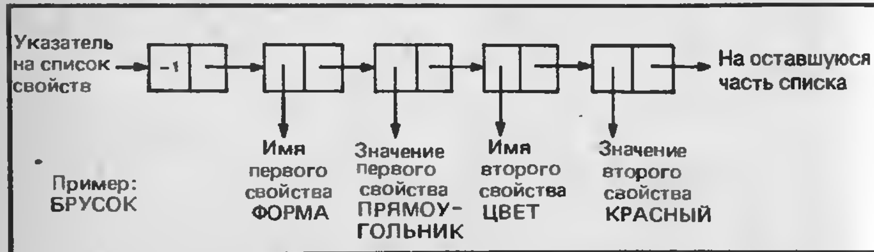


Рис. 7.5. Структура списка свойств с двумя свойствами БРУСКА

Слова, оперирующие со списками свойств, представляют собой основные функции доступа к базе данных. Одно из этих слов ПОЛУЧИТЬ (GET) описывается следующей стековой нотацией:

(I СВ -> ЗН)

где ЗН - значение свойства СВ (С-выражение), СВ -имя свойства (атом). Для того чтобы получить на стеке значение свойства ЦВЕТ атома БРУСОК, нужно ввести

БРУСОК ЦВЕТ ПОЛУЧИТЬ

В стек при этом помещается указатель на значение свойства ЦВЕТ. Затем, если мы введем слово ВЫДАТЬ, то получим значение КРАСНЫЙ. Слово ПОМЕСТИТЬ (PUT), или его синоним ПОМЕСТСВ (PUTPROP), помещает в список свойство и его значение. Если ввести

I ЗН СВ ПОМЕСТИТЬ

то свойство СВ со значением ЗН помещается в список как атом I. В том случае, когда свойство СВ со значением ЗН уже есть в данном списке, прежнее значение будет замещено ЗН. Слово ПОМЕСТИТЬ оставляет на стеке ЗН. Чтобы удалить некоторое свойство, достаточно ввести

I СВ УДАЛСВ

(REMPROP). При этом, помимо прочего, в стек помещается еще и флаг *истина*, если заданное свойство удалено, или *ложь* - если такого свойства в списке не оказалось. Наконец, имеется слово ПОЛУЧСП (GETL), формат использования которого следующий:

I СП ПОЛУЧСП

Это слово просматривает список свойств I в поисках первого элемента СП, представляющего собой свойство из I. Оно доставляет оставшуюся часть списка свойств, включая имя свойства, а в противном случае возвращает НУЛЬ.

Списки свойств и функции доступа к ним могут применяться для создания баз данных. Номер телефона директора предприятия, фамилии сотрудников - все эти сведения могут быть оформлены в виде атомов со списками свойств, содержащих свойства: адрес и номер телефона. Их значения могут ссылаться на атомы, которые являются строками (для адресов) или константами (для номеров телефона). Сохранение всего словаря Форта имело бы следствием сохранение и базы данных. Операции ПОМЕСТИТЬ И УДАЛСВ наиболее эффективно реализуются посредством функций преобразования списков (они будут рассмотрены ниже в настоящей главе). Исходные тексты функций доступа к спискам со свойствами в книге отсутствуют.

АССОЦИАТИВНЫЕ СПИСКИ

Списки свойств представляют собой один из видов структуры баз данных. *Ассоциативные списки*, или *А-списки*, имеют особую структуру, где пары *значение/свойство* размещаются в двух половинках ячейки связи внутри А-списка. (На голову и хвост ячеек связи иногда ссылаются как на "точечные пары".) Например, А - список с двумя свойствами ЦВЕТ и ФОРМА показан на рис. 7.6. Его структура отличается от приведенной на рис. 7.7:

((ФОРМА ПРЯМОУГОЛЬНИК) (ЦВЕТ КРАСНЫЙ))

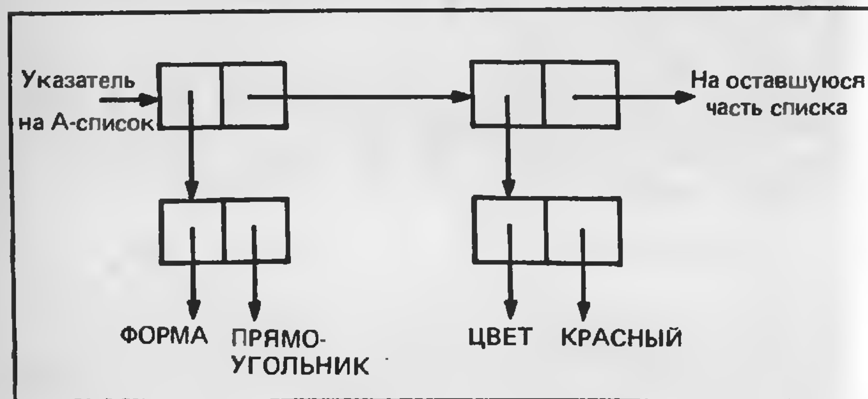


Рис.7.6. Ассоциативный список, или А-список, с теми же данными, что и список свойств, изображенный на рис. 7.5

В А-списках ячейки связи используются более эффективно, чем во вложенных списках, так как в первых реже применяются НУЛИ. В результате они не являются списками свойств.

Слова АССОЦ (ASSOC) и АССОЦ# (ASSOC#) осуществляют доступ к А-спискам. АССОЦ реализует просмотр списка СП в поисках пары свойство/значение, имя свойства которой такое же, как X. Если затем ввести выражение

X СП АССОЦ

то в стек будет помещена пара с именем X. Не обнаружив такой пары, АССОЦ занесет в стек НУЛЬ. ААСОЦ# действует аналогичным образом, только при сравнении заголовков пар использует слово РАВНО (EQUAL) вместо РАВ (EQ). Исходный текст функции доступа к А-спискам АССОЦ приведен на экране 51.

ФУНКЦИИ РАВНО И РАВ

Функция РАВ (EQ) в Лиспе адекватна операции "=" в Форте. Если два указателя в стеке арифметически равны, то как РАВ, так и = помещает в вершину стека истинное значение. Однако два списка, построенные по одной и той же схеме, могут быть разными. Равны ли эти списки? Их указатели не РАВ, хотя сами списки построены идентично. Выведенные операцией ВЫДАТЬ, они похожи.

```
   51
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2 ( С СП1 -> СП2)
 3 : АССОЦ DUP НОЛЬ
 4   IF NIP
 5   ELSE 2DUP ПЕРВЫЙ ПЕРВЫЙ =
 6     IF ПЕРВЫЙ NIP
 7     ELSE ХВОСТ РЕКУРСИЯ
 8     THEN
 9   THEN
10 ;
11
12
13
14
15
```

Экран 51. АССОЦ - слово для построения списков

Чтобы определить, имеют ли одну и ту же структуру два различных списка, применяют слово РАВНО (EQUAL). По отношению к атомам РАВ и РАВНО действуют одинаково, поскольку идентификаторы атомов (в отличие от списков их свойств) не обладают списковой структурой. Они расположены в некоторой области памяти, на которую ссылается указатель (pfa). Итак, различия между этими двумя функциями состоит в следующем: РАВ помещает в вершину стека истину, если одинаковы указатели списков, а РАВНО - если идентична списковая структура. Таким образом, функция РАВНО выполняет больше действий, чем РАВ, поскольку должна сравнить еще и структуры списков. Алгоритм РАВНО показан на рис. 8.4 и будет разъяснен ниже.

Различие между РАВ и РАВНО ведет к различию и между некоторыми словами. АССОЦ и АССОЦ# аналогичны, за исключением того, что АССОЦ при просмотре списка применяет функцию РАВНО, а АССОЦ# - РАВ. Слово ЧЛ (МЕМВ) использует РАВ, в то время как ЧЛЕН (MEMBER) - РАВНО. Еще одной функцией сравнения, помимо РАВ и РАВНО, является НОЛЬ. Если в вершине стека НУЛЬ, это слово доставляет истину. Поскольку несколько функций в Лиспе в случае неудачи оставляют на стеке НУЛЬ,

ФОРМА ПРЯМОУГОЛЬНИК ЦВЕТ КРАСНЫЙ

Рис. 7.7. Вложенный список свойств, содержащий данные А-списка, изображенного на рис. 7.6. Наклонная черта в хвостовой части ячейки связи означает НУЛЬ

данная функция весьма удобна. Например, если УДАЛСВ не может найти заданное свойство, чтобы удалить его из списка, оно помещает в стек НУЛЬ. НУЛЬ во многих случаях действует как ложь, а НОЛЬ используется для того, чтобы проверить наличие лжи на стеке и выдать соответствующее значение флага в терминах Форта. Так как структуры управления взяты из Форта, значения флагов проверяются словами типа IF, WHILE и UNTIL.

НУЛЬ содержит значение НУЛЬ, так что представляет ли НУЛЬ список или идентификатор списка, он все равно будет распознан как НУЛЬ. На самом деле НОЛЬ выбирает значение, на которое ссылается указатель, находящийся на стеке, и сравнивает его с НУЛЕМ, а поскольку это НУЛЬ, то и указывает он тоже на НУЛЬ, т.е. выражение "НУЛЬ @" также дает нуль.

ОПЕРАЦИИ ПРЕОБРАЗОВАНИЯ СПИСКОВ

Список создается при вызове слова СВЯЗЬ. В Лиспе СВЯЗЬ используется некоторыми функциями для построения новых списков путем преобразования исходных. Часто применение функции к существующему списку заключается в изменении указателей в ячейках связи этого списка, что экономит время и память. Например, функция ОБРАТНЫЙ (REVERSE) располагает списки в обратном порядке, выстраивая другой список (с помощью СВЯЗИ), перевернутый по отношению к исходному. Если исходный список можно не сохранять, то больший эффект в таком случае даст операция НОБРАТНЫЙ (DREVERSE), которая меняет направление хвостов исходного списка непосредственно на месте.

В Лиспе существует несколько операций преобразования списков. Основные из них - ЗАМЕНАГ (RPLACA) и ЗАМЕНАХ (RPLACD). Выражение

X Y ЗАМЕНАГ

меняет заголовок X на Y и возвращает модифицированный список. Выражение

X Y ЗАМЕНАХ

аналогичным образом меняет хвост X на Y и возвращает модифицированный список. Похожими по своим действиям словами являются НКОНК(NCONC), НОБРАТНЫЙ, НУДАЛИТЬ (DREMOVE) и ПРИСОЕД (ATTACH), рассматриваемые в следующей главе. Там же мы обсудим и слова преобразования списков, которые наиболее трудны для понимания (поэтому они описываются после изучения деталей реализации).

ДРУГИЕ ФУНКЦИИ РАБОТЫ СО СПИСКАМИ

Иногда требуется несколько списков "слить" в один. Эту операцию для двух списков выполняет слово *СОЕД. Большее число списков можно объединить с помощью функции СОЕД. Предположим, существуют два списка СП1 и СП2, такие что СП1 = (A B), СП2 = (C D). Выполнив операцию

СП1 СП2 *СОЕД

получим:

(A B C D)

При объединении нескольких списков *СОЕД доставляет указатель на результирующий список. Если преобразования списков осуществляются на месте, то результат в стек помещать не нужно. Такое преобразование выполняет функция 2СОЕД. Ее описание приведено на экране 42. Для того чтобы присоединить список СП1 к списку СП2 (причем СП1 должен предварять СП2), нужно ввести:

СП1 СП2 2СОЕД

Стековая нотация для 2СОЕД выглядит следующим образом:

(@СПИСОК1 ->)

Вы уже видели, что при работе со списками часто используются слова ХВОСТ и ПЕРВЫЙ. При работе с длинными списками выписывание громоздких цепочек ХВОСТОВ для того, чтобы добраться к конкретному элементу списка, - занятие утомительное. В таких ситуациях получить, скажем, N-й элемент списка СП, можно с помощью слова NНЫЙ (NTH):

```
СП N NНЫЙ
```

При этом в вершину стека помещается указатель на оставшуюся часть списка с N-м элементов во главе. Выражение

```
СП ПОСЛЕДНИЙ
```

помещает на стек последний элемент, но в случае атома возвращает атом.

Слово ОБРАТНЫЙ (его неразрушительный вариант) перестраивает список, находящийся в стеке, в обратном порядке, а слово УДАЛИТЬ (REMOVE) в приведенном ниже выражении удаляет из СП все вхождения С-выражения С, используя для сравнения операцию РАВНО:

```
С СП УДАЛИТЬ
```

С может быть либо списком внутри СП (так называемым *подсписком*), либо атомом. Приведенное ниже выражение помещает на стек число элементов в списке:

```
СП ДЛИНА
```

Подсчитываются элементы только верхнего уровня списка. Если некоторые элементы внутри данного списка составляют подсписок, то они учитываются как один элемент.

Как было показано выше, слово СПИСОК соединяет С-выражения в список. Список аргументов может быть произвольным, но должен начинаться с НУЛЯ.

```
НУЛЬ A B C D I1 СПИСОК
```

Приведенный выше фрагмент аналогичен следующему:

```
D I1 СВЯЗЬ C I1 СВЯЗЬ B I1 СВЯЗЬ A I1 СВЯЗЬ
```

Слово 2СОЕД отличается от слова СПИСОК тем, что сводит все существующие списки в один, а СПИСОК создает список из выражений.

Для того чтобы определить, является ли список СП1 элементом списка СП2, нужно ввести:


```
    47
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2 UNNEST
 3
 4 ( @СПИСОК -> @ПОСЛЕДНИЙ) \ ВОЗВРАЩАЕТ УКАЗАТЕЛЬ
 5 \ НА ПОСЛЕДНИЙ ЭЛЕМЕНТ СПИСКА С УКАЗАТЕЛЕМ @СПИСОК
 6 : ПОСЛЕДНИЙ DUP ХВОСТ НОЛЬ NOT
 7   IF ХВОСТ РЕКУРСИЯ
 8   THEN
 9 ;
10 \ ИТЕРАТИВНЫЙ ВАРИАНТ СЛОВА ПОСЛЕДНИЙ
11 : ПОСЛЕДНИЙ
12   BEGIN DUP ХВОСТ НОЛЬ NOT
13   WHILE ХВОСТ
14   REPEAT
15
    48
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2 ( @СПИСОК -> N) \ ВОЗВРАЩАЕТ ЧИСЛО ЭЛЕМЕНТОВ В СПИСКЕ
 3 : ДЛИНА DUP НОЛЬ
 4    IF DROP 0
 5    ELSE ХВОСТ РЕКУРСИЯ 1+
 6    THEN
 7 ;
 8
 9 UNNEST
10 \ ДЛИНА. ИТЕРАТИВНЫЙ ВАРИАНТ
11 : ДЛИНА 0
12    BEGIN OVER НОЛЬ NOT
13    WHILE 1+ SWAP ХВОСТ SWAP
14    REPEAT NIP
15
    52
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2 ( СП1 N -> СП2)
 3 : NНЫЙ DUP 1 <
 4    IF 2DROP НУЛЬ
 5    ELSE
 6      BEGIN 1- DUP
 7      WHILE SWAP ХВОСТ SWAP
 8      REPEAT DROP
 9    THEN
10
11
12
13
14
15
```

Экраны 47, 48, 52. Слова построения списков: ПОСЛЕДНИЙ, ДЛИНА и NНЫЙ

СП1 СП2 ЧЛ

(В Лиспе для слова ЧЛ есть синоним ЧЛРАВ - MEMQ.) ЧЛ возвращает остаток от СП2, начинающийся с первого встретившегося элемента, равного (РАВ) СП1. В противном случае на стек помещается НУЛЬ. Слово ЧЛЕН аналогично слову ЧЛ, за исключением того, что оно для сравнения элементов списка СП1 с элементами списка СП2 использует не РАВ, а РАВНО (см. разд. "Функции РАВНО и РАВ").

Наконец, фрагмент

С1 С2 С3 ПОДСТ

```
    49
0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
1 \ НА ФОРТЕ-83
2 ( ЭЛЕМЕНТ @СПИСОК -> @ХВОСТ) \ ЕСЛИ ЭЛЕМЕНТ В СПИСКЕ
3          \ ПРИСУТСТВ. ТО @ХВОСТ БУДЕТ ОСТАТКОМ СПИСКА,
4          \ НАЧИНАЮЩЕГОСЯ С ЭЛЕМЕНТА; ИНАЧЕ НУЛЬ
5 : ЧЛ SWAP OVER НОЛЬ
6   IF 2DROP НУЛЬ
7   ELSE OVER ПЕРВЫЙ OVER =
8     IF DROP
9    ELSE SWAP ХВОСТ РЕКУРСИЯ
10    THEN
11  THEN
12 ;
13
14
15
```

Экран 49. Слово ЧЛ работы со списками

осуществляет подстановку С-выражения С1 вместо всех вхождений С2 в С-выражении С3 и возвращает С3. ПОДСТ сравнивает элементы списка С3 с элементами списка С2 с помощью функции РАВНО. Атомы возвращаются неизмененными.

Слова ПОДСТ (SUBST), УДАЛИТЬ и ОБРАТНЫЙ в Лиспе имеются, но здесь они не реализованы.

УПРАЖНЕНИЯ

1. Для следующего списка:

(A B(C D (E F (G) H) I))

определите результат выполнения следующих операций:

а) ХВОСТ
б) ХВОСТ ПЕРВЫЙ
в) ПЕРВЫЙ ПЕРВЫЙ
г) ХВОСТ ХВОСТ ПЕРВЫЙ ХВОСТ
д) ХВОСТ ХВОСТ ХВОСТ ПЕРВЫЙ
е) ХВОСТ ХВОСТ ПЕРВЫЙ ХВОСТ ХВОСТ ПЕРВЫЙ
ж) ХВОСТ ХВОСТ ПЕРВЫЙ ХВОСТ ХВОСТ ПЕРВЫЙ ХВОСТ ХВОСТ ПЕРВЫЙ
з) ХВОСТ ХВОСТ ХВОСТ ПЕРВЫЙ

2. Объясните, почему в слове ХВОСТ (экран 40) мы не уменьшаем @СПИСОК безусловно. Что случится, если мы это сделаем?

3. Постройте схему, аналогичную приведенной на рис. 7.5, для списка из упр.1, давая подспискам имена.

4. Для всех приведенных ниже списков:

СП1: (С1 (С2 С3) С4)
СП2: НУЛЬ
СП3: (С2 С3)

определите результат применения каждого из следующих выражеиий:

а) С1 СПn ЧЛ ВЫДАТЬ
б) СПn ДЛИНА
в) СПn АТОМ?
г) СП3 @ ЧЛ
д) СП3 ЧЛ

5. Напишите определение слова ОБРАТНЫЙ, располагающего элементы списка в обратном порядке.

6. Напишите определение слова СГЛАЖИВАНИЕ, которое выбирает элементы подсписков и помещает их в список верхнего уровня, например:

(A (B C)(D (E F) G)) --> (A B C D E F G)

7. Напишите определения функций доступа к спискам свойств ПОЛУЧСВ (GETPROP) и ПОМЕСТСВ (PUTPROP).

Глава 8

МЕТОДЫ ПРОГРАММИРОВАНИЯ

В Лиспе как данные, так и команды выражаются в форме списков и взаимозаменяемы. Мы же будем использовать списки только в качестве данных (в прямом их смысле), поскольку интерпретатор Форта не интерпретирует списки - он интерпретирует шитый код. Управляющая логика программы отрабатывается соответствующими традиционными словами на Форте. Форт на самом деле имеет более обширный набор управляющих слов, чем Лисп. Универсальный оператор Форта CASE аналогичен функции COND Лиспа. Но в Лиспе отсутствуют такие конструкции, имеющиеся в Форте, как IF-THEN-ELSE, BEGIN-UNTIL, BEGIN-WHILE-REPEAT. Последние две конструкции являются *примитивами построения циклов.* Они заставляют последовательность команд повторяться неоднократно за счет передачи управления назад, на начало последовательности.

РЕКУРСИЯ

Несмотря на то что Форт предоставляет удобные средства для программирования рекурсивных процессов, к ним очень редко обращаются. Программирование, основанное на списках, составляет исключение. *Рекурсия* имеет место в тех случаях, когда какое-то слово вызывает само себя. Если параметры, используемые данным словом, при этом находятся в стеке, то они во время вызова очередного экземпляра этого слова остаются неизменными. Рекурсию можно представить как вызов *копии* исходного слова. Рекурсивный вызов может выполняться несколько раз и всякий раз параметры предыдущего вызова будут проталкиваться в глубь стека. В конце концов некоторое условие должно завершить последовательность вызовов новых копий. Рассмотрим пример:


```
: ПОСЛЕДНИЙ DUP ХВОСТ НОЛЬ NOT
  IF ХВОСТ РЕКУРСИЯ
  THEN ;
```

Слово ПОСЛЕДНИЙ берет с вершины стека указатель списка и осуществляет проверку на завершение. При этом проверяется, не пуст ли хвост списка. Если хвост - НУЛЬ, выполнение слова заканчивается. В противном случае выбирается хвост списка и вновь вызывается слово ПОСЛЕДНИЙ. Для этого применяется слово РЕКУРСИЯ, так как в Форте не предусмотрено нахождение в словаре слова, определяемого в данный момент. Для иллюстрации выполнения рекурсивных слов предположим, что слово ПОСЛЕДНИЙ было вызвано с указателем в вершине стека, ссылающимся на список: (А В). Последовательность выполнения слов показана стрелками на рис. 8.1.

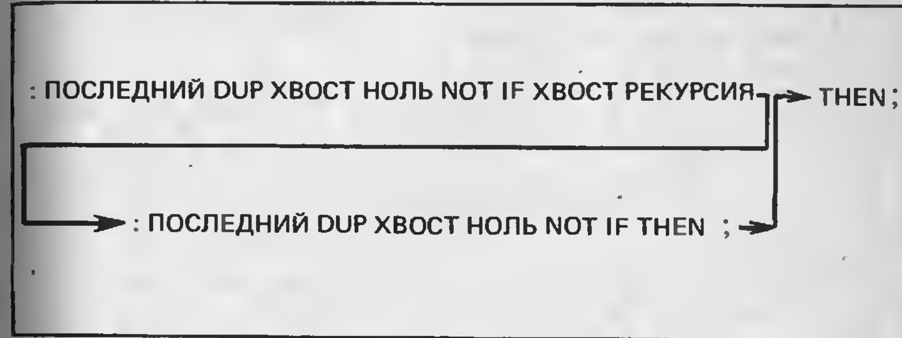


Рис. 8.1. Схема передачи управления в рекурсивном слове ПОСЛЕДНИЙ. РЕКУРСИЯ вызывает копию слова ПОСЛЕДНИЙ (вторая строка), которвя завершает рекурсию

Обратите внимание на то, что здесь две строки содержат определение слова ПОСЛЕДНИЙ. Верхняя строка соответствует исходному вызову со списком (А В) в стеке. Поскольку хвостом (А В) является (В) , т.е. хвост не пустой, то выполняется ветвь, следующая за IF. ХВОСТ от (А В) помещает в стек (В), а затем выполняется слово РЕКУРСИЯ. Оно вновь вызывает слово ПОСЛЕДНИЙ, при этом стрелка ведет ко второй строке. Это слово начинает выполняться, на сей раз с (В) в стеке. Хвост (В) является НУЛЕМ, поэтому ветвь за IF пропускается до THEN и ; , что приводит к выходу из данной копии слова ПОСЛЕДНИЙ и переходу к той, которая представлена верхней строкой, а именно к варианту, вызвавшему копию. Выполнение продолжается с того места, откуда вызывалась РЕКУРСИЯ. Выход из этого варианта осуществляется обычным образом и в стеке остается (В) - последний элемент из (А В).

Если бы список был длиннее, то последовательность списков, остававшаяся в стеке после каждого вызова слова ПОСЛЕДНИЙ, была бы следующей:

(A B C D)
(B C D)
(C D)
(D)

Поскольку всякий раз, когда рекурсивно вызывается слово ПОСЛЕДНИЙ, в стек возвращается хвост заданного списка, у этого списка последовательно отсекаются головы до тех пор, пока не останется один элемент. Такая рекурсия называется *хвостовой*, так как слово последовательно обрабатывает хвост списка. Кроме рассмотренных, хвостовую рекурсию осуществляют слова ДЛИНА и ЧЛ. При использовании ЧЛ ему передаются два параметра, но при каждом рекурсивном вызове ЧЛ ЭЛЕМЕНТ остается прежним, а от списка @СПИСОК - только хвост.

При первом знакомстве рекурсия, как правило, вызывает недоумение. Однако она так же естественна, как любой циклический процесс. Недоумение зачастую происходит от того, что одна и та же процедура обращается сама к себе несколько раз. Однако представьте себе, что всякий раз вызываются отдельные копии, а если это необходимо, то можно называть каждую копию (скажем слова ПОСЛЕДНИЙ) своим именем, например: ПОСЛЕДНИЙ1, ПОСЛЕДНИЙ2 и т.д. Напоминаем, что управление в любую копию передается дважды: первый раз при ее вызове, а второй - при возврате в нее после рекурсивного вызова. В определении слова ПОСЛЕДНИЙ выполняется проверка словом НОЛЬ, а затем словом ХВОСТ из оператора IF-THEN, после чего РЕКУРСИЯ вызывает слово ПОСЛЕДНИЙ в очередной раз. Управление снова возвращается в данную копию, причем в то место определения, откуда произошел рекурсивный вызов, т.е. происходит выход из оператора IF-THEN и из самого определения этой копии слова ПОСЛЕДНИЙ.

Для того чтобы понять, что происходит между рекурсивными вызовами, нужно проследить за изменением содержимого стека (рис. 8.2). Начиная с верхнего левого угла, мы имеем в стеке список(A B C). Каждый последующий вызов слова ПОСЛЕДНИЙ спускает нас на один уровень ниже по левой стороне рисунка. При первом вызове слова ПОСЛЕДНИЙ в момент рекурсии в стеке останется (B C).

После второго рекурсивного вызова в стеке останется (C). Заметьте, что управление все время "крутится" вокруг первой части слова ПОСЛЕДНИЙ - от его начала до слова РЕКУРСИЯ. При рекурсии повторяется одно и то же действие. Список как бы обрезается всякий раз на один элемент. Останавливает такой *рекурсивный спуск* выполнение условия завершения: если хвост списка, переданного очередной копии слова ПОСЛЕДНИЙ, - НУЛЬ, то рекурсия завершается.

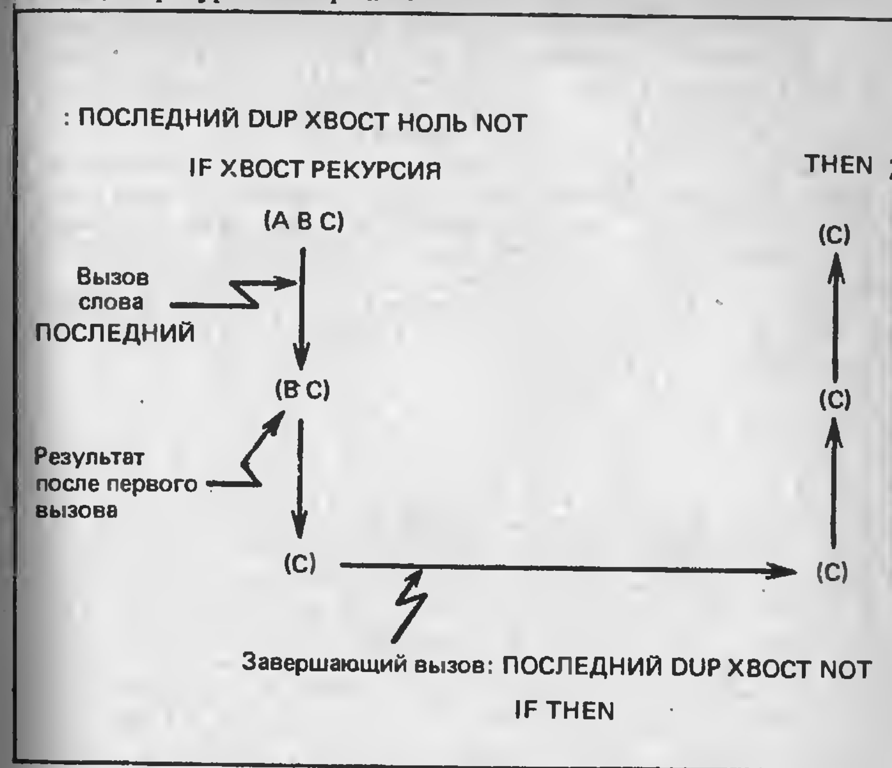


Рис. 8.2. Состояние стека при рекурсивном вызове слова ПОСЛЕДНИЙ и исходном списке в стеке (A B C) . Выполнвние фрагментов Форт-программы, помещенных над вертикальными линиями, для квждого уровня отмечается стрелкой. Фрагмент, выполняемый по финальному вызову (который завершает рекурсию), показан внизу

Когда в третий раз слову ПОСЛЕДНИЙ передается (C), хвост этого списка оказывается НУЛЕМ, что означает пустой список, и та часть оператора IF-THEN, которая вызывает рекурсию, будет обойдена. Третий вызов слова ПОСЛЕДНИЙ (показан в нижней части рис. 8.2) осуществляет возврат. Управление передается фрагменту, следующему за словом РЕКУРСИЯ, в копии слова ПОСЛЕДНИЙ, из которой и был произведен рекурсивный вызов. На схеме рассматриваемая копия находится на следующем относительно дна уровне. Здесь выполняется фрагмент THEN ; и происходит очередной выход из определения. Множественный выход из

копий слова ПОСЛЕДНИЙ продолжается до тех пор, пока не завершится выход из копии самого верхнего уровня.

Во время рекурсивного подъема по стрелкам в правой части рис. 8.2 список не изменяется. Он просто передается с самого нижнего уровня наверх. Изменяется только стек возвратов, из которого при выходе из очередной копии слова ПОСЛЕДНИЙ последовательно убираются адреса возвратов.

В качестве второго, чуть более сложного, примера рассмотрим определения слова ДЛИНА (экран 48). Для сравнения здесь приведены оба варианта - и итерационный, и рекурсивный. В реализации итерационного алгоритма цикл BEGIN-WHILE неоднократно исполняется до тех пор, пока список не будет усечен до НУЛЯ. На каждом шаге выполнения цикла счетчик, значение которого вначале было равно нулю, увеличивается. Состояние стека перед выполнением BEGIN следующее:

(список счетчик)

Рекурсивный вариант действует по аналогичной схеме, но вместо цикла внутри одной и той же копии слова ДЛИНА в нем вызываются последовательные копии, как показано на рис. 8.3. Части определения слова ДЛИНА, соответствующие рекурсивному спуску и подъему, расположены над вертикальными стрелками. Во время спуска выполняется начальный фрагмент определения слова ДЛИНА до слова РЕКУРСИЯ. Результатом выполнения этого фрагмента является усечение списка посредством слова ХВОСТ. При каждом последующем вызове слова ДЛИНА ему передается список, укороченный на один элемент. В конце спуска условие завершения истинно, а список пуст.

В этом последнем вызове слова ДЛИНА (ему соответствует текст в нижней части рисунка) последовательность команд уже иная, так как здесь выбирается ветвь IF, а не ELSE. Операторы данной ветви удаляют из стека пустой список и вместо него помещают начальное значение счетчика, равное нулю, после чего осуществляется возврат из нашей копии слова ДЛИНА в вызывающую копию, которая на рис.8.3 расположена уровнем выше. Поскольку управление возвращается фрагменту, следующему за словом РЕКУРСИЯ, происходит увеличение числа на стеке и возврат. Количество таких увеличений равно количеству уровней. Так как каждый уровень соответствует рекурсивному вызову, во время которого при рекурсивном спуске из списка удаляется один элемент, число прибавлений на обратном пути будет совпадать с числом вызовов.

Другой вид рекурсии - так называемая *двойная рекурсия*. В этом случае слово РЕКУРСИЯ вызывается дважды. Обычно при одном вызове передается голова списка, а при втором - хвост. Примером тому может служить определение слова РАВНО. Его алгоритм показан на рис. 8.4, а исходный текст приведен на экране 53. Стековая нотация данного определения следующая:

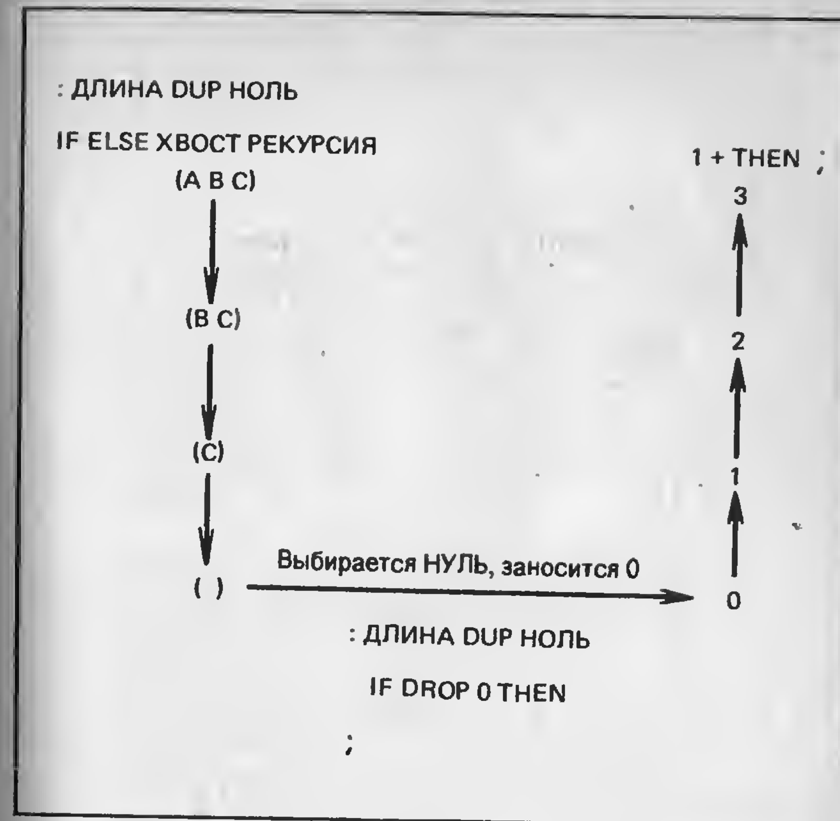


Рис. 8.3. Схема рекурсивных обращений к слову ДЛИНА. При завершающем вызове стек очищается от списка, а счетчик обнуляется

(C1 C2 -> F)

Если C1 и C2 - атомы, то они сравниваются, на стек помещается соответствующий флаг и рекурсия при этом не выполняется. Если же они представляют собой списки, то головы и хвосты C1 и C2 должны совпадать. Первая рекурсия осуществляется по головам всех С-выражений. Она продолжается до тех пор, пока не будут получены атомы, при условии, что сравниваемые головы сами являются списками, после чего исходный вариант слова РАВНО начинает рекурсивное выполнение по хвостам всех списков. Значения флагов от двух рекурсий логически умножаются (операция

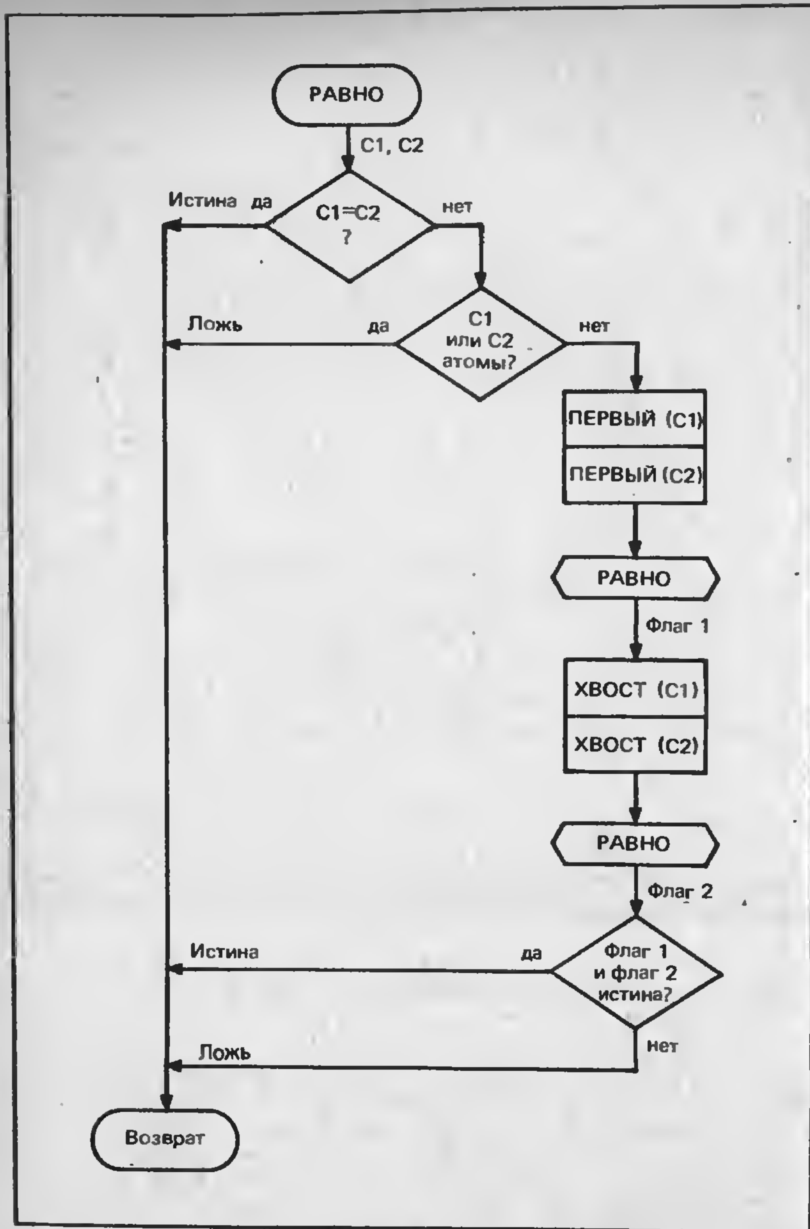


Рис. 8.4. Алгоритм выполнения слова РАВНО (пример двойной рекурсии)

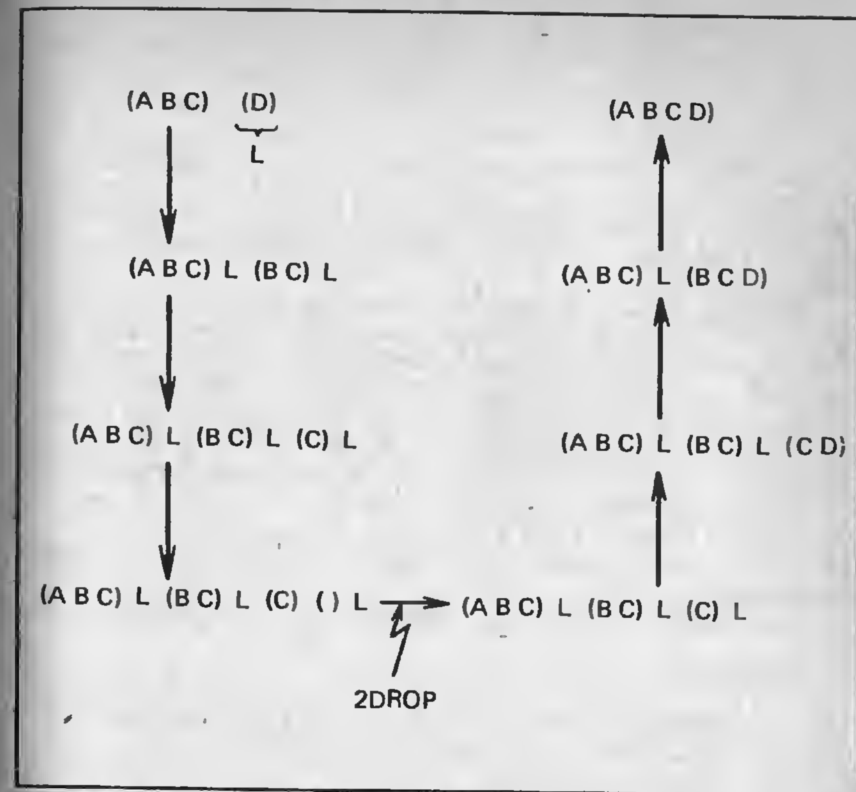


Рис. 8.5. Рекурсивная схема слова 2СОЕД. При каждом рекурсивном вызове передаются два элемента (оба списка). Списки (A B C) и (D) после соединения образуют (A B C D). Список (D) обозначен буквой L

AND), так как для равенства списковых структур оба они должны быть истинными. Общее действие в данном случае заключается в том, что рекурсивно сравниваются головы; затем этот процесс повторяется в виде рекурсии по хвостам.

Последний пример рекурсии, несомненно, сложнее двух предыдущих. На рис. 8.5 показана схема рекурсии, аналогичная схемам из первых двух примеров, но для слова 2СОЕД. Здесь при каждом рекурсивном вызове в стеке находится не один элемент, а несколько. 2СОЕД (текст приводится на экране 42) берет со стека список (@СПИСОК) и присоединяет его к началу другого списка с именем I. Главная идея алгоритма состоит в усечении @СПИСОК и одновременно передачи копии СПИСОК2. Эти копии пригодятся

во время рекурсивного подъема. Когда @СПИСОК становится НУЛЕМ, завершающий вызов 2СОЕД удаляет из стека пустой @СПИСОК, а также соответствующий ему список I и осуществляет возврат, как показано в нижней части рис. 8.5.

При рекурсивном подъеме на стек воздействует фрагмент 2СОЕД, находящийся непосредственно за словом РЕКУРСИЯ. Голова каждого варианта @СПИСОК связывается в I. Таким образом, всякий раз один элемент из исходного варианта @СПИСОК, начиная с конечного хвоста @СПИСОК, последовательно присоединяется к I.

Эти три примера помогут вам детально изучить рекурсивный вызов отдельной функции, и лишь после того, как вы окончательно разберетесь в нем, можно приступить к более абстрактному представлению проблемы. Если для понимания происходящего достаточно следить за ходом рекурсии по диаграмме,то для написания рекурсивных процедур требуются уже определенные знания. Попытаемся проиллюстрировать функционирование слова 2СОЕД на некотором абстрактом уровне мышления. Так как при рекурсии вызываются копии какого-либо слова, они должны выполнять в точности ту же функцию, что и исходное слово при его первоначальном вызове.

В функциональной записи обращение к 2СОЕД выглядит так:

2СОЕД(X,Y)

Слово 2СОЕД берет два аргумента, X и Y, и присоединяет X к Y путем присоединения хвоста списка @СПИСОК к I, или:

2СОЕД(ХВОСТ @СПИСОК, I)

Затем к результату добавляется голова списка @СПИСОК:

2СОЕД(ПЕРВЫЙ @СПИСОК, 2СОЕД (ХВОСТ @СПИСОК, I))

Обратите внимание на то, что слово 2СОЕД рекурсивно, поскольку используется в качестве аргумента к самому себе. Второй аргумент 2СОЕД можно развернуть следующим образом:

2СОЕД(ПЕРВЫЙ @СПИСОК, 2СОЕД(ПЕРВЫЙ ХВОСТ @СПИСОК, I), I)

Слово 2СОЕД внутри последнего выражения также может быть в свою очередь развернуто. Эти последовательные расширения могут продолжаться до тех пор, пока не встретится некоторое условие завершения.

Рассматриваемое слово 2СОЕД является реализацией такого функционального подхода. В подобном представлении остается неясным, что же процедура завершения передает самой вложенной копии слова 2СОЕД. На нижнем уровне рекурсивной схемы последний вызов должен выполнить подготовку к подъему, поскольку условие завершения инициирует переход только один раз. Например, в случае 2СОЕД при подготовке к подъему необходимо удалить из стека два верхних элемента, оставшихся в нем после спуска.

Возможен и другой путь представления процесса рекурсии - как пары последовательных итерационных процедур с процедурой завершения, разделяющей их в нижней части рекурсивной схемы.

Если вы следите за состоянием стека по ходу рекурсивного выполнения какого-то слова, то, подходя к слову РЕКУРСИЯ, нужно учитывать, что состояние стека должно в точности соответствовать его состоянию перед исходным выполнением этого слова (как это определено стековой нотацией для данного слова).

При выполнении слова 2СОЕД РЕКУРСИЯ выбирает из стека два элемента, но не помещает в стек ни одного. Два списка, переданные слову 2СОЕД при исходном обращении, остаются в стеке, а затем копируются посредством OVER ХВОСТ OVER для передачи слову РЕКУРСИЯ. После возврата из слова РЕКУРСИЯ очевидно, что завершилось выполнение

2СОЕД(ХВОСТ @СПИСОК, I)

Далее последовательность SWAP ПЕРВЫЙ SWAP СВЯЗЬ вычисляет ПЕРВЫЙ @СПИСОК, а СВЯЗЬ осуществляет присоединение. То, что РЕКУРСИЯ оставляет в стеке, определяет характер выполнения процедуры завершения. В нашем случае по крайней мере два элемента должны быть удалены из стека. Никаких иных действий не требуется, следовательно, ветвь IF содержит 2DROP. В предыдущем примере со словом ДЛИНА все необходимое для процедуры завершения можно было выяснить путем анализа содержимого стека до и после выполнения слова РЕКУРСИЯ. Список следовало вывести из стека, а число внести, что ветвь IF слова ДЛИНА и делала. Она удаляла посредством DROP из стека указатель списка и помещала в него начальное значение (0).

Двойную рекурсию можно организовать так же, как и обычную хвостовую рекурсию, которая использовалась в приведенных выше примерах. Стандартная схема действий такова: проверка условия завершения, затем выбор головы и хвоста списка, рекурсия по каждому из них и затем объединение результата по какому-нибудь принципу. Для слова РАВНО (рис. 8.4) условием завершения рекурсии является проверка на атомы, а значения флагов, оставляемые в стеке после двух рекурсивных вызовов, логически умножаются посредством AND.

Тщательно изучите приведенные выше примеры рекурсии, постарайтесь выработать свои приемы и понятие рекурсии станет для вас таким же привычным, как и понятие цикла. Слова Лиспа на

экранах с 40 по 49 (см. гл. 7 или листинги исходных текстов) тоже могут служить примерами слов, реализованных с помощью рекурсии.

СБОРКА МУСОРА

Важной составляющей общей реализации Лиспа является управление *списковой памятью* - областью памяти, выделенной под списки. В нашей несколько упрощенной реализации память под списки выделяется статически (во время компиляции), а не динамически. Но в силу того, что управление списковой памятью, как уже отмечалось, играет не последнюю роль для Лиспа в частности и для языков ИИ вообще, мы рассмотрим ее здесь.

Так как ячейки связи, из которых строятся списки, могут освобождаться, требуется некоторый механизм, позволяющий определить, какие ячейки свободны для использования их словом СВЯЗЬ, а какие из них уже заняты. Существуют различные схемы управления списковой памятью, обеспечивающие решение задачи динамического распределения памяти. Основная проблема заключается в распознавании свободных ячеек связи. Этот процесс называется *сборкой мусора.* В большинстве реализаций Лиспа применяется техника *отметить и удалить* (*mark and sweep*). Другая схема, описываемая ниже более подробно, называется *учетом ссылок* (*reference counting*).

Прежде чем объяснить различие между этими методами, мы должны упомянуть о двух нежелательных эффектах при работе с памятью - *висячих ссылках* и *мусоре*, причем первый из них представляет наибольшую опасность. Висячие ссылки появляются в тех случаях, когда некоторая ячейка считается свободной, хотя фактически остается все еще занятой. Если ячейки, входящие как составные части в активные (используемые) списки, ссылаются на ячейку, которая стала считаться свободной, то это не мешает слову СВЯЗЬ данную ячейку задействовать вновь и сделать ее частью нового списка. В результате появляются "висячие" указатели из активных ячеек, так как их значения фактически не определены. Отсюда следует, что нельзя активную ячейку связи делать свободной. Второй эффект противоположен первому. Мы полагаем, что ячейка активна, а она на самом деле свободна. Такая ячейка называется *мусором*, поскольку не может быть повторно использована, и нет способа узнать, что она не занята.

При отметке и удалении мы можем избежать неприятностей, так как разрешаем использовать всю свободную память, а затем собираем мусор. Специальная программа просматривает списки, на которые есть ссылки от идентификаторов, и, поскольку такие списки являются активными, в "бите сборки мусора" каждой активной ячейки делается отметка, показывающая, что данная ячейка не может быть освобождена. Далее просматривается вся списковая память и непомеченные ячейки собираются в отдельный свободный список, откуда будут вновь выделяться словом СВЯЗЬ. Но для задач реального времени схема отметки и удаления не совсем подходит, так как сборка мусора занимает очень много времени, а задача при этом не может выполняться. На больших компьютерах пауза для сборки мусора может занимать секунды. Не дай бог связываться с длительными циклами по непрерывному обновлению памяти!

Учет ссылок не требует длительных пауз для подчистки списковой памяти. Однако такой подход имеет недостаток: программисту приходится вести точный учет использования ячеек памяти, а потому каждая ячейка связи должна иметь дополнительную память для счетчика ссылок. Последний содержит число активных указателей, ссылающихся на данную ячейку. Всякий раз, когда указатель удаляется, счетчик ссылок ячейки, на которую ссылается указатель, уменьшается (посредством RC-), а когда активный указатель начинает ссылаться на эту ячейку, - увеличивается (посредством RC+). Нулевое значение счетчика ссылок свидетельствует о том, что ячейка свободна. Если над списками выполняются неразрушительные операции, то они в процессе выполнения производят копии исходных списков с помощью слова СВЯЗЬ. Счетчики ссылок ячеек таких списков не увеличиваются, так как эти ячейки не приписываются идентификатору и по определению не активны.

Управление счетчиками ссылок осуществляется операторами RC+ и RC-. Они различаются только тем, что RC+ увеличивает значение счетчиков, а RC- их уменьшает. Эти операторы просматривают всю структуру некоторого списка целиком и модифицируют счетчики ячеек каждой ячейки связи данного списка.

Для того чтобы сохранить только ячейки связи, являющиеся частью списка, на который ссылается идентификатор списка, применяется слово УСТАНОВИТЬ (или SETQ в Лиспе). Оно выполняет большую часть работы счетчиков ссылок по обновлению. Слово УСТАНОВИТЬ запускает RC+ на список, когда на тот начинает ссылаться идентификатор списка, и RC- , когда тот же самый идентификатор прекращает ссылки. Таким образом, управление ячейками осуществляется просто словом УСТАНОВИТЬ. Применяемые в настоящее время методы сборки мусора, конечно, более сложны, чем две предложенные вам здесь схемы, и, как правило, представляют их комбинации. Кроме того, современные языки ИИ для эффективного хранения массивов, символьных строк и чисел используют структуры данных, отличные от ячеек связи.

РЕАЛИЗАЦИЯ ФУНКЦИЙ ПРЕОБРАЗОВАНИЯ СПИСКОВ

В Лиспе функции преобразования применяются не для создания модифицированных копий списков, а для модификации структуры самих списков. Ниже будут описаны некоторые такие функции из числа наиболее часто употребляемых.

ЗАМЕНГ инициирует RC- для конкретного списка, а затем вызывает слово ЗАМЕНГ*, которое фактически осуществляет его изменение. Оно замещает заголовок списка заданным С-выражением. ЗАМЕНХ же замещает хвост. Остальные слова преобразования списков используют ЗАМЕНГ* и ЗАМЕНХ* , поскольку каждое слово для того, чтобы привести в соответствие счетчики ссылок, обращается к RC-. Слова ЗАМЕНГ* и ЗАМЕНХ* являются примитивами преобразования списков, поскольку они, в отличие от ЗАМЕНГ и ЗАМЕНХ, не затрагивают счетчики ссылок.

НОБРАТНЫЙ циклически просматривает заданный список СП1 и замещает прежние значения хвостов ячеек связи указателями на предыдущие ячейки, как показано на рис. 8.6. Хвост первого элемента СП1 устанавливается в НУЛЬ, чтобы обозначить конец преобразованного списка. СП2 является указателем обратного списка, поскольку ссылается на его заголовок.

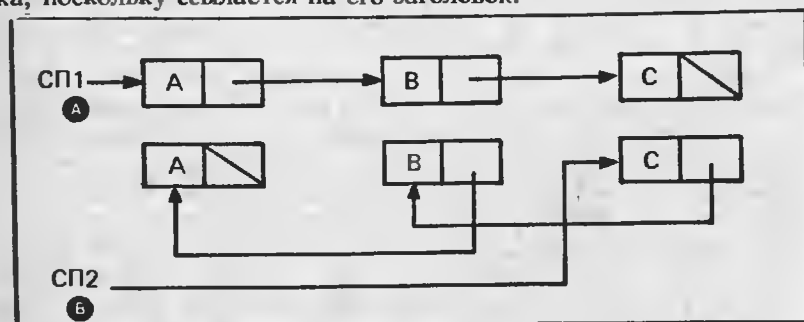


Рис. 8.6. Список СП2, реорганизованный в обратном порядке. Хвостовые значения ячеек связи исходного списка СП1 (вариант А) заменены указателями на предыдущие ячейки (вариант В)

*НКОНК (*NCONC), а также слово НКОНК (NCONC) осуществляет конкатенацию списков СП1 и СП2, замещая НУЛЬ последнего элемента СП1 указателем на СП2. Слово ПРИСОЕД (ATTACH) выбирает из стека аргументы С - С-выражение и СП-список. Если С представляет собой атом, то с помощью слова СВЯЗЬ оно присоединяет С к СП, а если - список, то так же, как и слово *НКОНК, замещает хвост С указателем на СП. С становится начальной частью нового списка.

НУДАЛИТЬ (DREMOVE) использует слово НУДАЛИТЬ1, которое, подобно НУДАЛИТЬ, выбирает С и СП как аргументы и удаляет из СП все ячейки с головами, равными (РАВ) С. Результат выполнения НУДАЛИТЬ показан на рис. 8.7. Хвост ячейки, указывающий на ячейку, голова которой РАВ С, модифицируется таким образом, что указываемая ячейка обходится, ее место

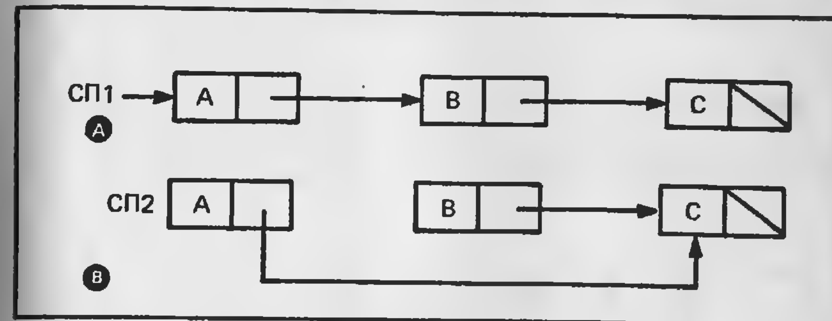


Рис. 8.7. Удаление элемента из списка с помощью слова НУДАЛИТЬ1. Элемент В удаляется из исходного списка СП1 (вариант А), в результате получается список СП2 (вариант В)

занимает следующая и тем самым средняя ячейка исключается из списка. НУДАЛИТЬ проверяет голову первой ячейки СП на равенство (РАВ) С. Если равенство не обнаруживается, то вызывается слово НУДАЛИТЬ1, которое перемещается дальше по СП, обходя ячейки с головами, равными (РАВ) С, путем изменения хвостовых указателей. Однако если должен быть удален первый элемент СП, то его голова и хвост замещаются содержимым этих полей из второго элемента - таким образом осуществляется эффективный обход второго элемента, поскольку хвост первой ячейки теперь указывает на третью ячейку, так как вторую ячейку мы предварительно удалили. Затем для преобразования этого списка можно вызвать слово НУДАЛИТЬ1. Схема выполнения слова НУДАЛИТЬ показана на рис. 8.8.

Функции преобразования списков нужно применять с осторожностью по следующим причинам:

1. Если на некоторый список ранее ссылался идентификатор списка, то после выполнения функции преобразования он может не ссылаться на голову этого списка, так как списки реорганизуются. Например, слово НОБРАТНЫЙ возвратит указатель на голову обратного списка, но идентификатор списка, который указывал на исходный список, теперь ссылается на последний элемент вновь полученного списка. После выполнения функций непосредственного преобразования списков все ссылающиеся на них идентификаторы должны быть переназначены.

2. Поскольку происходит переупорядочение элементов списка, сведения о том, какие ячейки связи освободились, не получаются автоматически. Должно быть выполнено динамическое перераспределение списковой памяти.

```
    54
 0 \ ПРИМЕР РЕКУРСИИ: ВЫЧИСЛЕНИЕ ФАКТОРИАЛА
 1
 2 ( N M -> )
 3 : *ФАКТОРИАЛ OVER 0=
 4   IF NIP
 5   ELSE OVER * SWAP 1- SWAP РЕКУРСИЯ
 6   THEN
 7 ;
 8
 9  ( M -> M! )
10 : ФАКТОРИАЛ 1 *ФАКТОРИАЛ ;
11
12
13
14
15
```

Экран 54. Определение рекурсивных слов ФАКТОРИАЛ и *ФАКТОРИАЛ

3. С помощью функций преобразования легко создать неправильные списки. Часто случайно образуются зацикленные списки. Как правило, они обнаруживаются при печати, инициируемой словом ВЫДАТЬ, одного из них. Вывод таких списков продолжается бесконечно и требует аппаратного переключения или какого-нибудь иного принудительного выхода из системы.

ФУНКЦИИ НЕПОСРЕДСТВЕННОГО ПРЕОБРАЗОВАНИЯ СПИСКОВ И УЧЕТ ССЫЛОК

Функции непосредственного преобразования списков усложняют управление памятью, так как изменяют структуру самого списка, а не возвращают его видоизмененную копию. Если существует идентификатор, ссылающийся на список, который подвергался преобразованиям посредством слова НОБРАТНЫЙ или НУДАЛЕНИЕ, то неясно, что делать со счетчиками ссылок. Не известно, вошли ли ячейки исходного списка в полученный список (в случае НУДАЛИТЬ) или добавлены новые ячейки (в случае ЗАМЕНГ и ЗАМЕНХ). К счастью, есть простой способ справиться с этой проб-

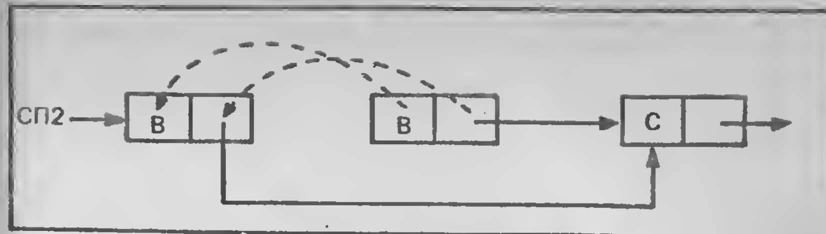


Рис. 8.8. Алгоритм слова НУДАЛЕНИЕ. Если из списка СП1, изображенного на рис. 8.7, должна быть удалена первая ячейка, то происходит замещение головы и хвоста первой ячейки содержимым второй. В противном случае вызывается НУДАЛЕНИЕ1

лемой. Если перед модификацией списка преобразующие слова уменьшают счетчики ссылок списка, используя RC- , то впоследствии ни одна ячейка, изъятая из списка, не становится мусором. Затем выполняется модификация списка.

Здесь необходимо отметить (позднее мы обсудим этот вопрос подробнее), что любой идентификатор измененного списка сейчас неверен, поскольку он продолжает ссылаться на ячейку, которая *была* заголовком списка, но теперь может им не быть. Должна быть выполнена последовательность действий, переназначающая идентификатор новому списку. Сначала выполняется с новым списком RC+ , а затем с помощью функций Форта указатель списка заносится в переменную идентификатора списка. Второй вопрос (более тонкий) состоит в определении влияния реорганизации списка на другие списки, содержащие подсписки вместе с реорганизованным списком. Таких ситуаций желательно избегать, выполняя функции преобразования списков только над теми словами, которые не имеют общих подсписков с другими словами.

УПРАЖНЕНИЯ

1. Используя рекурсию, напишите слово ФАКТОРИАЛ, которое вычисляло бы факториал n. Стековая нотация имеет вид: (n ⟶ n!)

2. Изобразите схему рекурсии слова ФАКТОРИАЛ из упр.1.

3. Напишите слово ОБРАТНЫЙ из упр.5 гл.7 как рекурсивное слово.

4. Напишите слова ЗАМЕНГ и ЗАМЕНХ. Примените их для написания слова НОБРАТНЫЙ.

7 Заказ № 299

Глава 9

ПРОЛОГ - ЯЗЫК РАЗРАБОТКИ СИСТЕМ, ОСНОВАННЫХ НА ЗНАНИЯХ

Компьютерные программы, интерпретирующие продукции, или правила вывода и содержащие знания специалистов, называются *экспертными системами*, или *системами, основанными на знаниях* (или просто *системами знаний*). В настоящей главе рассматривается построение интерпретатора правил вывода с использованием методов, описанных в гл.8. Этот интерпретатор является упрощенным вариантом интерпретатора правил языка Пролог - ведущего языка логического программирования.

ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ НА ПРОЛОГЕ

В 70-е годы из Франции и Шотландии до нас дошел язык логического программирования Пролог. Он был выбран для японского проекта компьютеров пятого поколения. Пролог (PROLOG - PROgramming LOGic) представляет собой простой, но в то же время мощный язык. Он относится в большей степени к *реляционным* или *декларативным* (*описательным*), а не к *функциональным* языкам. Функция - это вид процедуры, которая выбирает переданные ей аргументы, обрабатывает их и возвращает результат. Примером функционального языка может служить Форт. Программа на реляционном языке состоит из операторов, описывающих отношения между представляемыми объектами, на Прологе - в терминах логики предикатов. Предикаты используются для представления фактов и могут быть истинными или ложными. Предикат

```
млекопитающее (козел)
```

утверждает, что "козел является млекопитающим". Этот оператор не предписывает выполнение некоторой процедуры, а лишь декларирует отдельный факт. Программа на Прологе содержит перечень *предложений* (*clause*), которые суть либо факты, либо правила. Предложения записываются в форме:

$$\text{заголовок :- цель}_1,\ \text{цель}_2,\ \ldots,\ \text{цель}_n$$

Заголовок предложения называется заключением правила, в то время как *цели*, разделенные запятыми, - посылками или условиями. Символ :- может восприниматься как слово "если". Цели, следующие за символом "если", в совокупности называются *телом*. Правило при таких обозначениях может быть прочитано так: "Если цель$_1$ и цель$_2$ и ... и цель$_n$ истинны, то заголовок истинен".

Типы данных в Прологе, как это принято в логике, называются *термами*. На рис. 9.1 показано, что термами в Прологе могут быть константы, переменные или структуры. Константы - это либо атомы, либо целые числа. Атомы должны начинаться со строчной буквы или заключаться в одинарные кавычки. Переменные обозначаются именами с прописной первой буквой. Структуры состоят из двух частей: *функтора* (*functor*) - атома, и компонент - термов, заключенных в круглые скобки и разделенных запятыми. Структуры могут применяться в качестве логических предикатов, например:

```
отец(X, 'Билл').
```

Предлагаемый вам терм читается так: "X отец Билла" и является предикатом, поскольку может быть либо истинным, либо ложным. Слово "Билл" заключено в одиночные кавычки, чтобы показать, что это атом. Без кавычек данное слово интерпретировалось бы как переменная, например X. Переменная X и "Билл" считаются компонентами структуры.

Функтор, который может быть использован либо в префиксной, либо в инфиксной записи, представляет собой *оператор*. Например, символ "если" (:-) может применяться в такой записи:

```
:- (a, b, c)
```

или в инфиксной:

```
a :- b, c
```

Не все функторы - операторы, но те, которые мы привыкли рассматривать в качестве операторов (например, :- или +), определены как операторы.

Факты, а также заголовки и цели правил - термы. Помимо термов в Прологе имеются списки. (Обратите внимание: правила

имеют заголовки, а списки - головы, что не всегда одно и то же.) В Прологе списки заключаются в квадратные скобки. Например:

[a,b,c]

Это список из трех элементов, где a - голова, [b,c] - хвост.

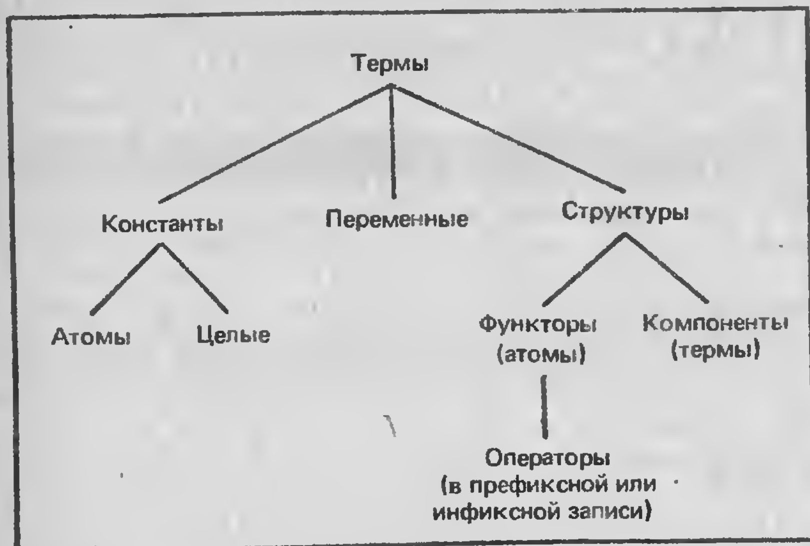


Рис. 9.1. Типы данных в Прологе. Функторы являются атомами, а компоненты - термами. Логические предикаты и арифметические операции представляют собой структуры. Такие операторы, как :- , могут использоваться в инфиксной форме

Пустой список записывается как []. В списке

[Г|X]

вертикальной чертой разделены голова и хвост.

В Прологе имеются встроенные предикаты, такие, как "если" (:-). Ниже приводятся еще несколько встроенных предикатов:

истина (true)
ложь (false)
пер(X) (var(X)) — Возвращает *истину*, если X - переменная
атом(X) (atomic(X)) — Возвращает *истину*, если X - константа

Комментарий в исходном тексте заключается в парные симметричные слэжи-звездочки:

/* ... */

Для всех программ в Прологе предусмотрена одна процедура вывода. Поскольку прикладные области применения Пролога весьма обширны, эта процедура, без сомнения, должна быть мощной и универсальной. Речь идет о процедуре поиска, дополненной правилом вывода, известным как *резолюция*. Вместо того чтобы иметь несколько правил вывода, по которым из существующих предложений логически выводятся новые факты, вполне достаточно пользоваться лишь резолюцией, что упрощает дело, в частности при выборе применимого правила на каждом шаге процесса рассуждения.

Правила, записываемые программистом как предложения, не являются правилами вывода Пролога, а представляют собой логическую импликацию и эквивалентны следующей записи:

$$\text{заголовок} \vee \neg\text{цель}_1 \vee \neg\text{цель}_2 \vee \ \ldots \vee \neg\text{цель}_n$$

где v - знак логической операции ИЛИ (дизъюнкции), & - знак логической операции И (конъюнкции), а ¬ - операция НЕ (логическое отрицание). Предложения Пролога связаны между собой логической конъюнкцией, т.е. в совокупности они истинны.

Теперь рассмотрим два предложения:

цель, заголовок :- цель

Первое предложение - факт, а второе - правило, которое логически может быть выражено связкой ¬цель v заголовок. Поскольку предложения объединяются конъюнкцией, такая запись эквивалентна логическому выражению:

цель & (¬цель v заголовок)

Так как выражение цель & ¬цель всегда ложно, то результирующим выражением должно быть следующее: цель & заголовок. Резолюция есть правило вывода, объединяющее два выражения - факт и правило. При этом цель правила сопоставляется с фактом и выводится новый факт - заголовок правила. В результате цель продолжает оставаться в числе предложений, а заголовок добавляется к их числу. Как факты, так и правила являются предложениями и имеют одну и ту же форму, поскольку факт это то же правило, но без целей:

факт :-

Иными словами, факт можно считать правилом, которое выполняется безусловно, не имея целей. Отрицания фактов могут быть выражены следующим образом:

:- факт

что эквивалентно выражению ￢факт по определению логической импликации.

Прежде чем описывать процедуру поиска, рассмотрим простой пример программы на Прологе. В гл.8 вам демонстрировалась рекурсия в определении слова 2СОЕД. Эта функция на Прологе может быть выражена в виде логического предиката (структуры Пролога):

соед(СП1,СП2,СП3)

означающего: "Соединение списков СП1 и СП2 дает в результате список СП3". Например, для СП1 (A B) и СП2 (C D) СП3 будет (A B C D). На Прологе функция соед может быть выражена следующим образом:

```
соед([],СП2,СП2)
соед([Г|СП1],СП2,[Г|СП3]) :- соед(СП1,СП2,СП3)
```

Первое предложение подобно условию завершения слова 2СОЕД. Оно означает, что если к СП2 присоединяется [] (или НУЛЬ), то в результате получается СП2. Во втором предложении содержится слово соед как в заголовке, так и в цели, что означает рекурсию. Если СП1 присоединяется к СП2 для образования СП3, то список с заголовком Г и хвостом СП1, присоединенный к СП2, даст в результате список с головой Г и хвостом СП3.

Чтобы проверить слово соед в действии, присоединим (a b) к (c d). Успешными подстановками в заголовок правила являются следующие:

Г	СП1	СП2	[Г|СП3]
a	(b)	(c d)	[a|СП3]
b	( )	(c d)	[a|[b|СП3]]

Пустой список СП1 в последней строке запускает условие завершения для слова соед. Теперь подбирается не правило, а факт:

соед([],СП2,СП2)

Здесь мы выполняем такие подстановки:

[]	СП2	[Г|СП3]
()	(c d)	[a|[b|(c d)]]

При последнем подборе вместо СП3 подставляется СП2. В результате получаем список:

[a|[b|(c d)]] -> [a b c d]

Обратите внимание: в то время как [Г|СП1] обеспечивает сопоставление с (a b), третий аргумент правила, [Г|СП3], конструирует список. Обратное действие состоит в подстановке СП3 из цели в СП3 заголовка. Это ведет к тому, что предыдущая подстановка, скажем [a|СП3] для СП3, заменит текущий список СП3 в [b|СП3]:

[a|[b|СП3]]

Во время последней итерации СП2 заменит СП3, и реорганизация списка будет завершена.

ИНТЕРПРЕТАТОР ПРОЛОГА

Повторяющееся применение фактов к правилам, приводящее к новым фактам, называется *прямым рассуждением* (*forward chaining*). При *обратном рассуждении* (*backward chaining*) осуществляется поиск заключений (заголовков правил), соответствующих цели. Если такое заключение найдено, то в свою очередь должны быть доказаны цели, входящие в данное правило. В Прологе используется обратное рассуждение.

В Прологе поиск осуществляется в глубину. Алгоритм поиска показан на рис.9.2. Здесь задействованы три списка. Первый, ПРЕДЛОЖЕНИЯ (CLAUSES), содержит предложения. Во втором, ЦЕЛИ (GOALS), находятся цели, подлежащие удовлетворению. Третий список, РЕШЕНИЯ (SOLVED), включает точки возврата (см. ниже) и хранит следы предложений, применявшихся для достижения целей. Это своего рода трасса нахождения решения процедурой поиска.

Как показано на рис. 9.2, процедура поиска ПОИСК (SEARCH) просматривает список ЦЕЛИ. Если он пуст, то, значит, не осталось ни одной цели, которую нужно доказать, и поиск считается успешным. В противном случае удаляется первая цель из списка ЦЕЛИ, и сканирование продолжается по списку ПРЕДЛОЖЕНИЯ в поисках подходящего предложения. Если таковое найдено, то указатель (в списке ПРЕДЛОЖЕНИЯ) на это предложение вместе с целью добавляется к списку РЕШЕНИЯ. Указатель отмечает, как далеко процедура ПОИСК продвинулась в списке ПРЕДЛОЖЕНИЯ перед нахождением нужного предложения. Затем проверяются цели выбранного предложения. Если хотя бы одна из них недостижима, указатель продвигается до следующего подходящего предложения в списке ПРЕДЛОЖЕНИЯ и его цели

помещаются в список ЦЕЛИ. Такая реакция на неудачу называется *возвратом*. Этот новый указатель замещает прежний в списке РЕШЕНИЯ, и всякий раз, когда совершается возврат, продвигается далее по списку ПРЕДЛОЖЕНИЯ. Если указатель достигает конца списка ПРЕДЛОЖЕНИЯ, то, следовательно, в списке нет предложений, доказывающих искомые цели, и ПОИСК завершается неудачей.

Чтобы объяснить действие процедуры поиска, нам потребуется пример списка предложений и цель. Предложения будут представлены в форме, принятой в данной реализации (см. экраны с 60 по 65). Для нашего примера на экране 70 приводится небольшая база знаний. Список ПРЕДЛОЖЕНИЯ содержит следующие предложения:

1. (ДАЕТ-МОЛОКО)
2. (ИМЕЕТ-ВОЛОС-ПОКРОВ)
3. (ИМЕЕТ-РОГА)
4. (МЛЕКОПИТАЮЩЕЕ ДАЕТ-МОЛОКО ИМЕЕТ-ВОЛОС-ПОКРОВ)
5. (КОЗЕЛ БОРЬКА)
6. (КОЗЕЛ МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-РОГА)

Предложения 1, 2, 3 и 5 - факты; предложения 4 и 6 - правила. Список ЦЕЛИ выглядит как (КОЗЕЛ). Итак, процедура ПОИСК будет пытаться доказать цель КОЗЕЛ, используя утверждения, содержащиеся в списке ПРЕДЛОЖЕНИЯ. Список РЕШЕНИЯ первоначально пуст.

Как показано на рис. 9.2, ПОИСК в первую очередь проверяет, не пуст ли список ЦЕЛИ, а затем удаляет из списка ЦЕЛИ объект КОЗЕЛ, после чего он становится пустым. Теперь ПОИСК находит первое предложение с заголовком, сопоставимым с фактом КОЗЕЛ (предложение 5), и помещает утверждение КОЗЕЛ, а также указатель на 5 в список РЕШЕНИЯ. Поскольку предложение (предложение 5) выбрано, так как его заголовок совпал с фактом КОЗЕЛ, тело данного правила добавляется к списку ЦЕЛИ. Теперь в списке ЦЕЛИ появляется содержимое:

(БОРЬКА)

Поиск продолжается. Цель удаляется из списка ЦЕЛИ, и ищется предложение, удовлетворяющее факту БОРЬКА. Так как его не существует, поиск заканчивается неудачей и происходит возврат.

Итак, список ЦЕЛИ пуст, а предыдущая цель (КОЗЕЛ) вместе с указателем на предложение 5 восстанавливается из списка РЕШЕНИЯ. Пара указатель/цель удаляется из списка РЕШЕНИЯ и предпринимается новая попытка поиска цели, начиная с предложения, следующего за предложением 5. Теперь уже предложе-

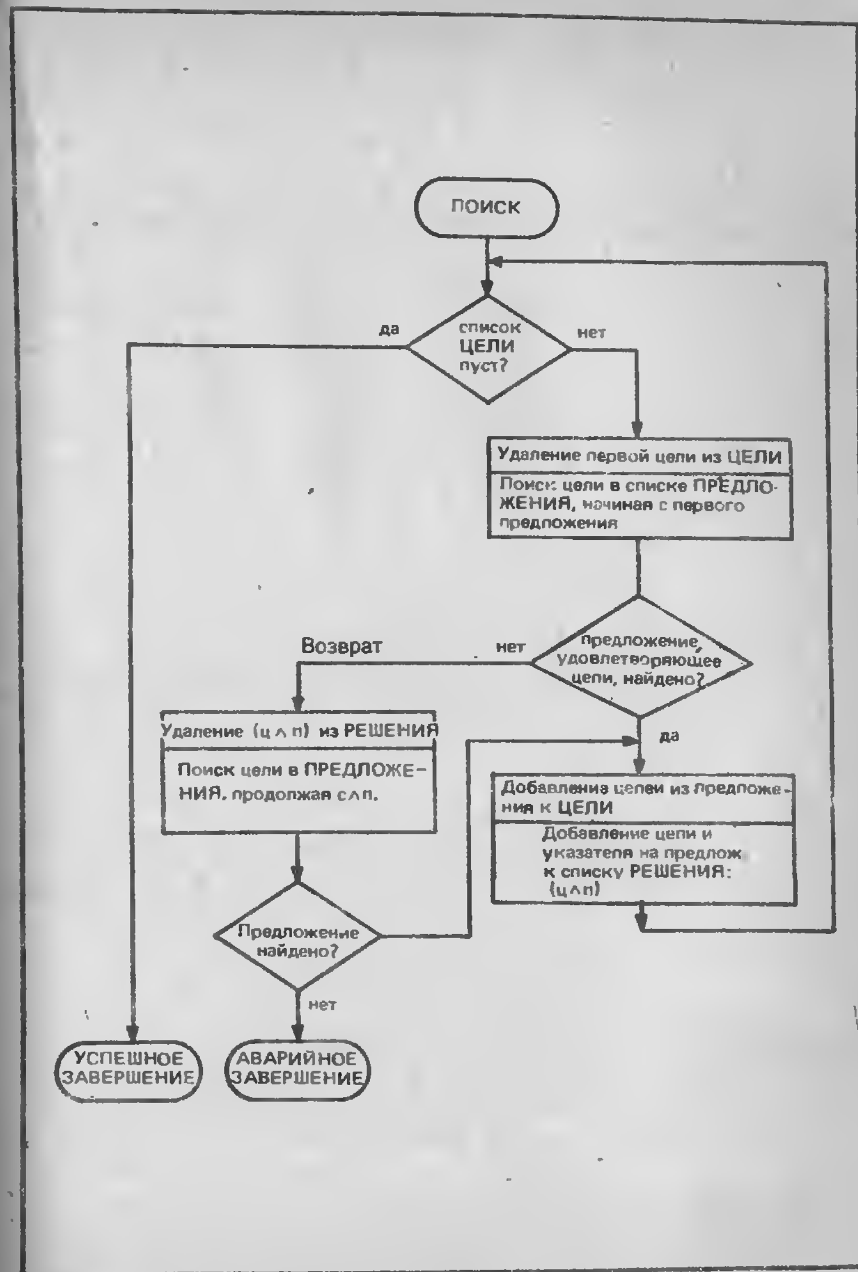


Рис. 9.2. Алгоритм поиска в глубину

ние 6 в списке ПРЕДЛОЖЕНИЯ сопоставляется с утверждением КОЗЕЛ. Это новая точка возврата, поэтому и КОЗЕЛ, и предложение 6 помещаются в список РЕШЕНИЯ. Цели предложения 6 добавляются к списку ЦЕЛИ. Возврат закончился и так как предложение, совпадающее с фактом КОЗЕЛ найдено (предложение 6), ПОИСК не завершает аварийно работу, а продолжает выполнение по описанному выше алгоритму до тех пор, пока не будут достигнуты все цели, или произведен возврат по последнему предложению из списка ПРЕДЛОЖЕНИЯ. Если поиск завершается успешно, то список РЕШЕНИЯ содержит следы пути поиска, показывающие, каким образом была доказана истинность цели. В случае же неудачи в списке РЕШЕНИЯ остается частичное решение, по которому можно судить о том, как далеко продвинулось выполнение процедуры ПОИСК перед аварийным завершением.

Продемонстрируем работу интерпретатора правил на примере. Загрузив экран 70, вы скомпилируете в словарь Форта описанную выше базу знаний и создадите списки. Активируя слово СЛЕД, вы можете нажатием любой клавиши останавливать выполнение, а затем продолжать его, причем на каждом шаге вывода (т.е. на каждом шаге цикла поиска) на экран будут выводиться списки ЦЕЛИ и РЕШЕНИЯ. На рис. 9.3 показаны результаты последовательного выполнения слова СЛЕД. Каждый фрейм ЦЕЛИ-РЕШЕНИЯ разделен пустой строкой.

Первый напечатанный фрейм характеризует начальное состояние списков перед первым выводом. В каждом последующем

```
    70
0 \ БАЗА ЗНАНИЙ
1 : МАРКЕР ;
2 ПРАВИЛО: ДАЕТ-МОЛОКО   (ДАЕТ-МОЛОКО )
3 ПРАВИЛО: ИМЕЕТ-ВОЛОС-ПОКРОВ  (ИМЕЕТ-ВОЛОС-ПОКРОВ )
4 ПРАВИЛО: ИМЕЕТ-РОГА    (ИМЕЕТ-РОГА )
5 ПРАВИЛО: МЛЕКОПИТАЮЩЕЕ (МЛЕКОПИТАЮЩЕЕ
6                          ДАЕТ-МОЛОКО
7                          ИМЕЕТ-ВОЛОС-ПОКРОВ )
8 ПРАВИЛО: КОЗЕЛ1   (КОЗЕЛ БОРЬКА )
9 ПРАВИЛО: КОЗЕЛ2   (КОЗЕЛ МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-РОГА )
10 ПРЕДЛОЖЕНИЯ НУЛЬ УСТАНОВИТЬ ПРЕДЛОЖЕНИЯ ЧТСП
11 (ДАЕТ-МОЛОКО @ ИМЕЕТ-ВОЛОС-ПОКРОВ  @ ИМЕЕТ-РОГА
       @ МЛЕКОПИТАЮЩЕЕ @ КОЗЕЛ1 @ КОЗЕЛ2 @ )
13 РЕШЕНИЯ НУЛЬ УСТАНОВИТЬ
14 ЦЕЛИ НУЛЬ УСТАНОВИТЬ
15 НУЛЬ КОЗЕЛ ЦЕЛИ СПИСОК
```

Экран 70. База знаний

фрейме из списка ЦЕЛИ удаляется каждая недостигнутая цель и добавляются цели из вновь выбранного предложения. Это предложение также появляется в виде первого элемента в списке РЕШЕНИЯ. При первом выводе (рис. 9.3) цель БОРЬКА не достигается. Во втором фрейме БОРЬКА удален из списка ЦЕЛИ, а цели первого предложения из списка РЕШЕНИЯ добавлены. Это предложение в списке РЕШЕНИЯ было найдено при возврате. Последний фрейм показывает, что все цели достигнуты, так как список ЦЕЛИ представляет собой НУЛЬ. Список РЕШЕНИЯ содержит последовательность вывода, примененную для доказательства ЦЕЛИ из списка РЕШЕНИЯ.

На экране 71 приводится несколько расширенная база знаний, в которой предложения расположены так, что дважды совершается возврат для утверждения КОЗЕЛ и один раз для утверждения ИМЕЕТ-ВОЛОС-ПОКРОВ. Поскольку факты ни БОРЬКА, ни ГРУБИЯН не могут быть доказаны, а факт ГРУБИЯН оказывался целью дважды, произошло три возврата.

РЕАЛИЗАЦИЯ ПОИСКА

Реализация алгоритма процедуры ПОИСК представлена на экранах с 60-го по 63-й, а весь "пакет" - на экранах с 60-го по 65-й. Она построена на использовании лиспоподобных слов, описанных в гл.7, а также программ, приведенных на экранах 70 и 71. Поскольку сам алгоритм поиска был уже рассмотрен выше, остановимся на деталях реализации.

На экране 63 показано слово ПОИСК, содержащее простой цикл повторения слова (ПОИСК), приведенного на экране 62. Слово (ПОИСК) выполняет свои функции и передает слову ПОИСК флаг успех/неудача, по которому и определяется момент завершения. В первую очередь (ПОИСК) посредством слова НОЛЬ проверяет список ЦЕЛИ. Если список не пуст, то с помощью команды ПОЛУЧИТЬ-ЦЕЛЬ определяется цель из списка ЦЕЛИ. Затем (ПОИСК) получает указатель на заголовок списка ПРЕДЛОЖЕНИЯ, чтобы приступить к поиску цели с начала списка.

Слово НАЙТИ-ПРЕДЛОЖЕНИЕ? получает цель и указатель в список ПРЕДЛОЖЕНИЯ и пытается найти предложение с заголовком, сопоставимым с целью. Если поиск завершается успешно, то, помимо всего, цель и указатель на выбранное предложение помещаются в список РЕШЕНИЯ. В стек возвращается флаг, который истинен, если предложение было найдено.

В слове (ПОИСК) выбор предложения осуществляет ветвь IF. В ней активируется слово ДОБАВИТЬ-ЦЕЛИ, которое помещает в список ЦЕЛИ цели выбранного предложения (указатель на него находится в списке РЕШЕНИЯ). Если подходящего предложения не оказалось, требуется возврат: выбирается ветвь ELSE и вызывается слово ВОЗВРАТ. Какая бы из ветвей ни выполнялась,

```
     71
0 \ БАЗА ЗНАНИЙ
1 : МАРКЕР ; ПРАВИЛО: ДАЕТ-МОЛОКО   (ДАЕТ-МОЛОКО )
2 ПРАВИЛО: ИМЕЕТ-ВОЛОС-ПОКРОВ1
3        (ИМЕЕТ-ВОЛОС-ПОКРОВ ГРУБИЯН )
4 ПРАВИЛО: ИМЕЕТ-ВОЛОС-ПОКРОВ2 (ИМЕЕТ-ВОЛОС-ПОКРОВ )
5 ПРАВИЛО: ИМЕЕТ-РОГА (ИМЕЕТ-РОГА )
6 ПРАВИЛО: МЛЕКОПИТАЮЩЕЕ (МЛЕКОПИТАЮЩЕЕ
7  ДАЕТ-МОЛОКО ИМЕЕТ-ВОЛОС-ПОКРОВ ) ПРАВИЛО: КОЗЕЛ1
8 (КОЗЕЛ БОРЬКА ) ПРАВИЛО: КОЗЕЛ2  (КОЗЕЛ ГРУБИЯН )
9 ПРАВИЛО: КОЗЕЛ3  (КОЗЕЛ МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-РОГА )
10  ПРЕДЛОЖЕНИЯ НУЛЬ УСТАНОВИТЬ ПРЕДЛОЖЕНИЯ ЧТСП
11  (ДАЕТ-МОЛОКО @ ИМЕЕТ-ВОЛОС-ПОКРОВ1 @
12  ИМЕЕТ-ВОЛОС-ПОКРОВ2 @ ИМЕЕТ-РОГА @ МЛЕКОПИТАЮЩЕЕ
13   @ КОЗЕЛ1 @ КОЗЕЛ2 @ КОЗЕЛ3 @ )
14 РЕШЕНИЯ НУЛЬ УСТАНОВИТЬ ЦЕЛИ НУЛЬ УСТАНОВИТЬ
15 НУЛЬ КОЗЕЛ ЦЕЛИ СПИСОК
```

Экран 71. Расширенная база знаний

она вносит в вершину стека флаг, предназначенный для UNTIL в слове ПОИСК. При ложном значении флага поиск продолжается, при истинном - завершается и на стек помещается следующий флаг, означающий успех или неудачу.

Для самой внешней конструкции IF-ELSE-THEN в слове (ПОИСК) ветвь ELSE выбирается тогда, когда список ЦЕЛИ пуст. На стеке остаются два флага: первый для завершения, второй для обозначения успеха. Слово ВОЗВРАТ, стоящее перед IF-THEN, осуществляет манипуляции с флагом. При успешном завершении ВОЗВРАТА оно передает истинное значение флага, ПОИСК не завершает свое выполнение и выдает ложный флаг. Если же возврат заканчивается неудачей, возвращается ложный флаг и в стеке остаются флаги в таком порядке: ложь-истина.

Слова НАЙТИ-ПРЕДЛОЖЕНИЕ? и ВОЗВРАТ приводятся на экране 61. Как (ПОИСК), так и ВОЗВРАТ используют слово НАЙТИ-ПРЕДЛОЖЕНИЕ?, а оно в свою очередь - слово НАЙТИ-ПРЕДЛОЖЕНИЕ, изображенное на экране 60, для фактического поиска подходящего предложения. НАЙТИ-ПРЕДЛОЖЕНИЕ получает цель и указатель на ПРЕДЛОЖЕНИЯ, а затем ищет предложение, заголовок которого сопоставим с целью. Если подходящее предложение обнаруживается, то в стек помещается как цель, так и указатель на выбранное предложение. Поскольку ПРЕДЛОЖЕНИЯ есть список предложений, каждое из которых само


```
ЦЕЛИ:    (КОЗЕЛ)
РЕШЕНИЯ: НУЛЬ

ЦЕЛИ:    (БОРЬКА)
РЕШЕНИЯ: (КОЗЕЛ БОРЬКА)

ЦЕЛИ:    (МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-РОГА)
РЕШЕНИЯ: (КОЗЕЛ МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-РОГА)

ЦЕЛИ:    (ДАЕТ-МОЛОКО ИМЕЕТ-ВОЛОС-ПОКРОВ ИМЕЕТ-РОГА)
РЕШЕНИЯ: (МЛЕКОПИТАЮЩЕЕ ДАЕТ-МОЛОКО
              ИМЕЕТ-ВОЛОС-ПОКРОВ)
              (КОЗЕЛ МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-РОГА)

ЦЕЛИ:    (ИМЕЕТ-ВОЛОС-ПОКРОВ ИМЕЕТ-РОГА)
РЕШЕНИЯ: (ДАЕТ-МОЛОКО)
      (МЛЕКОПИТАЮЩЕЕ ДАЕТ-МОЛОКО ИМЕЕТ-ВОЛОС-ПОКРОВ)
      (КОЗЕЛ МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-РОГА)

ЦЕЛИ:    (ИМЕЕТ-РОГА)
РЕШЕНИЯ: (ИМЕЕТ-ВОЛОС-ПОКРОВ) (ДАЕТ-МОЛОКО)
      (МЛЕКОПИТАЮЩЕЕ ДАЕТ-МОЛОКО ИМЕЕТ-ВОЛОС-ПОКРОВ)
      (КОЗЕЛ МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-РОГА)

ЦЕЛИ:    НУЛЬ
РЕШЕНИЯ: (ИМЕЕТ-РОГА) (ИМЕЕТ-ВОЛОС-ПОКРОВ)
      (ДАЕТ-МОЛОКО)
      (МЛЕКОПИТАЮЩЕЕ ДАЕТ-МОЛОКО
       ИМЕЕТ-ВОЛОС-ПОКРОВ)
      (КОЗЕЛ МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-РОГА)

УСПЕШНОЕ ЗАВЕРШЕНИЕ
```

Рис. 9.3. След поиска

является списком, указатель на ПРЕДЛОЖЕНИЯ будет ссылаться на некоторый список, например такой:

```
((КОЗЕЛ БОРЬКА)
(КОЗЕЛ МЛЕКОПИТАЮЩЕЕ ИМЕЕТ РОГА))
```

что соответствует второму фрейму на рис. 9.4.

Чтобы получить заголовок первого предложения этого "урезанного" (посредством слова ХВОСТ) списка ПРЕДЛОЖЕНИЯ, потребуется два раза применить слово ПЕРВЫЙ. После первого применения возвратится первое предложение, после второго - его заголовок. Затем указатель на заголовок предложения в теле НАЙТИ-ПРЕДЛОЖЕНИЕ будет сравниваться (посредством =) с

целью. Так как имя данной цели и указатель на нее представляет собой простую переменную Форта, то для сравнения можно воспользоваться операцией = . (В полном Прологе операция = замене-

```
    62
 0 \ ИНТЕРПРЕТАТОР ПРАВИЛ ПРОЛОГА
 1 ( -> ФЛАГ ИСТИНА|ЛОЖЬ) \ ФЛАГ=ИСТИНА => УСПЕШНОЕ
 2                  \ ЗАВЕРШЕНИЕ
 3 : (ПОИСК)
 4   ЦЕЛИ @ НОЛЬ NOT
 5   IF ПОЛУЧИТЬ-ЦЕЛЬ ПРЕДЛОЖЕНИЯ @ НАЙТИ-ПРЕДЛОЖЕНИЕ?
 6     IF ДОБАВИТЬ-ЦЕЛИ FALSE
 7     ELSE ВОЗВРАТ
 8       IF FALSE ELSE FALSE TRUE THEN
 9     THEN
10   ELSE TRUE DUP
11   THEN
12 ;
13
14
15
```

Экран 62. Интерпретатор правил Пролога: (ПОИСК)

на операцией UNIFY (УНИФИКАЦИЯ), объяснение которой приводится ниже.) Если сопоставления не произошло, а в списке еще остались элементы, то указатель хвоста предложении переместится на следующее предложение и сравнение повторится. После того как слово НАЙТИ-ПРЕДЛОЖЕНИЕ завершит свое выполнение, оно возвратит указатель. Если ни одно из сравнений не сработало, этот указатель будет ссылаться на пустой список предложений.

Слово НАЙТИ-ПРЕДЛОЖЕНИЕ? проверяет, не является ли возвращенный список (указатель) НУЛЕМ (применяя НОЛЬ) и помещает копию полученного флага (используя >R) в стек возвратов. Если поиск закончился неудачей, стек очищается, и флаг передается снова. Если же поиск прошел успешно, то как цель, так и указатель предложения связываются в список РЕШЕНИЯ. Таким образом, структура списка РЕШЕНИЯ следующая:

(цель$_1$ указатель-предложения$_1$ цель$_2$ указатель-предложения$_2$ ... цель$_n$ указатель-предложения$_n$)

На рис. 9.4 показано только первое предложение из оставшейся части списка ПРЕДЛОЖЕНИЯ, на которое ссылается указатель предложения.

Слово ВОЗВРАТ прежде всего восстанавливает цель и указатель на предложение (последний отмечает точку возврата в списке ПРЕДЛОЖЕНИЯ) из списка РЕШЕНИЯ, а затем устанавливает его в качестве нового значения в хвост списка РЕШЕНИЯ, т.е. оно продвигается по списку через два элемента, удаляя только что восстановленные цель и указатель на предложение. Далее, чтобы возобновить поиск цели, начиная с указателя пред-

```
    60
 0 \ ИНТЕРПРЕТАТОР ПРАВИЛ ПРОЛОГА
 1
 2 20   НОВСПИСОК ЦЕЛИ
 3 20   НОВСПИСОК РЕШЕНИЯ
 4 200  НОВСПИСОК ПРЕДЛОЖЕНИЯ
 5
 6 ( -> ЦЕЛЬ)
 7 : ПОЛУЧИТЬ-ЦЕЛЬ ЦЕЛИ @ DUP ПЕРВЫЙ SWAP ХВОСТ
 8   ЦЕЛИ  SWAP УСТАНОВИТЬ ;
 9 ( ЦЕЛЬ @ПРЕДЛОЖЕНИЯ1 -> ЦЕЛЬ @ПРЕДЛОЖЕНИЯ2)
10 : НАЙТИ-ПРЕДЛОЖЕНИЕ
11   BEGIN 2DUP ПЕРВЫЙ ПЕРВЫЙ DUP >R = R> НОЛЬ OR NOT
12   WHILE ХВОСТ
13   REPEAT
14 ;
15

    61
 0 \ ИНТЕРПРЕТАТОР ПРАВИЛ ПРОЛОГА
 1 ( ЦЕЛЬ @ ПРЕДЛОЖЕНИЕ -> ФЛАГ) \ ПОИСК
 2    \ ПОДХ. ПРЕДЛОЖЕНИЯ И ПОМЕЩ. В РЕШЕН.
 3    \ ФЛАГ ИСТИНЕН, ЕСЛИ ПРЕДЛОЖЕНИЕ НАЙДЕНО
 4 : НАЙТИ-ПРЕДЛОЖЕНИЕ? НАЙТИ-ПРЕДЛОЖЕНИЕ
 5   DUP НОЛЬ DUP >R
 6    IF 2DROP
 7    ELSE РЕШЕНИЯ СВЯЗЬ РЕШЕНИЯ СВЯЗЬ
 8    THEN R> NOT ;
 9 : ДОБАВИТЬ-ЦЕЛИ РЕШЕНИЯ @ ХВОСТ ПЕРВЫЙ
10   ПЕРВЫЙ ХВОСТ ЦЕЛИ 2СОЕД ;
11 ( -> ФЛАГ) \ ФЛАГ = ЛОЖЬ, ЕСЛИ СПИСОК РЕШЕНИЯ ПУСТ
12 : ВОЗВРАТ РЕШЕНИЯ @ DUP ПЕРВЫЙ SWAP
13   ХВОСТ ПЕРВЫЙ ХВОСТ РЕШЕНИЯ DUP @ ХВОСТ ХВОСТ
14   УСТАНОВИТЬ НАЙТИ-ПРЕДЛОЖЕНИЕ? DUP
15   IF ДОБАВИТЬ-ЦЕЛИ THEN ;
```

Экраны 60, 61. Интерпретатор правил Пролога: ПОЛУЧИТЬ-ЦЕЛЬ, НАЙТИ-ПРЕДЛОЖЕНИЕ, НАЙТИ-ПРЕДЛОЖЕНИЕ? и ВОЗВРАТ

```
ЦЕЛИ:     (КОЗЕЛ)
РЕШЕНИЯ:  НУЛЬ

ЦЕЛИ:     (БОРЬКА)
РЕШЕНИЯ:  (КОЗЕЛ БОРЬКА)

ЦЕЛИ:     (ГРУБИЯН)
РЕШЕНИЯ:  (КОЗЕЛ ГРУБИЯН)

ЦЕЛИ:     (МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-РОГА)
РЕШЕНИЯ:  (КОЗЕЛ МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-РОГА)

ЦЕЛИ:     (ДАЕТ-МОЛОКО ИМЕЕТ-ВОЛОС-ПОКРОВ ИМЕЕТ-РОГА)
РЕШЕНИЯ:  (МЛЕКОПИТАЮЩЕЕ ДАЕТ-МОЛОКО
                ИМЕЕТ-ВОЛОС ПОКРОВ)
          (КОЗЕЛ МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-РОГА)

ЦЕЛИ:     (ИМЕЕТ-ВОЛОС-ПОКРОВ ИМЕЕТ-РОГА)
РЕШЕНИЯ:  (ДАЕТ-МОЛОКО)
          (МЛЕКОПИТАЮЩЕЕ ДАЕТ-МОЛОКО
                ИМЕЕТ-ВОЛОС- ПОКРОВ)
          (КОЗЕЛ МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-РОГА)

ЦЕЛИ:     (ГРУБИЯН ИМЕЕТ-РОГА)
РЕШЕНИЯ:  (ИМЕЕТ-ВОЛОС-ПОКРОВ ГРУБИЯН)
          (ДАЕТ-МОЛОКО)
          (МЛЕКОПИТАЮЩЕЕ ДАЕТ-МОЛОКО
                ИМЕЕТ-ВОЛОС-ПОКРОВ)
          (КОЗЕЛ МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-РОГА)

ЦЕЛИ:     (ИМЕЕТ-РОГА)
РЕШЕНИЯ:  (ИМЕЕТ-ВОЛОС-ПОКРОВ)
          (ДАЕТ-МОЛОКО)
          (МЛЕКОПИТАЮЩЕЕ ДАЕТ-МОЛОКО
                ИМЕЕТ-ВОЛОС-ПОКРОВ)
          (КОЗЕЛ МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-РОГА)

ЦЕЛИ:     НУЛЬ
РЕШЕНИЯ:  (ИМЕЕТ-РОГА)(ИМЕЕТ-ВОЛОС-ПОКРОВ)
          (ДАЕТ-МОЛОКО)
          (МЛЕКОПИТАЮЩЕЕ ДАЕТ-МОЛОКО
                ИМЕЕТ-ВОЛОС-ПОКРОВ)
          (КОЗЕЛ МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-РОГА)

УСПЕШНОЕ ЗАВЕРШЕНИЕ
```

Рис. 9.4. Еще одна трасса выполнения слова ПОИСК с возвратом в первых двух фреймах

ложения, ВОЗВРАТ вызывает слово НАЙТИ-ПРЕДЛОЖЕНИЕ. При успешном завершении поиска новые цели из подобранного предложения посредством вызова ДОБАВИТЬ-ЦЕЛИ добавляются к списку ЦЕЛИ. Если подходящего предложения не оказалось, поиск завершается неудачей. ВОЗВРАТ возвращает в стек флаг, означающий успех или неудачу.

Теперь нам осталось лишь объяснить два второстепенных слова из определения ПОИСК. Слово ДОБАВИТЬ-ЦЕЛИ выбирает цели из списка, на который ссылается первый указатель предложения, и присоединяет его к списку ЦЕЛИ. Достижение целей в списке РЕШЕНИЯ нетривиально - чередованием слов ПЕРВЫЙ и ХВОСТ. Выражение РЕШЕНИЯ @ ХВОСТ осуществляет перемещение за первую цель в списке РЕШЕНИЯ, возвращая в качестве результата оставшуюся часть данного списка. Затем слово ПЕРВЫЙ возвращает список, содержащий предложения, - урезанный посредством слова ХВОСТ список ПРЕДЛОЖЕНИЯ. Следующее вхождение слова ПЕРВЫЙ возвращает первое предложение из этого списка, а последнее слово ХВОСТ перемещается за заголовок. Оставшаяся часть представляет собой список, содержащий цели первого предложения.

Слово ПОЛУЧИТЬ-ЦЕЛЬ восстанавливает первый элемент из списка ЦЕЛИ и удаляет его, превращая этот список в его хвост. Помимо перечисленных слов, нам требуются три списка, которые занимают небольшой объем памяти и показаны на экране 60.

Кроме собственно слова ПОИСК, на экранах с 63-го по 65-й описаны несколько дополнительных слов для пользовательского интерфейса и отладки. В Прологе запрос цели может быть осуществлен следующим образом:

```
?- цель
```

Здесь приводится еще одно похожее слово, которому, правда, нужно задавать список целей. Чтобы доказать цель, нужно набрать:

```
?-(ЦЕЛЬ)
```

Слово ?- для получения списка целей, находящихся в списке ЦЕЛИ, использует слово ЧТСП. Это слово опустошает список РЕШЕНИЯ, подготавливая его для нового поиска. Ответом на запрос будет либо УСПЕХ, либо НЕУДАЧА. Если вы хотите увидеть следы поиска, то можно встроить в ?- слово СЛЕД (как было показано ранее), для чего необходимо выполнить переустановку вектора вычисления ВЫВОД на СЛЕД:

```
' СЛЕД IS ВЫВОД
```

В исходном состоянии вектор ВЫВОД установлен на слово ПОИСК, которое не оставляет следов.

```
63
0 \ ИНТЕРПРЕТАТОР ПРАВИЛ ПРОЛОГА
1
2  ( -> ФЛАГ) \ ФЛАГ = ИСТИНА => УСПЕХ
3 : ПОИСК
4   BEGIN (ПОИСК)
5   UNTIL
6 ;
7
8 DEFER ВЫВОД ' ПОИСК IS ВЫВОД
9
10 : ?- РЕШЕНИЯ НУЛЬ УСТАНОВИТЬ ЦЕЛИ DUP НУЛЬ
11   УСТАНОВИТЬ ЧТСП ВЫВОД CR  ['] ВЫВОД >BODY @ [']
12   ПОИСК = \ УСТАНОВЛЕН ЛИ ВЫВОД НА ПОИСК ?
13   IF
14     IF ." УСПЕХ" ELSE ." НЕУДАЧА" THEN
15   THEN ;
64
0 \ ИНТЕРПРЕТАТОР ПРАВИЛ ПРОЛОГА
1
2 \ МАКСИМУМ ЦЕЛЕЙ В ПРАВИЛЕ = 4
3 : ПРАВИЛО: CREATE HERE DUP 2+ , НУЛЬ , 10 ALLOT ЧТСП ;
4
5 : .КЛ ПРЕДЛОЖЕНИЯ @ ВЫДАТЬ ;
6 : .ЦЕЛИ ЦЕЛИ @ ВЫДАТЬ ;
7
8 : КАК? РЕШЕНИЯ @ DUP НОЛЬ
9   IF ВЫДАТЬ
10  ELSE
11    BEGIN DUP НОЛЬ NOT
12    WHILE DUP ХВОСТ ПЕРВЫЙ ПЕРВЫЙ ВЫДАТЬ ХВОСТ ХВОСТ
13    REPEAT DROP
14  THEN
15 ;
65
0 \ ИНТЕРПРЕТАТОР ПРАВИЛ ПРОЛОГА
1 : ДОСТАТОЧНО? KEY?
2        IF KEY DROP KEY 13 =
3          ELSE FALSE
4        THEN
5 ;
6
7 \ СЛЕДЫ ПОИСКА
8 : СЛЕД
9   BEGIN CR ." ЦЕЛИ:" ЦЕЛИ @ ВЫДАТЬ
10     CR ." РЕШЕНИЯ:" КАК? CR (ПОИСК) DUP >R
11     IF
12       IF ." УСПЕХ" ELSE ." НЕУДАЧА" CR THEN
13     THEN R> ДОСТАТОЧНО? OR
14   UNTIL
15 ;
```

Экраны 63, 64, 65. Интерпретатор правил Пролога: ПОИСК, ?- , ПРАВИЛО:, .КЛ, .ЦЕЛИ, КАК?, ДОСТАТОЧНО? и СЛЕД

Описание слова СЛЕД приведено на экране 65. Оно аналогично слову ПОИСК, но на каждом шаге выдает фреймы списков ЦЕЛИ и РЕШЕНИЯ. Имеется также слово ДОСТАТОЧНО?, которое позволяет управлять выводом следов с клавиатуры. Если во время вывода следов вы нажмете любую клавишу, то трассировка приостановится до следующего нажатия любой клавиши, исключая клавишу <ВК>. Если во время паузы вы нажмете клавишу <ВК>, слово СЛЕД завершит свое выполнение.

С помощью слова КАК? СЛЕД печатает первое предложение из списка РЕШЕНИЯ. Слово .КЛ упрощает вывод списка ПРЕДЛОЖЕНИЯ, а слово .ЦЕЛИ - списка ЦЕЛИ. Слово ПРАВИЛО: упрощает составление правил (см.экраны 70 и 71). Формат его применения таков:

ПРАВИЛО: имя-предложения список

Здесь список считывается внутри слова ПРАВИЛО: посредством ЧТСП. Чтобы минимизировать число имен, имя заголовка предложения должно совпадать с именем предложения (например, МЛЕКОПИТАЮЩЕЕ или ДАЕТ-МОЛОКО на экране 70). В данной реализации правило может содержать до четырех целей, поскольку для него выделено всего 10 байт: два на заголовок и восемь на цели.

ДЕРЕВЬЯ ВЫВОДА

Дерево вывода облегчает процесс слежения за выполнением поиска. На рис.9.5 изображено дерево вывода для следа, показанного на рис.9.3. Это графическая иллюстрация последовательности вывода, где каждый шаг представлен в виде узла, и все узлы пронумерованы в порядке следования фреймов. Цели правила соединены с соответствующими предложениями линиями. Структура дерева отражает общий алгоритм поиска, а его уровни - ход построения цепочек интерпретатором правил.

На верхнем уровне задана цель КОЗЕЛ. Поиск подходящего предложения приводит нас к правилу 2 на следующем уровне. Поскольку данное предложение не подходит, происходит возврат на верхний уровень и выбирается следующее предложение, в результате чего мы выходим на правило 3. Правило 3 определяет цели МЛЕКОПИТАЮЩЕЕ и ИМЕЕТ-РОГА, что ведет к поиску на очередном уровне. Обратите внимание на то, что вид логических отношений между узлами чередуется через уровень. На верхнем уровне, если задано несколько целей, *все* они для успешного завершения общего поиска должны быть удовлетворены. Каждая цель может рассматриваться как подзадача, и для того чтобы общая задача была решена, необходимо решить все подзадачи.

Таким образом, логические отношения между целями на верхнем уровне представляют собой конъюнкцию.

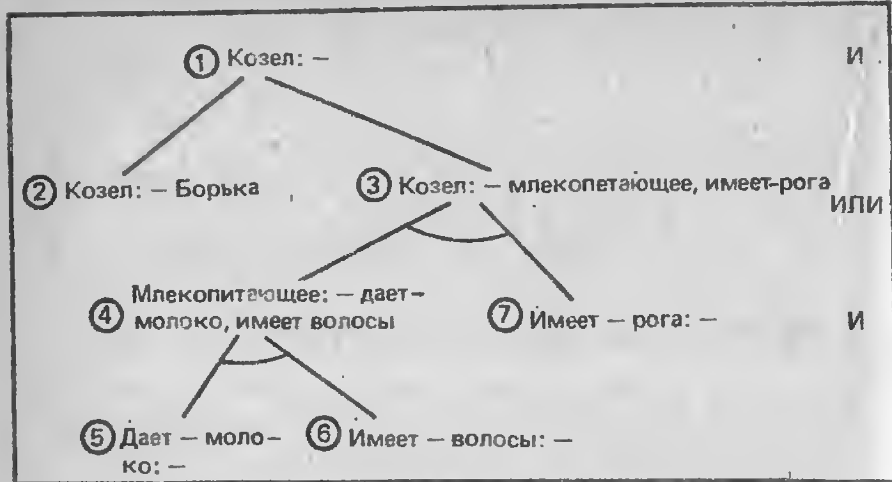


Рис. 9.5. Дерево вывода для цели КОЗЕЛ, показывающее чередование уровней ветвления типа И и ИЛИ. Числа указывают порядок использования предложений

Если же спуститься уровнем ниже, то здесь каждая ветвь отображает альтернативное предложение, и для успешного завершения поиска требуется, чтобы успешно завершилось хотя бы одно из них, т.е. должен благополучно завершиться вывод в узле 2 *или* в узле 3. На этом уровне логические отношения между узлами определяются как дизъюнкция. На следующем уровне обе цели МЛЕКОПИТАЮЩЕЕ и ИМЕЕТ-РОГА должны быть достигнуты - ветви логически конъюнктивны (дуга, соединяющая линии, показывает, что они логически связаны конъюнкцией). Итак, с одной стороны, дерево вывода графически представляет состояние памяти во время поиска, а с другой - показывает путь разбиения задачи достижения цели верхнего уровня на подзадачи.

ПОИСК В ШИРИНУ И ЭВРИСТИЧЕСКИЙ ПОИСК

Как было показано выше, поиск в глубину осуществляется путем обхода дерева вывода сверху вниз до завершения - успешного или неудачного. Только после этого можно подняться вверх по дереву и, если поиск был неудачным,попытаться выбрать следующую ветвь, а затем спуститься по ней вниз (отсюда и название поиска "в глубину"). В Прологе алгоритм поиска реализован по принципу "в глубину", поскольку новые цели добавляются в начало

списка ЦЕЛИ. А что бы произошло, если бы цели добавлялись в конец списка? Тогда, вместо того чтобы продвигаться вниз по уровням, мы сначала перебирали бы все альтернативные предложения. т.е. обходили бы все узлы данного уровня и лишь потом спускались уровнем ниже. Другими словами, мы не прорабатывали бы одно решение на всю глубину, а параллельно пытались бы реализовать альтернативные решения. Конечно, это более длинный путь, но и такой подход имеет свои преимущества. При поиске в глубину можно успеть продвинуться достаточно глубоко перед тем, как вы потерпите неудачу. При поиске в ширину исключается возврат.

На заре создания ИИ поиск представлял собой одну из главных областей исследования. В то время были разработаны алгоритмы обхода дерева на основе эвристических правил, которые зависели от конкретной прикладной области. Например, в шахматной игре для каждой позиции возможно множество вариантов продолжения. Количество ветвей из заданного узла дерева вывода огромно, и для нахождения решения (а это выигрышная позиция) каждую ветвь нужно проработать на достаточную глубину, так как число вероятных ходов велико. Поэтому поиск в глубину в данном случае будет неэффективным. Применяя поиск в ширину, мы можем анализировать все возможные продолжения позиции на ход-два вперед. Но и тогда вряд ли возможно просмотреть все варианты в отведенное время. Требуется некоторый способ выбора потенциально хорошего пути обхода дерева. Чтобы избежать заведомо неудачных ходов, можно применить, например, такое правило: "Предпочтение отдавайте ходам, ведущим к усилению вашей позиции в центре доски". Подобные правила в большей степени приходят с опытом, чем как результат изучения теории шахматной игры. Эти правила называются *эвристическими*. Они позволяют управлять поиском, делая последний более эффективным. С помощью так называемого алгоритма *формирования весов* (*uniform-cost*) можно рассчитать вес каждой ветви и выбрать ветвь, имеющую в настоящий момент времени наименьший вес. Другой важный алгоритм поиска, так называемый А*алгоритм, осуществляет поиск по оптимальному, эвристически выбираемому пути.

УНИФИКАЦИЯ

Основной недостаток реализации поиска в данном варианте Пролога заключается в том, что термами могут быть только атомы. Все они являются переменными Форта, главное назначение которых - хранить имена. Наш вариант Пролога может быть усилен введением переменных особого рода: не обычных переменных Форта, а логических. Тогда мы сможем записывать на Прологе утверждения такого типа:

```
козел(X) :- млекопитающее(X), имеет-рога(X)
```

Здесь X - переменная, которая может заменять имя конкретного козла. Пользуясь подобными переменными, можно средствами Пролога получать атомы, являющиеся подстановками для этих переменных (или *конкретизированные* переменные), что позволяет посредством вывода отвечать на вопросы или восстанавливать данные. К переменным,как вы помните, мы уже обращались при определении слова соед. Чтобы убедиться в том, что с их помощью можно отвечать на вопросы, рассмотрим утверждения:

```
козел(борька)
козел(грубиян)
```

Введя следующее выражение:

```
?- козел(X)
```

мы получим конкретизацию X. Вместо X могут быть подставлены и "борька", и "грубиян". Пример с использованием правила приведен ниже:

```
дед(X,Z) :- отец(X, Y), отец(Y, Z)
отец(карл, сэм)
отец(сэм, льюис)
```

Применяя эти предложения для выражения

```
?- дед(x, льюис)
```

путем конкретизации переменных в правиле мы получим: "X=карл". Подстановка переменных называется *связыванием* (*bindings*). Связывания могут быть представлены списком пар (связок). Для приведенного выше примера возможны такие связки:

```
((X карл)(Y сэм)(Z льюис))
```

Первым элементом каждой пары является переменная, а вторым - ее подстановка. Переменная задает область определения для подстановки. Совокупность связок служит *окружением*, или *средой* (*environment*).

Вы можете наблюдать за выполнением нашего примера интерпретатором Пролога, отмечая среду на каждом шаге. Целевое предложение верхнего уровня - дед(X, льюис), следовательно, Z - область определения для льюис. Две цели из правила помещаются в список целей. Производя поиск на сопоставление с предикатом отец(X,Y), интерпретатор находит второе предложение,

отец(карл, сэм), и осуществляет подстановку со следующим результатом:

```
((Y сэм)(X карл)(Z льюис))
```

Вторая цель, также с предикатом "отец", сопоставляется с тем же самым предложением и осуществляется попытка выполнить следующие подстановки:

```
((Y карл)(Z сэм))
```

Обе эти подстановки противоречат предыдущим. В данной ситуации сопоставление заканчивается неудачей, и поиск подходящего предложения продолжается. В следующем предложении имеются связки для Y и Z, которые полностью согласуются с предыдущими подстановками. Иными словами, мы приходим к согласованию цели верхнего уровня с базой знаний. Теперь подстановки можно вывести.

В приведенном выше примере аргументами предикатов являются атомы. В более общем случае они могут быть также предикатами или переменными. Ниже приводится пример более сложного сопоставления. Два предиката:

```
f(X Y), f(сэм, g(X))
```

имеют связки:

```
((X сэм) (Y g(X)))
```

Эти предикаты могут быть выражены в виде списка:

```
(f X Y) (f сэм (g X))
```

Процесс, который мы кратко рассмотрели, называется *сопоставлением с образцом* (*pattern matching*). Здесь осуществляется попытка *унификации* двух логических выражений путем подстановок согласующихся переменных. Наиболее универсальный *алгоритм унификации* представлен на рис. 9.6.

Покажем на примере сопоставления нескольких выражений, как выполняется алгоритм унификации - УНИФИКАЦИЯ (UNIFY). Допустим, у нас имеются два предиката:

```
(a X X), (a Y b)
```

Их первые элементы сопоставимы. Оба вторых элемента представляют собой переменные со связкой (X Y). Из сопоставления третьих элементов вытекает, что X - область определения для b, что выражается связкой:

```
((X Y)(X b))
```

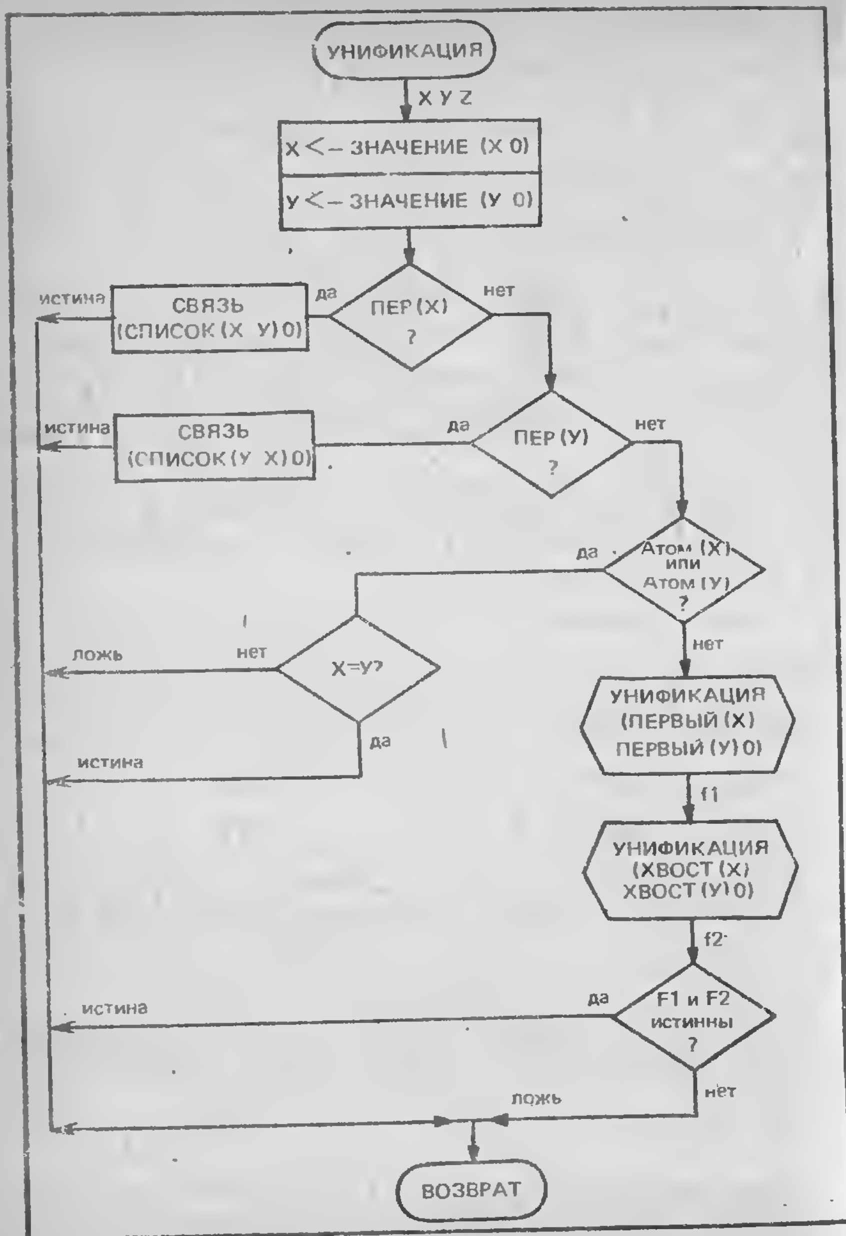


Рис. 9.6. Алгоритм унификации Пролога. Предполагается, что переменные могут входить в предикат только один раз

В контексте данного окружения Y должно бы быть областью определения для b, поскольку X служит такой областью для Y, но это не так. Кроме того, X является областью определения более чем для одной переменной. А чтобы получить окружение вида

(X Y) (Y b))

```
   67
 0 \ АЛГОРИТМ УНИФИКАЦИИ
 1 ( C СП1 -> СП2) \ 2АССОЦ-СПИСОК СОДЕРЖИТ
 2                 \ ПАРЫ ПЕРСПИС-ЗНАЧЕНИЕ
 3 : 2АССОЦ DUP НОЛЬ
 4   IF NIP
 5   ELSE 2DUP ПЕРВЫЙ =
 6     IF NIP
 7     ELSE ХВОСТ ХВОСТ РЕКУРСИЯ
 8     THEN
 9   THEN
10 ;
11
12
13
14
15

   68
 0 \ АЛГОРИТМ УНИФИКАЦИИ
 1
 2 ( C1 @O -> СП2)
 3 : ЗНАЧЕНИЕ OVER ПЕР?
 4   IF 2DUP 2АССОЦ DUP НОЛЬ
 5     IF 2DROP
 6     ELSE ROT DROP ХВОСТ ПЕРВЫЙ SWAP РЕКУРСИЯ
 7     THEN
 8   ELSE DROP
 9   THEN
10 ;
11
12
13
14
15
```

Экраны 67, 68. Алгоритм унификации: определение 2АССОЦ и ЗНАЧЕНИЕ

вместо переменной в сопоставлении должна участвовать ее напарница по связке. Тогда переменная X окажется связанной с Y, и при сравнении третьих элементов сопоставляться с b будет напарница X по связке, в результате чего получится (Y b).

Получение напарницы для переменной осуществляется алгоритмом ЗНАЧЕНИЕ (VALUE), приведенным на рис. 9.7. Слово ЗНАЧЕНИЕ получает аргументы X и O, где X - переменная, а ее

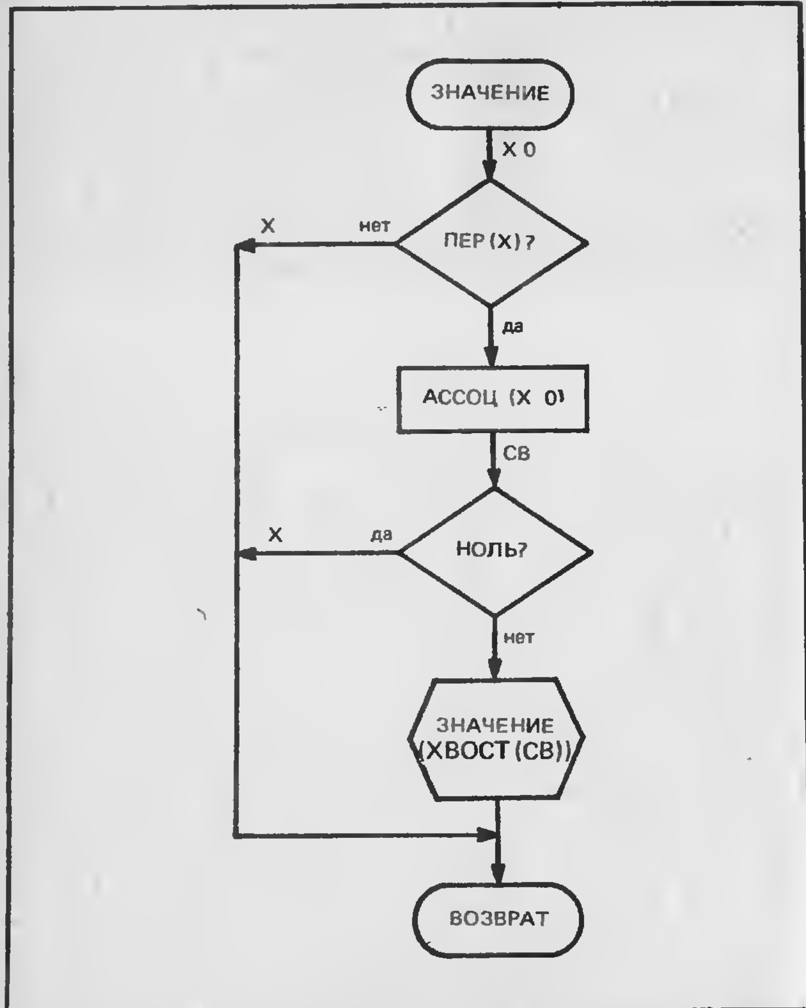


Рис. 9.7. Применение ЗНАЧЕНИЯ словом УНИФИКАЦИЯ для получения связки логической переменной X в окружении O

напарница по связке восстанавливается из списка окружения O. ЗНАЧЕНИЕ оставляет нетронутыми аргументы, не являющиеся переменными. В противном случае с помощью АССОЦ отыскивается X в O. Если поиск неудачен, аргумент X возвращается, поскольку для него нет связки. Если X найден, выполняется рекурсия по хвосту связки, который может быть следующей переменной.

Слово ЗНАЧЕНИЕ просматривает цепочку подстановок переменной до последней. Например, при заданном O вида

((X W)(W U)(U V)(V Y))

ЗНАЧЕНИЕ возвратит Y как связку для X. Поскольку для данной переменной возвращается ее ближайшая связка, то при подстановках достаточно задавать имя только одной переменной. Кроме того, выдерживается условие, согласно которому для переменной, служащей идентификатором области определения по отношению к другой переменной, ЗНАЧЕНИЕ доставляет атом. Так, в окружении

((X Y)(Z a)(Y Z))

связкой для X будет a. Хотя и известно, что слову ЗНАЧЕНИЕ первоначально из УНИФИКАЦИИ передается переменная X, тем не менее необходимо предварительно проверить, является ли новый аргумент X переменной, так как ЗНАЧЕНИЕ рекурсивно вызывает само себя. Связки для исходного аргумента X может не существовать.

Вернемся к алгоритму УНИФИКАЦИЯ. Первый предпринимаемый шаг - замещение X и Y их связками. Затем, если X оказывается переменной, она сцепляется с Y и такая пара добавляется к окружению O, которое и возвращается в виде результата. Если же переменной будет Y, то X и Y меняются местами, поскольку первым элементом пары в связке должна быть переменная. Далее пара помещается в O. Несмотря на то что в реализации списков, описанной в гл.7, не используется механизм ячеек связи, при следующем обращении к лиспоподобному слову СПИСОК(X Y) создается ячейка связи (или точечная пара) с головой X и хвостом Y, а это уже А-список Лиспа (см. гл.7). В нашем примере O реализовано в виде линейного списка.

В том случае, когда ни X, ни Y не являются переменными, осуществляется их проверка на атомы. Если один из них оказывается атомом, то другой должен быть сопоставимым атомом. Иначе не получится новой связки, поскольку нет переменной. Если же выясняется, что X и Y - не атомы и не переменные, значит, они должны быть списками, используемыми для представления предикатов. В такой ситуации УНИФИКАЦИЯ вызывается рекурсивно по первым двум элементам списков X и Y.

```
   69
0 \ АЛГОРИТМ УНИФИКАЦИИ
1  ( C1 C2 O -> F) \ F = ИСТИНА, ЕСЛИ C1 и C2  УНИФИЦИРУЕМЫ;
2                \ O СОДЕРЖИТ СВЯЗКИ
3 : УНИФИКАЦИЯ DUP >R @ ROT OVER ЗНАЧЕНИЕ -ROT
4   ЗНАЧЕНИЕ OVER ПЕР?
5    IF НУЛЬ -ROT R> СПИСОК TRUE
6    ELSE DUP ПЕР?
7      IF SWAP НУЛЬ -ROT R> СПИСОК TRUE
8      ELSE 2DUP ATOM? SWAP ATOM? OR NOT
9        IF 2DUP ПЕРВЫЙ SWAP ПЕРВЫЙ SWAP R@ РЕКУРСИЯ
10         IF ХВОСТ SWAP ХВОСТ SWAP R> РЕКУРСИЯ
11         ELSE 2DROP R> DROP FALSE THEN
12       ELSE R> DROP =
13       THEN
14     THEN
15   THEN ;
```

Экран 69. Алгоритм унификации: определение слова УНИФИКАЦИЯ

Если при таком вызове УНИФИКАЦИИ добавляются новые связки, то в результате получается обновленное окружение O' . Далее уже в новом окружении УНИФИКАЦИЯ вызывается по оставшейся части списков X и Y. Это пример двойной рекурсии (применявшейся нами ранее при определении слова РАВНО). После второго вызова создается окончательное окружение. Слово УНИФИКАЦИЯ, описанное на экране 69, при удачном сопоставлении возвращает истинное значение флага. По завершении двух рекурсивных вызовов флаги логически умножаются (AND), поскольку результирующее сопоставление заканчивается успешно только при условии, что оба вызова оказались удачными. Если нет ни одного ложного флага, можно считать, что сопоставление состоялось. Теперь попытаемся применить алгоритм УНИФИКАЦИЯ на практике. Рассмотрим пример:

```
(a X b)
(a (c Y) Y)
```

Здесь аргумент X будет связан с (c Y), а Y с b. Переменная Y в c(Y) неявно служит и областью определения для b, поскольку это указано непосредственно. Такие связки допустимы. Приведем другой пример:

```
(a X X X)
(a Y Y Y)
```


Здесь X - область определения Y при сопоставлении вторых элементов каждого списка. Что касается третьих элементов, то связка X или Y представляет собой область определения Y. Наконец, при последнем сопоставлении слово ЗНАЧЕНИЕ ищет связку для X. Такой связкой будет Y. Затем ЗНАЧЕНИЕ отыскивает связку для Y, а это Y. ЗНАЧЕНИЕ снова ищет связку для Y. Получается бесконечный циклический процесс - *зацикливание.* Во избежание зацикливания нужно не допускать, чтобы переменные связывались сами с собой. Если переменная входит в некоторый предикат более одного раза, то УНИФИКАЦИЯ не может выполнить процесс унификации правильно. В следующем примере:

```
(X)
((a X))
```

УНИФИКАЦИЯ также не выдаст совместимую связку. X - область определения (a X), но подстановка вместо X даст (a (a X)). При повторе получим:

```
(a (a (a X)))
```

Зацикливание в данном случае произошло потому, что имеется вхождение X в функцию, для которой он сам служит областью определения. В алгоритме УНИФИКАЦИЯ отсутствует *контроль вхождений*, поскольку не отслеживается ситуация, когда X входит в (a X).

Алгоритм УНИФИКАЦИЯ - типичный представитель алгоритмов унификации, применяющихся при реализации Пролога. Быстрота его выполнения достигается ценой следующих ограничений:

1. Переменная должна входить в некоторую функцию не более одного раза.

2. Переменная не должна входить в функцию, по отношению к которой сама является границей. Как правило, эти ограничения не столь существенны, но позволяют достигать большой скорости вычислений.

Существует еще одно осложнение, с которым необходимо считаться при включении алгоритма УНИФИКАЦИЯ в ПОИСК: на различных уровнях дерева вывода для одних и тех же имен переменных связки будут различными. Вы видели это в примере с предикатом дед(X,Y). Отслеживать различные связки одной и той же переменной можно двумя способами - *копированием структур* и *совместным использованием структур.* Первый способ состоит в копировании граничных выражений на каждом уровне вывода. второй - в том, что каждая переменная в связке снабжается индексом, отражающим уровень. При таком подходе связки имеют вид ((X i) a), где i - индекс X, отражающий номер использования X. В литературе по логическому программированию приводится

множество методов, повышающих эффективность реализаций Пролога. Один из них (параллельные вычисления) излагается в гл. 10.

```
   73
0 \ ВЫВОД РАСШИРЕННОГО ПЕРСП
1 ( @ -> ФЛАГ) \ ФЛАГ=ИСТИНЕ, ЕСЛИ @ ЯВЛЯЕТСЯ
2            \ PFA ПЕРЕМЕННОЙ
3 : ATOM? BODY> @ ['] НУЛЬ @ = ;
4
5 ( @СПИСОК -> )
6 : ВЫДАТЬСП CR ." ("
7   BEGIN DUP ПЕРВЫЙ DUP ATOM?
8     IF DUP НОЛЬ
9       IF DROP ELSE BODY> >NAME .ID THEN
10    ELSE DUP ПЕР?
11      IF 2- BODY> >NAME .ID ELSE РЕКУРСИЯ THEN
12    THEN ХВОСТ DUP НОЛЬ
13  UNTIL 8 ( ЗАБОЙ) EMIT ." ) " DROP
14 ;
15
```

```
   74
0 \ ВЫВОД РАСШИРЕННОГО ПЕРСП
1
2  ( @СПИСОК -> )
3 : ВЫДАТЬ DUP @ НОЛЬ
4   IF DROP CR ." НУЛЬ"
5   ELSE DUP ATOM?
6     IF BODY> >NAME .ID
7     ELSE DUP ПЕР?
8       IF 2- BODY> >NAME CR ." ПЕРСП " .ID
9       ELSE ВЫДАТЬСП
10       THEN
11    THEN
12  THEN
13 ;
14
15
```

Экраны 73, 74. Вывод на печать расширенного слова ПЕРСП

УПРАЖНЕНИЯ

1. Измените порядок предложений в списке ПРЕДЛОЖЕНИЯ (экран 71) так, чтобы предотвратить:
   а) возврат для цели КОЗЕЛ;
   б) возврат вообще.

2. Разработайте алгоритм сохранения недоказанных целей в списке НЕУДАЧИ с тем, чтобы и просматривать этот список перед поиском цели в списке ЦЕЛИ. Если искомая цель будет обнаружена в списке НЕУДАЧИ, то должен быть прекращен поиск и активирован возврат.

```
   66
0 \ АЛГОРИТМ УНИФИКАЦИИ
1
2
3
4 \ ПЕРСП ОПРЕДЕЛЯЕТ ЛОГИЧЕСКИЕ ПЕРЕМЕННЫЕ
5  ( -> PFA+2)
6 : ПЕРСП CREATE HERE 2+ , 0 , DOES>
7    2+ ;
8
9
10 ПЕРСП FOO ' FOO @ FORGET FOO
11
12
13   ( C -> F) \ F ИСТИНА, ЕСЛИ ПЕРСП
14  ПЕР? DUP @ 0= SWAP 4 - @ [ DUP ] LITERAL = AND ;
15 DROP
```

Экран 66. Алгоритм унификации: определение ПЕРСП и ПЕР?

3. Перепишите слово ДОБАВИТЬ-ЦЕЛИ таким образом, чтобы новые цели присоединялись к концу списка ЦЕЛИ, а не к его началу. (Вам может понадобиться новый список для переупорядочения целей.)

4. Используя модифицированный вариант ДОБАВИТЬ-ЦЕЛИ из упр.3, напишите слово ШПОИСК для выполнения поиска в ширину. Сравните его эффективность с эффективностью слова ПОИСК, приведенного на экране 71.

5. Только для модели APPLE II: объясните слова, применяемые при создании логических переменных на экране 66. Число, предшествующее C в (ПЕРСП), - код

машинной команды перехода. Почему логическими переменными не могут служить переменные Форта ?

6. Объясните слово 2АССОЦ на экране 67 и структуры данных, с которыми оно выполняется. Объясните порядок использования этого слова в определении слова ЗНАЧЕНИЕ на экране 68?

7. Модифицируйте слово УНИФИКАЦИЯ таким образом, чтобы имелась возможность применения в одном предикате нескольких вхождений одной и той же переменной.

8. Осуществите контроль вхождений. Создайте слово Форта, которое выбирало бы два выражения в качестве аргументов, X и Y, и проверяло бы, не входит ли X в Y.

9. Присоедините контроль вхождений из упр.8 к слову УНИФИКАЦИЯ из упр.7.

10 Модифицируйте алгоритм УНИФИКАЦИЯ из упр.7 так, чтобы его можно было встроить в слово НАЙТИ-УТВЕРЖДЕНИЕ.

Глава 10

ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ

Обработка списков, поиск и сопоставление по образцу являются важными понятиями для систем, основанных на знаниях, но это лишь базовые методы, применяющиеся в данной области. В настоящей главе мы кратко рассмотрим и другие идеи, возникавшие в процессе исследований по ИИ. Более подробно они описаны в литературе, ссылки на которую приведены в приложениях.

ВСТРОЕННЫЕ ПРЕДИКАТЫ ПРОЛОГА

В гл.9 обсуждалась простая реализация Пролога в виде алгоритма поиска в глубину. Алгоритм содержал единственную процедуру. Программы представляли собой набор целевых предложений в базе знаний. Программирование на Прологе было скорее описательным, чем предписательным. Вместо того чтобы *предписывать* выполнение тех или иных действий над данными, мы *описывали* сами данные. Поскольку Пролог реализует некоторый вариант логики предикатов, а ПОИСК используется для доказательства теорем, то одной только этой процедуры достаточно для решения задач из области искусственного интеллекта. Этот язык может применяться при построении экспертных систем, где список предложений представляет собой экспертные знания.

Пролог тем не менее обладает такими дополнительными средствами, как встроенные предикаты. Один из них, *отсечение* (символ !), может быть вставлен в тело некоторого правила между целями, чтобы форсировать возврат. Например, для правила:

```
C :- A, !, B
```

если вывод В заканчивается неудачей, то и С форсированно заканчивается неудачей. Отсечение вынуждает слово ПОИСК удалить альтернативные ветви С на уровне ИЛИ и вернуться на предыдущий уровень И. Предикат отсечения не является описательным, так как он влияет на функционирование слова ПОИСК. Приведем еще несколько предписательных предикатов Пролога:

* включить(X) (assert(X)) - включение утверждения X в виде предложения в список ПРЕДЛОЖЕНИЯ и успешное завершение;

* удалить(X) (retract(X)) - удаление X из списка ПРЕДЛОЖЕНИЯ и успешное завершение. Если X не совпал ни с одним утверждением в списке ПРЕДЛОЖЕНИЯ, - это означает неудачу;

* вызвать(X) (call(X)) - вызов X в качестве цели и успешное завершение поиска, если цель доказывается. В Прологе имеются и другие встроенные предикаты, которые здесь не рассматриваются.

ПРОЦЕДУРНОЕ ДОПОЛНЕНИЕ И ВЫЗОВ ПО ОБРАЗЦУ

Некоторые интерпретаторы правил используют правила, которые представляют собой пары условие-действие. Если условия удовлетворяются, то происходит не занесение заключения в базу знаний, а выполнение данного заключения как процедуры. Такая схема более удобна, чем чисто описательная, поскольку одна из процедур может использоваться для занесения заключения. Этот способ известен под названием *процедурное дополнение.* Он сочетает в себе элементы предписательных методов, например арифметических вычислений, и описательных.

Вместо (почти) чисто описательного программирования на Прологе можно создавать интерпретаторы правил-предписаний. Развивая идею процедурного дополнения, мы можем выражать правила в форме образца, сцепленного с процедурой, выполняемой в том случае, если выполнено сопоставление с конкретным образцом. Эта процедура аналогична слову Форта с той лишь разницей, что слово Форта вызывается по имени, а процедура - в результате сопоставления с образцом. Алгоритм сопоставления с образцом соответствует алгоритму УНИФИКАЦИЯ Пролога, в то время как выполняемые процедуры являются частью функции ПОИСК. С образцами могут работать процедуры: ДОБАВИТЬ(ADD), УДАЛИТЬ (REMOVE) или ВОССТАНОВИТЬ (RETRIEVE), которые добавляют к базе данных, содержащей доказанные факты, новые факты, удаляют факты из нее или восстанавливают их так же, как и функция ?- в Прологе. В системах, основанных на знаниях подобного рода, имеется база данных для хранения доказанных фактов.

В Прологе функция "включить(X)" добавляет X к списку ПРЕДЛОЖЕНИЯ.

Чтобы продемонстрировать использование правил в виде процедур, приведем пример правила с обратным рассуждением:

```
дед(X,Z)
   ВОССТАНОВИТЬ    отец(X,Y)
   ВОССТАНОВИТЬ    отец(Y,Z)
   КОНЕЦ
```

где отец(X,Z) - образец, а слово КОНЕЦ означает завершение процедуры. Правило прямого вывода проиллюстрируем на следующем примере:

```
отец(X,Y),отец(Y,Z)
   ДОБАВИТЬ  дед(X,Z)
   КОНЕЦ
```

Алгоритм сопоставления с образцом связывает переменные из процедуры с переменными образца. В случае правила обратного вывода X в первой команде ВОССТАНОВИТЬ служит областью определения для X образца. При втором вызове команды ВОССТАНОВИТЬ это связывание остается. Такие связывания являются локальными по отношению к выполняемой процедуре поиска образца.

НЕМОНОТОННЫЕ РАССУЖДЕНИЯ

Следующим за логикой предикатов шагом в преодолении перечисленных выше ограничений явилось создание *немонотонной логики.* Как для систем с прямым, так и с обратным выводом, факты добавляются в базу данных по мере их доказательства. Если цель не может быть достигнута, то считается, что она ложна. Такой подход называется *допущением о замкнутости мира.* Это равносильно предположению о том, что база знаний полна. Были разработаны и другие виды доказательств, основанные на возможности существования неполных знаний. В частности, если некоторое предположение как цель не была доказана, отсюда вовсе не вытекает, что оно ложно. Может быть, наша база знаний недостаточно полна для получения истинного результата. Более того, в систему, основанную на знаниях, можно вводить противоречивые данные. В базе данных такой системы имеется способ оценки обоснованности фактов, который позволяет для достижения их совместимости изменять базу данных.

В рассмотренных выше системах, основанных на знаниях, интерпретатор правил применяет правила к фактам, получая новые факты. Последние помещаются в базу данных, которая пополняет-

ся по мере поступления новых доказательств. Поскольку новые факты истинны и конкретны, они никогда не удаляются из этой базы данных. Прямое и обратное рассуждения представляют собой монотонный вид доказательств, так как число фактов со временем монотонно увеличивается.

Команда УДАЛИТЬ (из предыдущей главы) может быть использована при простом немонотонном доказательстве, называемом *доказательством по умолчанию*. При этом к традиционным правилам вывода, исполняемым словом ПОИСК, добавляются правила вывода, применяемые по умолчанию. Они используются наряду с традиционными правилами с той лишь разницей, что их заключения проверяются на непротиворечивость с базой данных и добавляются только в том случае, если противоречия не обнаружены.

Сами эти правила могут быть противоречивыми, и требуется только упорядоченность их по приоритетам применения для обеспечения совместимости в базе данных. Обычно, чем конкретнее правило, тем выше его приоритет. Например, правило определения вида топлива для автомобилей выглядит следующим образом:

```
топливо(X, бензин) :- автомобиль(X)
```

Но для паровых машин Стэнли это подразумеваемое правило приведет к ложному заключению. Для них требуется более конкретное правило:

```
топливо(X, пар) :- Стэнли(X)
```

которое должно применяться до общего правила, принимаемого по умолчанию. В интерпретаторах, управляющих выбором правил сопоставления, такое упорядочение может быть встроено в механизм управления или в процедурные правила систем с активацией, управляемой по образцу.

Более сложный вид немонотонных рассуждений связан с удалением утверждений из базы данных и называется *поддержкой истинности*. В системе поддержки истинности (СПИ) (truth maintenance system - TMS) доказанные факты вместе с их обоснованием называются *записями зависимости* (*dependency records*) и хранятся в базе данных. В традиционных системах считается, что если факт попал в базу данных, он достоверен. В СПИ элементы базы данных уместнее называть *предположениями* (*beliefs*), поскольку они зависят от предположений, не всегда верных. Возможны гипотетические допущения, которые в случае неудачного доказательства удаляются. Если какое-либо утверждение оказалось некорректным, то все основанные на нем предположения должны быть также удалены из базы данных. Области обоснования предположений образуют сеть, где обоснованиями некоторого предположения служат другие предположения и правила. В конечном итоге эта сеть

обусловленностей должна состоять из высокодостоверных обоснований предположений, которые можно назвать фактами.

Преимущество метода СПИ над логикой предикатов заключается в возможности использования обоснований предположений при ведении базы данных. Когда база данных обновляется путем добавления или исключения предположений, активируется система СПИ, которая проверяет, справедливы ли еще обоснования для содержащихся в базе предположений. При получении отрицательного результата они удаляются. Тем самым упрощается сопровождение базы данных, которая в противном случае разрасталась бы до неимоверных размеров и содержала бы несовместимые факты. Задача корректного обновления базы данных называется *проблемой границ* (*frame problem*).

Вы уже видели, каким образом в Прологе предложения, используемые словом ПОИСК для доказательства цели, накапливаются в списке РЕШЕНИЯ. Такие предложения являются обоснованием цели и должны заноситься в базу данных вместе с целями. Если позднее некоторое предложение окажется ложным, то доказательства, на основании которых получено это предложение, будут удалены из базы данных. Например, цель КОЗЕЛ (см. рис. 9.4) обосновывается утверждениями, содержащимися в последнем фрейме списка РЕШЕНИЯ. Если любое из них, например факт ИМЕЕТ-РОГА, признается ложным, то цель КОЗЕЛ, не имеющая более обоснований, должна быть удалена из базы данных.

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

В базе знаний Пролога правила не организованы в какие-либо группы по родственным признакам, разве что упорядочены по смыслу программистом. Базы знаний, не имеющие внутренней классификации предложений, называются *плоскими* (*flat*) системами. Все правила в них расположены на одном иерархическом уровне. Такой способ организации, хотя и упрощает базу знаний, также не является эффективным, поскольку при сопоставлении каждой цели должен просматриваться весь список предложений целиком. Если бы предложения были сгруппированы по своим контекстам, то интерпретатор правил мог бы по индексу выбирать соответствующую группу и пытаться применять правила, содержащиеся в ней. Системы, организация которых основана на разбиении знаний по группам, называются *объектно-ориентированными* (*object-based*).

Смоллток - объектно-ориентированный язык, используемый для создания систем, основанных на знаниях. Он состоит из *объектов*, с помощью которых можно представлять объекты в той или иной области знаний. В Смоллтоке программы и объекты посылают друг другу *сообщения*. Несмотря на то что Смоллток снабжен опи-

сательными средствами, это язык предписаний, поскольку в его основе лежит механизм обмена сообщениями. Сообщения вынуждают объекты-приемники выполнять отдельные процедуры и возвращать сообщения-результаты. Такие процедуры называются *методами (methods)*.

В отличие от других рассмотренных выше языков каждый объект Смоллтока содержит и метод его выполнения. Здесь вместо процедур, которым передаются данные, достаточно иметь объекты, поскольку они сами владеют нужными им процедурами. Это позволяет избежать проблем, возникающих при вызове процедуры, обрабатывающей, скажем, числа с фиксированной точкой, когда ей передаются числа с плавающей точкой. Данные объекта совместимы со своими методами. Если объект получает сообщение на выполнение сложения, то он выбирает требуемый собственный метод выполнения такого задания, чего не случается, если задание передается не той процедуре.

Так как объекты зачастую используют одни и те же методы, они группируются по *классам*. Объекты, принадлежащие одному и тому же классу, имея свои собственные, уникальные, методы, могут наследовать и общие методы, принадлежащие всему классу. Иными словами, объекты могут иметь свойства, уникальные для конкретного объекта, но могут обладать и общими свойствами, характерными для всех объектов, принадлежащих данному классу. По своим возможностям объектно-ориентированные языки хорошо подходят для построения семантических сетей и фреймовых систем.

Простая семантическая сеть изображена на рис. 10.1. Ее узлы представляют объекты или их свойства. Линии, соединяющие узлы, отражают связи между ними. На рисунке показаны два вида связи: ЭТО (или ЯВЛЯТЬСЯ) и СВОЙСТВО. Первый вид связи ЭТО (IS-A) наиболее часто употребляется в семантических сетях и определяет класс, которому принадлежит конкретный объект. КОЗЕЛ принадлежит классу МЛЕКОПИТАЮЩЕЕ, БОРЬКА принадлежит классу КОЗЕЛ и является представителем этого класса. Вид связи СВОЙСТВО приписывает объектам свойства. У объекта КОЗЕЛ одно свойство - ИМЕЕТ-РОГА. Но будучи включенным в класс МЛЕКОПИТАЮЩЕЕ, он наследует также и свойства данного класса: ДАЕТ-МОЛОКО и ИМЕЕТ-ВОЛОС-ПОКРОВ. Несмотря на то что БОРЬКА сам не обладает этими свойствами, но, являясь КОЗЛОМ и МЛЕКОПИТАЮЩИМ, наследует их от указанных классов.

Основная форма организации таких сетей - *классификационная иерархия (taxonomic hierarchy)*, где между объектами существуют только связи типа ЯВЛЯТЬСЯ. Возможности выражения взаимоотношений между объектами в семантических сетях не ограничиваются лишь принципом иерархии. Линии на рисунке могут обо-

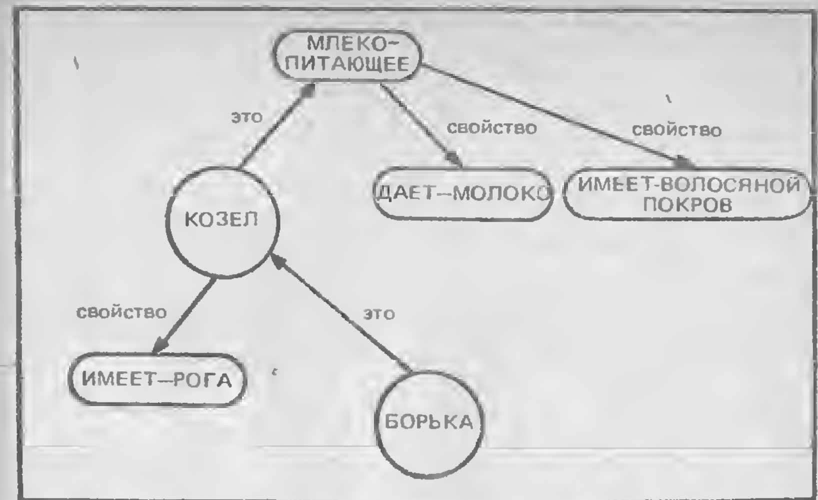


Рис. 10.1. Семантическая сеть. БОРЬКА - это КОЗЕЛ, и он наследует свойства как понятия КОЗЕЛ, так и понятия МЛЕКОПИТАЮЩЕЕ

значать и другие виды взаимоотношений между объектами. Более сложная семантическая сеть включает уже свойства со значениями. Вместо некоторого универсального свойства здесь могут быть изображены конкретные свойства, например для связывания слов РОГА и КОЗЕЛ можно использовать свойство "является-частью".

Семантические сети считаются более сложной формой представления знаний, чем логика предикатов. Однако и в этих сетях логика неявно присутствует. Чтобы не получить в результате семантической бессмыслицы, взаимосвязям должны быть присвоены конкретные значения. Даже на первый взгляд очевидно, что простому отношению ЭТО может соответствовать несколько значений. На рис. 10.1 таких значений два. Линия, связывающая объект БОРЬКА с объектом КОЗЕЛ, в терминах логики, обозначает предикатное утверждение о том, что БОРЬКА есть представитель класса КОЗЕЛ:

```
козел(борька).
```

Такая интерпретация отношения ЭТО выражает принадлежность определенному классу. Другой вариант связи типа ЭТО, показанный на рис. 10.1, может быть выражен логическим языком Пролога следующим образом:

```
млекопитающее(X) :- козел(X)
```

что означает: для всех X, если X является козлом, то X также является и млекопитающим. Второе значение ЭТО показывает, что объект "козел" относится к подмножеству млекопитающих.

Наследование свойств логически выражается не так просто, как операции Пролога, выполняемые над переменными. В Прологе все переменные имеют *квантор всеобщности*, так что истинность некоторого предположения распространяется на все подстановки переменных. Если все объекты класса наследуют некоторое свойство, характерное для данного класса, например КОЗЕЛ из класса МЛЕКОПИТАЮЩЕЕ, то для всех млекопитающих свойства объектов класса МЛЕКОПИТАЮЩЕЕ считаются истинными. Из рис. 10.1 ясно, что свойство ДАЕТ-МОЛОКО логически связано с классом МЛЕКОПИТАЮЩЕЕ следующим образом:

```
дает-молоко(X) :- млекопитающее(X).
```

Объединив это правилом с предыдущим, получим: дает-молоко(X):-козел(X).

До последнего пункта семантическая сеть эквивалентна своему логическому представлению в виде двух утверждений Пролога. Отличия от логики предикатов начинают проявляться, когда некоторый объект получает право отказаться от свойств путем введения локального отношения. Если у козла БОРЬКИ не было рогов, а это свойство наследовано им от объектов класса КОЗЕЛ, то он должен отказаться от наследованного свойства ИМЕЕТ-РОГА. В данном случае X в утверждениях Пролога применимо не для всех X, поскольку БОРЬКА составляет исключение.

Описанная выше возможность локального отказа от наследования свойств по связи ЭТО представляет собой стандартное средство в семантических сетях и привносит в них элементы нестандартной логики. Значением наследованного свойства теперь служит ожидаемое свойство типичного представителя конкретного класса. Исключения из данного класса получают свои уникальные свойства через непосредственное отношение свойства. При таком использовании наследованные свойства, не отмененные локальными исключениями, действуют как подразумеваемые и являются механизмом реализации логики умолчания (см. раздел "Немонотонные рассуждения", рассмотренный выше).

Семантические сети не следует воспринимать как один из аспектов (усовершенствованный) инженерии знаний. Лишь в последнее время стало проясняться логическое следствие приписывания такому фундаментальному типу связи, как ЯВЛЯТЬСЯ (ЭТО), различных значений путем "вычленения" его возможных значений посредством отдельных связей. Несмотря на свою неформальную природу, семантические сети могут найти широкое применение, особенно в области понимания естественного языка.

МЕТАРАССУЖДЕНИЯ: УПРАВЛЕНИЕ ВЫВОДОМ

Выше мы рассмотрели вызов по образцу как способ более гибкого управления интерпретатором. Кроме того, в гл.9 ("Поиск в ширину и эвристический поиск") обсуждался эвристический поиск в качестве средства для повышения эффективности функционирования интерпретатора. Если правильный путь на низлежащие уровни выбирается с привлечением эвристических знаний, то поиск пути к достижению цели будет короче. Интересным вариантом воплощения данного подхода является снабжение интерпретатора его собственными знаниями о методах выбора пути при поиске, причем эти управляющие знания могут быть представлены в той же форме, что и знания предметной области (в форме предложений). Таким образом, интерпретатор при выборе следующего предложения из списка ПРЕДЛОЖЕНИЯ может пользоваться своими знаниями по управлению. Правила, которые содержат управляющие знания, называются *метаправилами*.

Процедура поиска интерпретатора с привлечением метаправил несколько отличается от алгоритма ПОИСК, изложенного в гл. 9. Сначала для заданной цели находятся все сопоставимые с нею предложения. Такой набор предложений называется *конфликтным набором (conflict set)*. "Конфликт" заключается в необходимости выбора правила из нескольких возможных. Поскольку все они альтернативны и расположены на уровне ИЛИ дерева вывода, поиск будет вестись вниз по каждому из них. Метаправила применяются при выборе одного из альтернативных продолжений.

Отдельные метаправила основаны на некотором знании характера самой решаемой задачи. Например, следующее правило:

> *Из конфликтного набора должно быть выбрано утверждение с наименьшим числом целей*

эвристически утверждает, что в первую очередь должен осуществляться поиск по наикратчайшему пути.

Смысл применения метаправил заключается в том, чтобы разбить область поиска задачи на иерархические уровни абстракции, а затем найти путь решения для наивысшего уровня. Каждый шаг решения сопровождается спуском на более низкий уровень. Этот процесс продолжается до тех пор, пока не будут найдены решения на самом нижнем уровне. Например, при строительстве дома сначала создается общий проект, на основе которого могут быть разработаны более детальные проекты. Область данной задачи можно разбить на следующие три уровня:

Уровень здания: Размещение, внешние размеры, выбор расположения комнат.
Уровень комнаты: Выбор техслужб и отделки.

Уровень техслужб:	Размещение систем (водоснабжения, электроосвещения, обогрева) с учетом расположения стен.

Каждый уровень абстракции должен представляться базой знаний своего уровня. В нашей задаче на уровне здания детали нижних уровней можно не учитывать. После создания удовлетворительного плана на уровне здания можно проектировать каждую комнату. Выработав решение на этом уровне, можно приступать к проработке элементов самих комнат.

В такой области, как проектирование зданий, уровни абстракции выделяются вполне естественно, поскольку они мало зависят друг от друга. Отделка, например, ванной комнаты не влияет на проектирование интерьера кухни. Однако элементы общего для многих комнат назначения, скажем системы водоснабжения, электроосвещения и обогрева, образуют уже взаимосвязанные подзадачи уровня комнаты. Если мы хотим, чтобы эти системы были спроектированы оптимально, то должны учитывать расположение всех комнат. Данный аспект задачи является глобальным, и мы не можем выделять здесь отдельные подзадачи, решаемые локально для каждой комнаты. На уровне же здания эти ограничения на длину электропровода и трубопровода не имеют значения, поэтому решение (неоптимальное) находится в предположении, что подзадачи на уровне комнаты независимы.

При независимых подзадачах задача после декомпозиции имеет вид дерева, а при взаимосвязанных - форму сети. Для решения задач второго типа разработан метод, называемый *распространением ограничений (constraint propagation)*. Ограничения выражают связь между подзадачами путем уменьшения числа возможных решений отдельной подзадачи. В примере с проектированием здания подзадача размещения комнат на уровне здания порождает ограничения на размещение водопровода внутри комнат.

Ограничения используются по *принципу наименьших свершений (least commitment principle)*, т.е.при решении глобальных проблем на подзадачи накладываются минимальные ограничения с тем, чтобы облегчить нахождение оптимального решения на локальном уровне.

Похожий метод, *называемый управляемым возвратом (dependency-directed backtracking)*, аналогичен методу поддержания истинности. Но вместо того, чтобы каждый раз обновлять базу данных для внесения в нее записей обусловленностей или их удаления, при данном методе эти записи служат основой для анализа тупиковых правил и фактов с тем, чтобы учитывать их во время выбора альтернативного пути при возврате. Информация, собранная при прохождении тупикового пути, впоследствии может быть использована для предотвращения повторения одних и тех же ошибок.

НЕОПРЕДЕЛЕННОСТЬ И ДОСТОВЕРНОСТЬ

Управляемый возврат помогает проводить доказательства с неполными данными. Сами же доказательства выполняются по точным правилам вывода. Отказ от строгих рассуждений сделал возможным использование вероятностей для описания достоверности правил. Достоверность правил выражается с помощью чисел от 1 до 100, где большее число означает большую достоверность. В одной общеупотребительной системе эти числа называются *факторами достоверности (certainty factors)*. Поскольку теория вероятности хорошо разработана, естественным было бы считать факторы достоверности некоторой мерой вероятности. Но это не так. Хорошо известная формула теории вероятности и на первый взгляд наиболее употребимая - *правило Байеса*. Однако оно здесь не годится, так как для определения вероятности какого-либо заключения вероятности фактов должны быть независимыми. Создать же базу знаний, в которой обоснованность всех правил проводилась бы независимо от других, почти невозможно.

Поскольку методы стандартной статистики применять крайне сложно, в медицинской экспертной системе MYCIN были использованы факторы достоверности. Каждому правилу ставился в соответствие фактор достоверности (ФД). Вычисление ФД доказанного факта выполнялось по формуле

$$X + Y - XY$$

где X - ФД факта, а Y - ФД правила, причем значения X и Y находятся в диапазоне от 0 до 1. Технология M.1 основана на той же формуле определения ФД, но в диапазоне от 0 до 100, например:

$$X + Y - XY/100$$

Факторы достоверности могут передаваться по цепочке доказательств. Чем больше применяется правил, тем выше значение ФД при приближении к достоверности. Такое увеличение ФД происходит в результате накопления достоверности для финального заключения и не зависит от порядка применения правил. В основе формулы определения факторов достоверности не лежит какая-либо теория обоснованности. Она используется лишь постольку, поскольку отдельные свойства отражают степень их обоснованности экспертами. Из теории вероятности следует, что частично определенные гипотезы являются также и частично не определенными. Но частичная обоснованность гипотезы не должна служить частичной аргументацией против нее.

Правила, содержащие факторы достоверности, эквивалентны обоснованию гипотез (их заключений). Каждое обоснование ведет к сужению круга возможных гипотез, поскольку любой фрагмент обоснования поддерживает только определенные гипотезы. Для

всех этих фрагментов отдельные гипотезы, обосновываемые ими, образуют перекрывающиеся подмножества. Существует *теория достоверности Демпстера-Шейфера* (D-S теория), которая представляет собой математическую теорию, включающую как специальные случаи и функции Байеса, и ФД, с той лишь разницей, что если степень достоверности некоторой гипотезы принимается за X (в диапазоне от 0 до 1), то на другие возможные поднаборы данной гипотезы остается значение (1 - X). В D-S теории X называется *базовым распределением вероятностей* (БРВ) - *basic probability assignment* (BPA). Это понятие отличается от вероятности Байеса, где (1 - X) представляет вероятность подгруппы, содержащей все остальные гипотезы. Согласно же D-S - теории (1 - X) считается распределением по объединению всех остальных подгрупп.

БРВ пересечения двух подгрупп равно произведению их индивидуальных БРВ. Пересечения могут иметь несколько подгрупп. Произведения их БРВ складываются. Так как одно правило с одним заключением дает обоснование для группы с одним элементом - заключением данного правила, то комбинация из двух правил приводит к пересечению четырех групп, выражающемуся в нашем случае комбинацией БРВ. Пусть БРВ двух правил обозначаются через X и Y, а группа-заключение будет представлена группой, содержащей заключение этих правил. Возможны пересечения следующих объектов:

1. Групп, содержащих заключения двух правил, в каждой из которых имеется по одному и тому же элементу. БРВ = XY.
2. Группы-заключения правила один и остальных возможных подгрупп, содержащих X: БРВ = X(1-Y).
3. Группы-заключения правила два и остальных возможных подгрупп, содержащих Y: БРВ = Y(1-X).
4. Остальных возможных подгрупп (1-X)(1-Y).

Четвертый случай представляет пересечение подгрупп, не содержащих группы-заключения. Только первые три пересечения дают в результате заключение-группу, и, значит, объединенное БРВ есть их сумма:

$$XY + X(1 - Y) + Y(1 - X)$$

или после преобразования:

$$X + Y - XY$$

Мы получили формулу, в точности совпадающую с формулой определения фактора достоверности. Она является результатом комбинирования БРВ правил, которые либо оба подтверждают некоторую гипотезу (группа-заключение), либо оба ее опровергают.

Последняя рассматриваемая форма представления неполных знаний известна как *нечеткая логика (fuzzy logic)*, которая также не совсем согласуется с теорией вероятности. В нечеткой логике нечто, состоящее в некоторой группе, получает в качестве веса число в диапазоне от нуля до единицы. Таким образом можно выразить количественно неопределенные понятия, например "много". Так, дальность поездки можно представить следующими цифрами:

Принадлежность к группе	группа (диапазон чисел)
0.1	от 0 до 2км
0.3	от 2 до 20км
0.5	от 20 до 50км
0.9	свыше 50 км.

Степень принадлежности к группам, полученная в результате выполнения операций объединения или пересечения, выражается формулами:

$$m(X \cup Y) = max(m(X), m(Y))$$
$$m(X \cap Y) = min(m(X), m(Y))$$

где m(X) и m(Y) - степени принадлежности к группам X и Y соответственно.

КОНТЕКСТНЫЕ СЛОВАРИ ФОРТА

В Форте есть еще одно дополнительное средство, не описанное в гл.6, - *контекстные словари (vocabularies)*. Они могут использоваться для модуляризации знаний примерно так же, как это принято в рамках объектно-ориентированного программирования. Основной словарь Форта (dictionary) содержит связанный список слов. Новое определяющее слово VOCABULARY формирует новый список:

VOCABULARY *имя*

Тем самым создается контекстный словарь с именем, следующим за словом VOCABULARY. Например:

```
VOCABULARY EDITOR
```

создает словарь текстового редактора. После вызова EDIT список слов из этого словаря становится доступным внешнему интерпретатору, осуществляющему чтение из входного потока. В Форте 83 корневым словарем является FORTH, состоящий из стандартных слов Форта. Если мы введем слово WORDS, то на экран будут выведены слова из доступного, текущего словаря.

Такие слова-словари, как FORTH и EDITOR, содержат указатель на nfa последнего слова в обозначаемых ими словарях. Значение указателя хранится в pfa словарей. На рис. 10.2 изображена общая схема словаря. Обратите внимание на то, что слово FORTH расположено в общем словаре и имеет в своем pfa указатель на

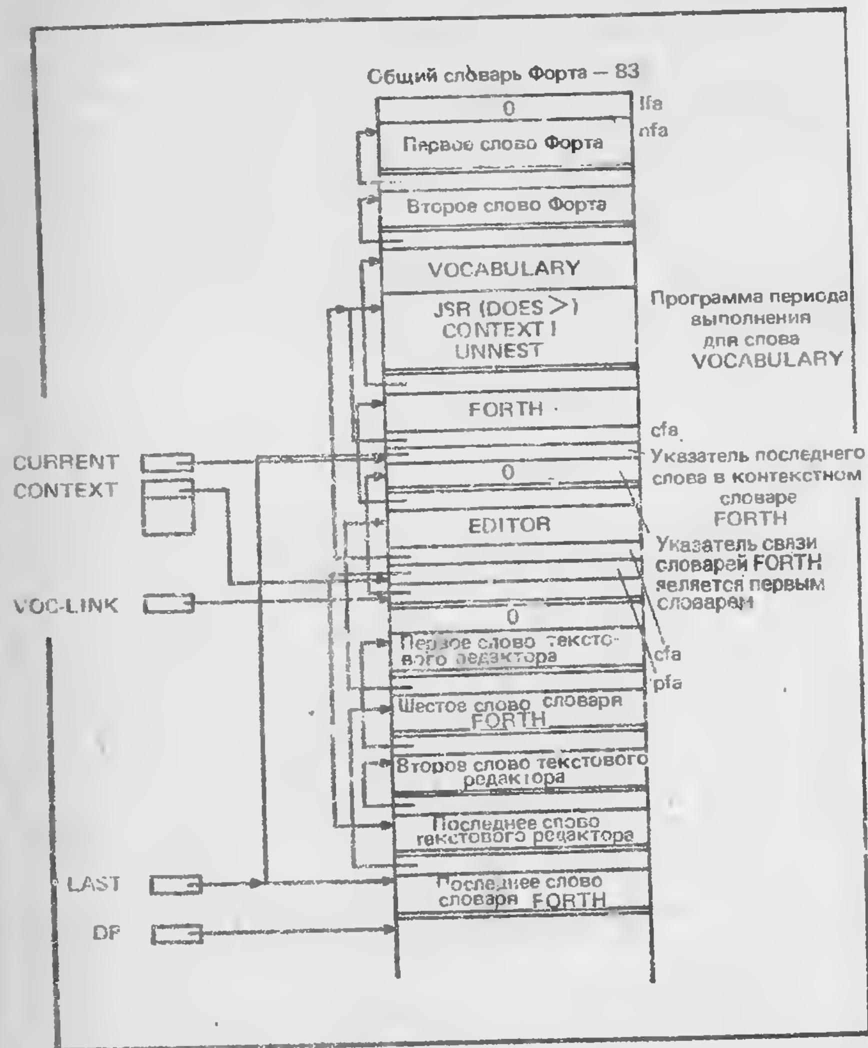


Рис. 10.2. Структура контекстного словаря FORTH

последнее слово из словаря FORTH. lfa этого последнего слова указывает на предпоследнее слово (шестое) из данного контекстного словаря FORTH и т.д. вплоть до первого слова словаря FORTH,

в lfa которого содержится нуль, означающий конец данного списка слов. Аналогично слово-словарь EDITOR содержит указатель на последнее определенное слово в словаре текстового редактора. FORTH находится в своем собственном контекстном словаре. Оба эти слова-словаря входят в словарь FORTH по определению.

В системной переменной CONTEXT содержится указатель на pfa того словаря, в котором будет осуществляться поиск словом DEFINED (используется внешним интерпретатором для поиска слов, прочитанных из входного потока). Следующая системная переменная, CURRENT, тоже указывает на контекстный словарь – на тот, куда должны вноситься определения новых слов.

Списки, составляющие различные контекстные словари, в основном словаре переплетены, так что при описании некоторого слова его определение может попасть в любой словарь, но только в тот, на который указывает CURRENT. Компилятор также применяет системную переменную LAST, указывающую на последнее определенное слово.

Так как контекстные словари определяются посредством слова VOCABULARY, cfa каждого из них содержат указатель на процедуру периода выполнения VOCABULARY. Поскольку слово (DOES>) оставляет на стеке pfa контекстного словаря, программа периода выполнения помещает его в переменную CONTEXT, делая тем самым активированный словарь очередным просматриваемым словарем. Словарь, отмеченный как CURRENT, можно отметить как CONTEXT с помощью слова DEFINITIONS. Это стандартное слово Форта, и его определение выглядит следующим образом:

```
DEFINITIONS CONTEXT @ CURRENT ! ;
```

Таким образом, выражение FORTH DEFINITIONS EDITOR отмечает FORTH как словарь, в который должны помещаться вновь вводимые определения, а EDITOR – как словарь, где будет осуществляться поиск слов, требуемых для определения новых слов. Побочный эффект слова : заключается в том, что оно устанавливает содержимое CONTEXT в CURRENT, поэтому при употреблении не совсем обычных операторов типа : внутри определения их необходимо заключать в квадратные скобки. Контекстные словари связаны воедино указателями, расположенными в теле каждого контекстного словаря по адресу pfa+2. На "поле связи" словаря последнего определенного словаря указывает системная переменная VOC-LINK. В поле связи первого словаря обычно находится нуль, означающий завершение списка. Таким словарем является FORTH.

В экспериментальном расширении Форта-83 CONTEXT превращен в стек указателей на словари, которые просматриваются в порядке размещения в стеке их указателей. pfa слова CONTEXT является вершиной стека словарей. Слово ALSO применяет

операцию DUP с тем, чтобы при активации какого-либо словаря предыдущий указатель также находился в стеке. Во время поиска очередного слова в словаре просмотр осуществляется по словарю, указатель которого хранится в вершине стека словарей. Если в данном словаре слова не оказалось, то поиск продолжается во втором словаре из стека.

На дне каждого стека словарей имеется словарь ONLY, применяемый по умолчанию. Кроме имен других словарей, он содержит небольшую дополнительную информацию и представляет собой словарь специального назначения. При активации этот словарь помещается в стек словарей, предварительно очистив последний. Таким образом, чтобы установить следующий порядок поиска: EDITOR, FORTH, а затем по умолчанию ONLY, необходимо ввести:

```
ONLY FORTH ALSO EDITOR
```

Тогда ONLY очистит стек и оставит в нем только словарь ONLY. Поскольку в ONLY хранится лишь FORTH, он будет найден и активирован. FORTH поместит сам себя в вершину стека, заместив ONLY (Вспомните, что ONLY всегда есть на дне стека), после чего ALSO сделает дубль FORTH. Наконец, EDITOR заместит верхний экземпляр FORTH. Теперь просмотр должен осуществляться в порядке расположения словарей: EDITOR, FORTH и затем подразумеваемый словарь ONLY.

С использованием контекстных словарей не только убыстряется поиск в общем словаре Форта, но и появляется возможность давать словам из разных словарей одинаковые имена. Какое конкретно слово будет найдено, зависит от того, который из контекстных словарей, содержащий одинаковые имена, будет просматриваться первым. Контекстные словари можно до некоторой степени представлять как объекты со своими процедурами (аналогичными методам Смоллтока) и локальными переменными. Недостаток предлагаемой схемы заключается в отсутствии механизма наследования свойств. Однако путем соответствующего упорядочения можно располагать слова в других словарях глубже по стеку контекстов, и они будут обнаруживаться там, а не в верхнем словаре. Для организации наследования свойств программисту достаточно задавать порядок просмотра словарей.

Для автоматического определения принадлежности к классу и наследования свойств можно ввести специальные слова, устанавливающие порядок просмотра. Если заданы два словаря, МЛЕКОПИТАЮЩЕЕ и КОЗЕЛ, то связь ЭТО между ними

устанавливается, как показано на рис. 10.1, словом:

```
: КОЗЕЛ! ONLY МЛЕКОПИТАЮЩЕЕ ALSO КОЗЕЛ ;
```


Слово КОЗЕЛ! задает такой порядок просмотра, при котором КОЗЕЛ принадлежит к классу МЛЕКОПИТАЮЩЕЕ, так как этот словарь просматривается вслед за словарем КОЗЕЛ в поисках наследуемых подразумеваемых свойств. Сами свойства могут быть представлены в виде переменных Форта или других типов данных. Чтобы БОРЬКА принадлежал к классу КОЗЛЫ, нужно определить:

```
: БОРЬКА! КОЗЕЛ! БОРЬКА ;
```

Теперь БОРЬКА наследует свойства класса КОЗЛЫ, который в свою очередь наследует свойства класса МЛЕКОПИТАЮЩЕЕ.

Словарная организация открывает широкие возможности для сторонников Форта. С помощью словарей можно разрабатывать и другие средства, способствующие реализации систем построения баз знаний.

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ

Описанный выше интерпретатор правил просматривает дерево вывода последовательно, по одному узлу. С удешевлением микропроцессоров становится возможным параллельное решение задачи, т.е. нужно так разбить задачу на отдельные подзадачи, чтобы каждая из них решалась своим собственным интерпретатором. Мы предлагаем вам три способа параллельных вычислений применительно к Прологу:

1. Выполнение унификации параллельными процессорами.
2. Параллельный просмотр альтернативных узлов уровня ИЛИ.
3. Параллельный просмотр узлов, соединенных по принципу И.

Унификация представляет собой наиболее времяемкую процедуру Пролога, и, применив здесь параллелизм, вы получили бы большой выигрыш в скорости вычислений. К сожалению, это в принципе последовательный процесс. Конечно, было бы неплохо, если бы аргументы сравниваемых предикатов могли сопоставляться параллельно. Как показано в гл.9, для выполнения совместимых подстановок переменные должны сопоставляться со своими предшествующими связками. Если процессор П1 осуществляет связывание переменной X с переменной Y до того, как процессор П2 сможет связать X с a, то П2 должен иметь связку от П1 с тем, чтобы он мог связать с a Y, а не X. Однако тогда весь процесс протекает последовательно, что исключает возможность параллельных вычислений.

Существует подход, называемый *псевдоунификацией (mock unification)*. Это унификация без сцепления. Поскольку сцепление последовательно по своей природе, в данном случае из унификации заимствовано сопоставление символов. Преимущество псевдоунификации состоит в том, что аргументы предикатов могут сопоставляться параллельно. В результате выбирается набор сопоставимых предложений без учета сцепления переменных, а затем выбранные предложения просматриваются последовательно.

Память, где может осуществляться параллельный просмотр с целью сопоставления символов, называется *ассоциативной*. Такая память имеется, но ее возможности ограничены. Были разработаны и ассоциативная память, и псевдоунификатор, но выигрыш в скорости оказался меньше одного порядка.

Различные подходы к параллельному решению диктуют и свои порядки распараллеливания знаний между процессорами. Одну и ту же цель можно поставить перед всеми процессорами одновременно с тем, чтобы каждый из них просмотрел свой список утверждений на предмет сопоставления (с унификацией). Результатом будут успешные варианты сопоставлений. Такой подход приводит к эффективному просмотру дерева вывода в ширину на уровне ИЛИ, т.е. ИЛИ-параллелизм.

В альтернативном варианте каждому процессору задается весь список утверждений и соответствующая цель из списков целей. Так как цели связаны конъюнкцией, все они должны быть доказаны. Такой И-параллелизм более сложен, чем ИЛИ-параллелизм, поскольку цели не могут доказываться независимо в силу наличия совместно используемых термов в предикатах. Однако как только в цели появляется частичная связка, она может служить началом доказательства других целей.

Помимо Пролога как еще одна успешная альтернатива традиционной архитектуре (Фон-Неймана) выступает *потоковая архитектура*, или *архитектура потоков данных (data-flow architecture)*. Существенным ограничением фон-неймановской архитектуры (ее *узким местом*) представляется скорость обмена между центральным процессором (ЦП) и памятью. В потоковых машинах концептуальной основой архитектуры является функциональное программирование.Многочисленные ЦП получают *кванты программы (tokens)* - единицы данных, содержащие имя операции и аргументы. Они передаются в виде пакетов информации коммуникационной сети с идентифицирующими их "конвертами". Если некоторый ЦП может выполнить операцию по обработке такого кванта, он ее выполняет, передавая результат в том же конверте с тем, чтобы ЦП, выдавший квант данных, смог бы опознать результат и им воспользоваться. Потоковая архитектура - одна из параллельных структур вычислений, находящихся в поле зрения исследователей и выпускаемых промышленно.

УПРАЖНЕНИЯ

1. Напишите следующие функции из реализации Пролога, описанной в гл.9:

   а) ВКЛЮЧИТЬ (X)
   б) ВЫЗВАТЬ (X)

2. В Прологе реализован вывод с обратным рассуждением. Объясните, каким образом можно реализовать прямое рассуждение без внесения изменений в интерпретатор Пролога.

3. Разработайте алгоритмы расширения интерпретатора Пролога, рассмотренного в гл.9, с тем, чтобы включить в него следующие средства:

   а) вывод по умолчанию;
   б) метадоказательства;
   в) факторы достоверности;
   г) нечеткую логику;
   д) множество баз знаний на различных уровнях абстракции.

4. Выделите в приведенных ниже системах структурные и функциональные уровни абстракции:

   а)автомобиль;
   б)цифровой компьютер;
   в)почтовая система;
   г)система армейских строевых приемов;
   д)разум/рассудок.

Объясните, почему вы выбрали именно такое разбиение и как это согласуется с теорией для данных систем (если такая существует).

Глава 11

ОБУЧЕНИЕ И РАСПОЗНАВАНИЕ ОБРАЗОВ

Главным в повышении интеллектуального уровня экспертных систем является их способность к обучению. Обучение представляет собой важную отрасль ИИ, первые исследования в которой относятся к началу 50-х годов нашего столетия. В 80-е годы интерес к данной проблеме резко повысился.

Один из видов обучения - формирование плана, например плана движения роботов при выполнении поставленной перед ним задачи. Планирование связано с обучением. Планировщику, который функционирует и в то же время формирует свои планы (т.е. работает в масштабе *реального времени*), зачастую требуется сбор данных об окружающей среде. Полученные данные могут быть им использованы для обновления его же модели этой среды.

Однако не все данные об окружающей среде имеют символьное представление. Поскольку с экспертными системами больше связаны *символьные* преобразования, а не *численные*, нет необходимости рассматривать здесь численные методы. Тем не менее робот или инструментальная экспертная система могут получать при обработке числовых данных нечисловые результаты. Эта область исследований получила название *распознавание образов* (РО). Первоначально РО входило составной частью в ИИ, но позднее выделилось в самостоятельную дисциплину. Скорее всего построение интеллектуальных систем невозможно без объединения принципов ИИ и РО. В настоящей главе вашему вниманию предлагаются основные концепции двух направлений исследований в области ИИ - обучения и распознавания образов.

ОБУЧЕНИЕ

В работе А. Л. Самуэля "Исследования в области машинного обучения на примере игры в шашки" (1959 г.) описывается программа игры в шашки, способная обучаться на основе опыта. Дерево обхода шашечных позиций содержит огромное число вариантов, что требует больших временных затрат на поиск оптимального хода. Для такого поиска (узла на следующем уровне просмотра) необходимы эвристические правила. А.Л.Самуэль оценивает "полезность" хода с помощью полинома. Члены полинома представляют различные аспекты ситуации, скажем, позиционное преимущество или инициативу. Обучение программы состоит в пересчете коэффициентов полинома через несколько ходов после их оценки.

В шашечной программе использованы две формы обучения. Первая форма - *обучение заучиванием* (*rote learning*). При такой форме в памяти запоминаются шашечные позиции и результат игры с тем, чтобы применить их впоследствии. Поскольку исход хранимых позиций известен, для них просчет вариантов не производится. Сэкономленное при этом время можно использовать для анализа иных продолжений игры. Как недостаток обучения заучиванием следует отметить отсутствие обобщения опыта и быстрое заполнение памяти отдельными позициями. Однако в некоторых случаях, в частности, когда нужно проанализировать ситуацию на много ходов вперед, такой подход имеет свои преимущества.

При второй форме обучения обобщение опыта происходит путем изменения полиномиальных коэффициентов. Программа с такой формой обучения никогда не учится разыгрывать стандартные позиции и имеет слабый дебютный репертуар. Ее сильная сторона - эндшпиль, в котором главную роль играет перебор. Обучение заучиванием имеет смысл в дебюте и эндшпиле, где позиции ограничены и хорошо запоминаются.

Большинство форм обучения основано на обобщении опыта. Наиболее известно *концептуальное обучение*, в процессе которого на основе примеров, иллюстрирующих некоторую концепцию, вырабатывается общая концепция. Так как существует множество вариантов обобщения примеров, требуется введение дополнительных ограничений. Прямым ограничением является решение выбирать наименее универсальное описание, совместимое с примерами.

Индукция - это рассуждение от частного к общему. Концептуальное обучение (обучение на примерах) представляет собой форму индукции. Приведем несколько областей, где применимо концептуальное обучение:

* построение конечного автомата по строкам символов;
* генерация математических функций по парам ввод-вывод;
* разработка общих процедур по особенностям поведения.

Что касается последней области, то здесь подвижный робот обучается выполнять процедуры, реализуя отдельные конкретные задания. Впоследствии он будет распознавать итерационные циклы и отождествлять параллельные пути выполнения процедуры, которые по своему результату эквивалентны.

Следующий подход к обучению был исследован Дж. Карбонеллом из университета Карнеги-Меллона - обучение по аналогии. При таком подходе запоминаются решения задач (в соответствии со списками РЕШЕНИЯ программы ПОИСК, описанными в гл. 9). Каждая новая задача начинает решаться обычным способом. Как только ход решения в зависимости от заданной цели становится ясным, он сравнивается с решениями, хранимыми в памяти.

Если обнаруживаются аналогичные решения, то решатель задач ставится о них в известность. Исследуется ход решения уже решенных задач (хранимых в памяти) в поисках наиболее удовлетворяющего данной задаче. Алгоритм поиска по готовым решениям может быть таким же, как и для решаемой задачи. Отыскания решения путем просмотра решенных задач - хорошее средство для нахождения оптимального варианта решения текущей задачи.

Существует несколько методов применения известных решений к текущей задаче. Задачи с известным решением сравниваются с данной задачей, чтобы выяснить их аналогию. Если аналогия обнаруживается, то можно применять уже готовое решение со всеми вытекающими отсюда преимуществами.

Другой метод обучения состоит в задании обучающейся программе функциональных описаний объектов, для которых по примерам таких объектов программа учится строить общие структурные описания. Этот метод применяется для распознавания изображений в системе технического зрения ACRONYM, разработанной в Станфордском университете. Для создания "моста" между функциональными и структурным описаниями в систему должны быть, кроме того, введены сведения о физических ограничениях и физическом поведении объектов вообще.

Сравнительно недавно предложен еще один метод обучения, так называемый метод *стратегического обучения* (*strategy learning*), основанный на эвристическом подходе к просмотру области состояний задачи. В любом состоянии для получения следующего состояния может быть применен один из множества операторов. Для выбора наиболее оптимального пути решения, конечным состоянием которого является состояние цели, используется эвристический подход.

Применительно к Прологу это стратегия выбора одного из сопоставимых правил. Обычно выбирается первое встретившееся сопоставимое правило. Однако, как показано в гл.9, первое выбранное правило не всегда является самым лучшим, поскольку к правильному решению могут вести и другие пути. Эвристический поиск осуществляется по специальным эвристическим правилам, согласно которым выбор требуемого оператора осуществляется на основе информации о состоянии. Существует способ обучения, где техника выбора по эвристическим правилам совершенствуется за счет повышения степени достоверности оператора при его удачном применении и понижения степени достоверности при неудачном. *Оценка достоверности* (*credit assignment*) при таком способе производится после того, как задача будет полностью решена.

Примером реализации на Прологе может служить система ELM, созданная Браздилем. В ней на каждом шаге оператор решаемой задачи сравнивается с оператором решенной ранее. Если между ними обнаруживается различие, ELM переупорядочивает свои операторы таким образом, чтобы правильный оператор оказался первым. В том случае, когда обучение происходит на примере нескольких решенных задач, возможны конфликты. В такой ситуации в операторы, функционирование которых зависит от различия между предикатами двух задач, включаются условия. Если при использовании оператора некоторый предикат истинен в одной задаче, но ложен в другой, то создаются два оператора ограничения.

Поскольку операторами в Прологе считаются правила, на базе конфликтного правила формируются два новых правила, которые помещаются перед ним и просматриваются в первую очередь. В каждом из новых правил имеются различающие предикаты, добавленные к ним как цели и поэтому ограничивающие исходный оператор. Повышение оценки достоверности в ELM достигается упорядочением и ограничением использования списка операторов. При таком подходе к обучению мы начинаем с применения слабых эвристических правил, более универсальных, а получаем упорядоченные и ограниченные эвристические правила, что должно способствовать повышению эффективности нахождения решения новых задач в данной области.

РАСПОЗНАВАНИЕ ОБРАЗОВ

Инженерия знаний - это область символьного программирования, хотя в практических экспертных системах используются и численные преобразования. Например, при выводе с применением факторов достоверности требуется вычисление значений ФД. Сами методы вывода, как и рассмотренные выше методы обучения, строятся на чисто символьных преобразованиях. Однако программа игры в шашки Самуэля в качестве функции оценки задействует полином. Принятие решений в эвристической программе Самуэля ориентировано на вычисления. В основе распознавания образов (не смешивайте с сопоставлением по образцу при унификации или с вызовом, управляемым образцом) лежат численные методы классификации *образов*. Образы представляют собой совокупность свойств, извлекаемых из входных данных. Функционирование систем распознавания образов (РО) разбивается на два этапа:

1. *Извлечение свойств* из входных (числовых) данных и получение на их основе *образов*, включаемых в совокупность образов, или совокупность свойств.

2. *Классификация образов*, т.е. отнесение их к одному, двум или более *классам* классификационной совокупности.

Наибольшей трудностью при разработке систем РО является выбор свойств, подлежащих извлечению из данных и приведение их к форме, делающей классификацию простой (или вообще осуществимой).

Любой образ может быть представлен как упорядоченный набор чисел, в котором каждое число соответствует значению некоторого свойства. Это значение может не совпадать с самим значением извлеченного свойства, а быть результатом масштабирования,

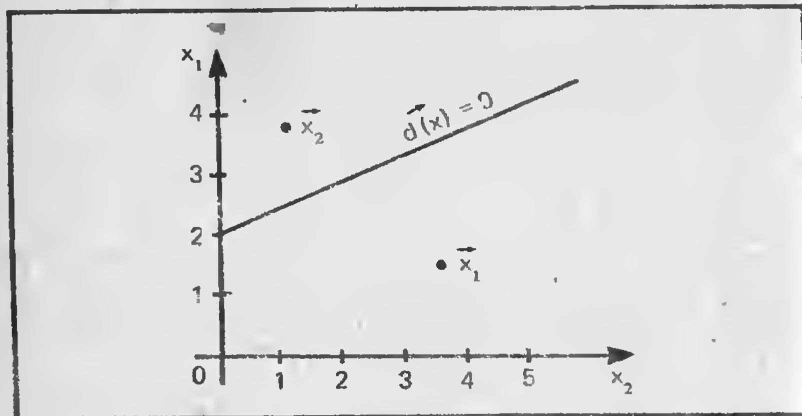


Рис. 11.1. Решающая функция d(x)=0 разделяет области двумерного пространства, занимаемые векторами образов x_1 и x_2

нормализации или обработки иного вида. Математически образ эквивалентен вектору и выражается точкой в гиперпространстве. При n свойствах векторы образов считаются n-мерными и занимают n-пространство, или гиперпространство. В нашем примере векторы имеют n + 1 значений, последнее из которых является дополнительным. В общей форме образ X выражается так:

$$X = [X_1 \; X_2 \ldots X_n \; 1]^T$$

Здесь верхний индекс T означает, что данный вектор транспонирован, т.е. в действительности является вектором-столбцом, или вертикальным вектором. Смысл аргумента 1 будет объяснен позднее. Область образов может быть описана вектором из m векторов образов - матрицей X.

Область образов можно представить в виде гиперпространства с рассеянными внутри него точками, отображающими образы. В том случае, когда выбор свойств удачен, образы, принадлежащие одному и тому же классу, группируются в так называемые кластеры отдельно от точек, принадлежащих другим классам. Если нет пересекающихся кластеров, то с помощью гиперплоскостей можно разделить гиперпространство таким образом, что области, определяемые этими гиперплоскостями, будут представлять различные классы. Например, в двумерном пространстве (с двумя свойствами) гиперплоскость изображается прямой. На рис. 11.1 показана обычная кластеризация в подобном пространстве. Прямая выделяет два кластера так, что образы выше ее могут быть отнесены к классу 1, а ниже - к классу 2.

Математические уравнения плоскостей разбиения на кластеры называются *решающими функциями* (*decision functions*). Прямые или гиперплоскости в общем случае выражаются линейными решающими функциями и могут правильно классифицировать только линейно разделимые образы. Если образы, принадлежащие различным классам, пересекаются, то требуется либо улучшить отбор свойств и повысить качество обработки, либо использовать более сложную решающую функцию. При пересечении кластеров подход, основанный на решающих функциях, не применим к детерминистским функциям. Здесь для определения вероятности попадания образа в определенный класс необходимы статистические функции. Большинство практических задач решаются с помощью статистических средств распознавания образов.

Проиллюстрируем на примере двумерного пространства получение решающей функции. Пример легко экстраполируется на случай с n-мерным пространством. Уравнение прямой на плоскости выглядит следующим образом:

$$y = mx + b$$

Здесь m определяет угол наклона, b - пересечение с осью y. Данное уравнение можно переписать в форме:

$$x_1 - mx_2 - b = 0$$

где $x_1 = y$, а $x_2 = x$. Преобразуем уравнение еще раз:

$$w_1x_1 + w_2x_2 + w_3 = 0$$

где $w_1 = 1$, $w_2 = -m$, а $w_3 = -b$. Расширив это уравнение прямой на n-мерное пространство, получим:

$$w_1x_1 + w_2x_2 + \ldots + w_nx_n + w_{n+1} = 0$$

Запишем полученное уравнение в более компактной векторной форме:

$$W * X = 0$$

где

$$W = [w_1\ w_2\ \dots\ w_{n+1}]$$
$$X = [x_1\ x_2\ \dots\ x_n 1]^T$$

Решающая функция d(X) является уравнением прямой в пространстве с образами. Рассмотрим пример с двумя образами:

$$X_1 = [1\ 4]^T$$
$$X_2 = [3\ 1]^T$$
$$d(X_1) = [1\ -0.5\ -2][1\ 4\ 1]^T = -3\ .$$

Решающая функция, которая их разделяет, имеет вид:

$$d(X) = x_1 - 0.5x_2 - 2 = 0$$

или в векторной форме:

$$d(X) = [1\ -0.5\ -2][x_1\ x_2\ 1]^T$$

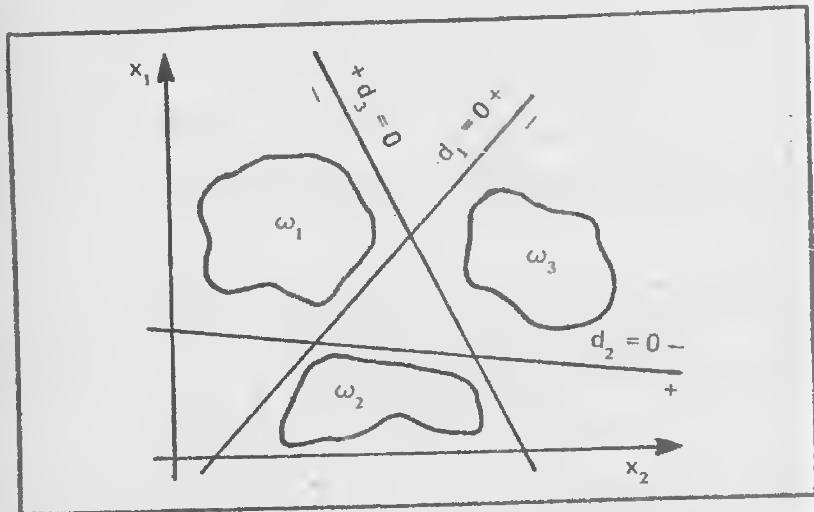


Рис. 11.2. Классификатор типа 1 отделяет заданный класс от всех остальных

Подставив x_2 в решающую функцию, получим:

$$d(X_2) = [1\ -0.5\ -2][3\ 1\ 1]^T = 0.5$$

Для d(X) > 0 точка X находится над решающей функцией, а для d(X) < 0 – под ней. Следовательно, в случае двух классов

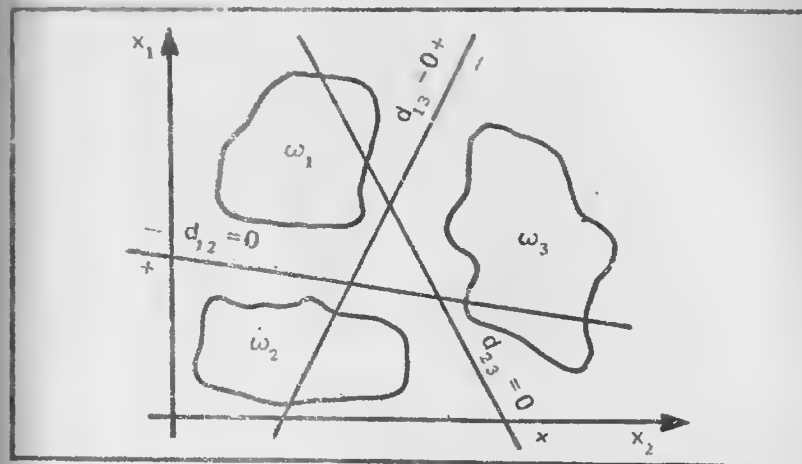


Рис. 11.3. Классификатор типа 2 разбивает классы на пары

класс, которому принадлежит образ, определяется знаком d(X). Для d(X) = 0 точка лежит на прямой и может быть произвольно отнесена к любому из классов, а возможно, и к некоторому третьему классу.

В примере с двумя классами требуется только одна решающая функция. При нескольких классах одной решающей функции недостаточно. Здесь могут быть три типа классификаторов:

1. Класс 1 считается первым классом, а все остальные классы в совокупности – классом 2. Находится решающая функция, разделяющая эти два класса. При n классах требуется n решающих функций. Для данного образа, принадлежащего классу 1, $d_1(X) > 0$, а для других значение решающей функции отрицательно. Классификатор такого типа показан на рис. 11.2.

2. Множество классов разбивается на пары. Находится решающая функция для разделения каждой пары. Решающее значение будет положительным для образа X в $d_{ij}(X)$, если X принадлежит классу i (где j не равно i). Классификатор этого типа изображен на рис. 11.3.

3. Классификаторы первых двух типов комбинируются в целях устранения неопределенных областей (которые не могут быть отнесены ни к одному классу). Если X принадлежит классу i, то $d_i > d_j$ для всех j, не равных i (рис. 11.4).

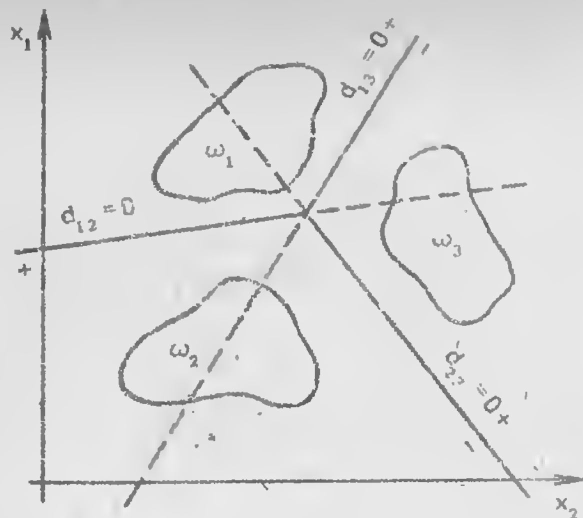


Рис. 11.4. Классификатор типа 3 объединяет классификаторы типов 1 и 2 и не оставляет неопределенных областей

Наибольшее число неопределенных областей получается при пользовании классификатором 1: треугольник в центре и клиновидные участки, образуемые лучами исходящими из углов этого треугольника. Классификатор типа 2 имеет только неопределенный треугольник в центре, а классификатор типа 3 не оставляет неопределенных областей.

Свойства гиперплоскости

В трех схемах классификации используются решающие функции, создание которых входит как составная часть в разработку классификаторов. В общем случае решающие функции представляются n-мерными гиперплоскостями в форме W * X =0, где W - *вектор весов* (*weight vector*). Его компоненты w_1, w_2, ..., w_{n+1} служат коэффициентами уравнения функции решения. Уравнение гиперплоскости выглядит следующим образом:

$$W_0 * X + w_{n+1} = 0$$

где

$$W_0 = [w_1\ w_2\ \dots\ w_n]^T$$

Вектор W_0, как показано на рис. 11.5, является ортогональным, или *нормальным* (перпендикулярным в двумерном пространстве) по отношению к гиперплоскости. Ориентация гиперплоскости мо-

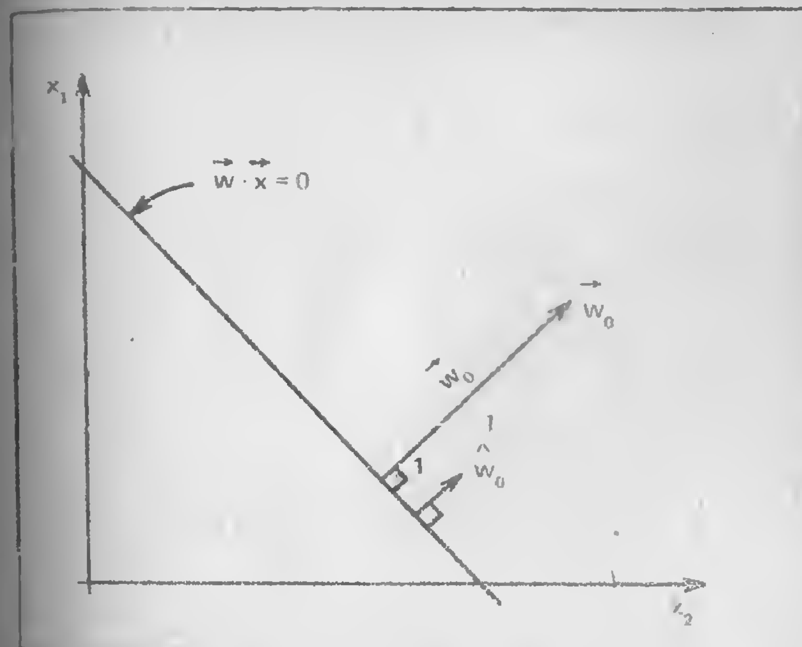


Рис. 11.5. Прямая W * X = 0 имеет вектор ориентации W_0 с нормой $|W_0|$ и единичным вектором ориентации W'_0

жет определяться единичным вектором W'_0, который имеет *норму* (или длину), равную единице, в направлении W_0. Единичный вектор определяется по формуле

$$W'_0 = W_0 / |W_0|$$

а норма вектора, $|W0|$, - по формуле

$$|W_0| = (w_1^2 + w_2^2 + \dots + w_n^2)^{1/2}$$

Можно показать, что W'_0 нормален по отношению к гиперплоскости W * X = 0 на примере двух измерений (рис. 11.5). Пусть X и P - точки на прямой W * X = 0. Эти векторы могут быть изображены направленными отрезками (стрелками) к точкам прямой, которые также представляют и сами векторы. Разность векторов X - P

также лежит на прямой, как показано на рис. 11.6. Выполнив подстановку в уравнение прямой, получим:

$$W * (X - P) = 0$$

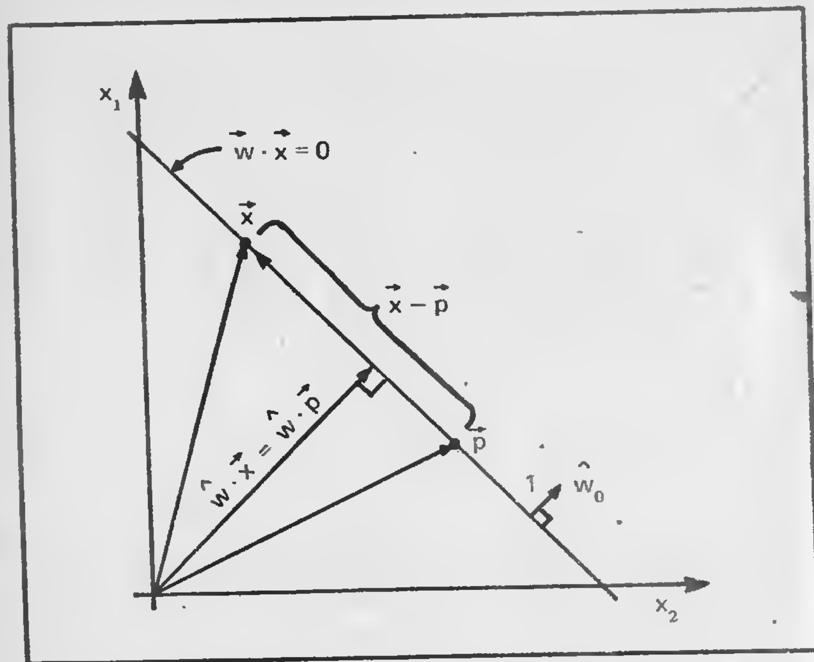


Рис. 11.6. Расстояние от начала координат до прямой равно $W_0 * X$, где X - любая точка, принадлежащая прямой

или

$$W * X = W * P$$

Раскрывая W и удаляя w_{n+1} из обеих частей равенства, имеем:

$$W_0 * X = W_0 * P$$

или

$$W_0 * (X - P) = 0$$

Произведение векторов равно нулю, а это означает, что вектор W_0 нормален по отношению к отрезку (X - P), который является частью данной прямой. Следовательно, W_0 нормален и ко всей прямой.

Обратите внимание на то, что расстояние от начала координат до прямой (рис. 11.4) равно норме $W_0 * P$. Это справедливо для любой точки X, принадлежащей прямой, так как

$$W_0 * X = W_0 * P$$

Направленное расстояние, D, до $W * X = 0$ определяется по формуле

$$D = W'_0 * X = -w_{n+1}/|W_0|$$

Знак показывает направление измерения относительно прямой.

Для любой точки Y, принадлежащей гиперплоскости, расстояние от $W * X = 0$ до Y равно:

$$D_y = W'_0 * Y - W'_0 * X$$

Первый терм представляет собой направленное расстояние от начала координат до точки Y, а второй - направленное расстояние от начала координат до прямой, Их разность и есть направленное расстояние от точки до прямой. Второй терм можно представить следующим образом:

$$W'_0 * X = W_0 * X/|W_0| = W * X/|W_0| - w_{n+1}/|W_0|$$

Так как $W * X = 0$, то

$$W'_0 * X = - w_{n+1}/|W_0|$$

Наконец, направленное расстояние от точки Y до гиперплоскости с вектором ориентации W_0 равно:

$$D_y = (W_0 * Y + w_{n+1})/ |W_0|$$

Суммируя все наши выкладки, можно охарактеризовать свойства гиперплоскостей таким образом:

* единичный вектор W_0 является нормалью к гиперплоскости $W * X = 0$ и определяет ее ориентацию;
* постоянный член w_{n+1} в уравнении гиперплоскости пропорционален расстоянию от начала системы координат до гиперплоскости. Если $w_{n+1} = 0$, то гиперплоскость проходит через начало координат.

Следовательно, рассматривая компоненты вектора W, мы можем многое узнать о представляющей его гиперплоскости.

Классификаторы, построенные по критерию минимального расстояния

Изучив некоторые свойства гиперплоскости, мы можем теперь обратить ваше внимание на разработку решающих функций. Чтобы создать решающую функцию в виде гиперплоскости, разделяющей две группы точек (принадлежащих двум различным классам), требуется найти на гиперплоскости две точки, каждая из которых представляет весь кластер целиком. Выбирается точка-*прототип*, или центр кластера, определяемый средним расстоянием до него от всех точек кластера, которое является средним арифметическим N точек кластера:

$$Z = (X_1 + X_2 + \ldots + X_n)/N$$

Поскольку способы определения прототипа нам известны, нужно теперь найти средства сопоставления заданного образа с прототипом. Одно из таких средств - эвклидово расстояние, т.е. абсолютная величина направленного расстояния. Для заданной классифицируемой точки прототип, находящийся ближе всего к ней, считается наиболее подходящим и точка относится к тому же классу, что и прототип. Такой подход называется *классификацией, построенной по критерию минимального расстояния*.

Расстояние от точки X до прототипа Z находится по формуле

$$D = |X - Z|$$

Для двумерного пространства (берется квадрат расстояния) оно определяется следующим образом:

$$D^2 = (x_1 - z_1)^2 + (x_2 - z_2)^2 = \\ = x_1^2 - 2x_1z_1 + z_1^2 + x_2^2 - 2x_2z_2 + z_2^2$$

В общем случае для множества классов расстояние до прототипа класса i

$$D_i^2 = |(X - Z_i)|^2 = (X - Z_i) * (X - Z_i) = \\ = X*X - 2X*Z_i + Z_i*Z_i$$

Теперь, имея формулу для D^2, построим с ее помощью решающую функцию классификации по критерию минимального расстояния. Поскольку выражение X * X не зависит от класса, его можно устранить. Тогда, умножая оставшуюся часть на -1/2, получаем следующую решающую функцию:

$$d_i(X) = X * Z_i - (1/2)Z_i * Z_i$$

В формуле для определения d_i мы умножаем D на -1/2, чтобы привести выражение к виду, при котором чем больше значение di, тем полнее аналогия (меньше расстояние).

Далее можно вычислить компоненты W. Так как

$$d(X) = W * X = 0 \quad \text{то}$$
$$w_i = z_i \quad \text{для всех } i = 1, \ldots, n$$
$$w_{n+1} = -(1/2)Z_i * Z_i$$

Геометрическая интерпретация d_i показана на рис. 11.7. Первый терм d_i является проекцией X на Z, а второй равен половине длины Z. В том случае, когда проекция X на Z больше

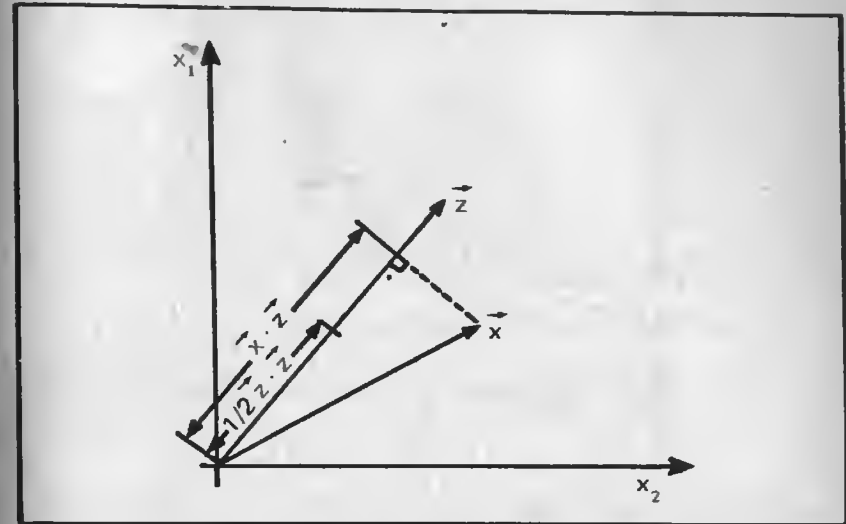


Рис. 11.7. Классификация X по критерию минимального расстояния относительно точки-прототипа Z. Здесь приводится графическая иллюстрация решающей функции d(X) = X * Z - (1/2)Z * Z. Если проекция X на Z больше половины длины Z, то d>0

половины Z, решающая функция d_i положительна.

Если при классификации по критерию минимального расстояния используется классификатор типа 3, то решающие функции (при двух измерениях) представляют собой прямые, перпендикулярные к отрезкам, соединяющим точки-прототипы, и делящие их пополам. Таким образом, расстояния от решающей функции до прототипов, расположенных по обе ее стороны, одинаковы. Поскольку решающая функция занимает серединное положение между кластерами, очевидно, что она выполняет классификацию по критерию минимального расстояния.

9 Заказ № 299

В качестве примера создания классификатора, действующего по критерию минимального расстояния, рассмотрим образы, приведенные на рис. 11.8, где представлены три кластера, а следовательно, и три класса. Для осуществления необходимых вычислений была разработана Форт-программа, текст которой приведен на экранах с 90 по 95. Структуры данных, создаваемые словом КЛАСС, изображены на экране 90, а словом КЛАССЫ - на экране 92. КЛАСС создает отдельные классы кластерных точек, используемых при определении решающих функций. Память под слово КЛАСС распределяется следующим образом:

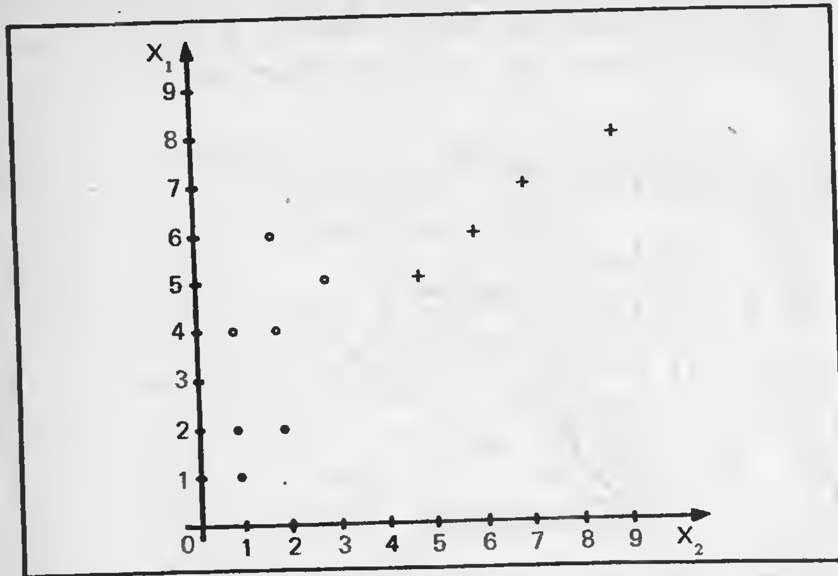


Рис. 11.8. Три класса образов (по четыре в каждом) а двумерном пространстве

Адрес	Содержимое памяти
pfa:	Число кластерных точек в классе
pfa + 2:	Значение x точки-прототипа
pfa + 4:	Значение y точки-прототипа
pfa + 6:	w_{n+1}
pfa + 8:	Значение x первой точки кластера

Слово, определенное через КЛАСС, при своем исполнении выбирает из стека индекс, а возвращает в него указатель на индексированную точку кластера. Слово ТОЧКА выполняет скалярное умножение двух точек, указатели которых находятся в стеке. На стек возвращается число, равное скалярному произведению точек.


```
   90
 0 \ АЛГОРИТМЫ РАСПОЗНАВАНИЯ ОБРАЗОВ
 1 \ КЛАСС - ОПРЕДЕЛЯЮЩЕЕ КЛАССЫ СЛОВО; СОДЕРЖИТ:
 2 \ # ТОЧЕК В КЛАССЕ,
 3 \ ТОЧКУ-ПРОТОТИП, W(N+1), ТОЧКИ КЛАСТЕРА (X, Y)
 4
 5  ( N -> ) \ # ОБУЧАЮЩИХ ТОЧЕК В КЛАССЕ
 6 : КЛАСС CREATE DUP , 6 ALLOT 4 * ALLOT
 7  ( N -> @) \ @ УКАЗЫВАЕТ НА ТОЧКУ (X, Y)
 8  DOES> DUP -ROT @ MIN 4 * + 8 +
 9 ;
10
11 \ СКАЛЯРНОЕ ПРОИЗВЕДЕНИЕ ВЕКТОРОВ
12  ( @V1 @V2 -> N)
13 : ТОЧКА 2DUP @ SWAP @ * -ROT 2+ @ SWAP 2+ @ * + ;
14
15
```

```
   91
 0 \ АЛГОРИТМЫ РАСПОЗНАВАНИЯ ОБРАЗОВ
 1
 2 ( 'КЛАСС -> ) \ УСРЕДНЕННЫЙ ВЕКТОР -'КЛАСС ЭТО CFA КЛАССА
 3 : ПРОТОТИП DUP >BODY DUP 2+ 6 0 FILL @ DUP >R 0
 4  DO DUP >BODY 2+ OVER I SWAP EXECUTE
 5     2DUP @ SWAP +! 2+ @ SWAP 2+ +!
 6  LOOP >BODY 2+ DUP DUP @ R@ / OVER ! 2+ DUP @ R> / SWAP ;
 7  DUP 4 + SWAP DUP ТОЧКА 2/ SWAP ! \ ВЫЧИСЛЕНИЕ W(N+1)
 8 ;
 9 \ РЕШАЮЩАЯ ФУНКЦИЯ ПО ПРИНЦИПУ МИНИМАЛЬНОГО
10 \ РАССТОЯНИЯ
11  ( 'КЛАСС @X -> N)
12 : D(X) SWAP >BODY 2+ DUP 4 + @ -ROT ТОЧКА SWAP - ;
13
14
15
```

```
   92
 0 \ АЛГОРИТМЫ РАСПОЗНАВАНИЯ ОБРАЗОВ
 1
 2  ( N ->)
 3 : КЛАССЫ CREATE DUP , 2* ALLOT
 4  ( N -> 'КЛАССN)
 5  DOES> DUP @ ROT MIN 2* + 2+ @
 6 ;
 7
 8  ( @X 'КЛАССЫ -> N) \ КЛАССИФИКАЦИЯ ТОЧКИ (X, Y) В @X
 9 : КЛАССИФИКАЦИЯ DUP 0 0 ROT >BODY @ 0
10   DO I 3 PICK EXECUTE 4 PICK D(X) DUP 3 PICK >
11     IF -ROT 2DROP I ELSE DROP THEN
12   LOOP NIP NIP NIP
13 ;
14
15
```

Экраны 90, 91, 92. Определения слов КЛАСС, ТОЧКА, ПРОТОТИП, D(X), КЛАССЫ, КЛАССИФИКАЦИЯ

```
     93
 0 \ АЛГОРИТМЫ РАСПОЗНАВАНИЯ ОБРАЗОВ
 1
 2 ( X1 Y1 Y2 ... XN YN N 'КЛАСС -> )
 3 : >КЛАСС >BODY DUP >R 8 + DUP ROT DUP R> ! 4 * + 2-
 4 DO I ! -2 +LOOP
 5 ;
 6
 7  ('КЛАСС1 'КЛАСС2 ... 'КЛАССN -> )
 8 : >КЛАССЫ ' >BODY DUP >R 2+ DUP ROT DUP R> ! 2* + 2-
 9    DO I ! -2 +LOOP
10 ;
11
12
13
14
15
     94
 0 \ АЛГОРИТМЫ РАСПОЗНАВАНИЯ ОБРАЗОВ
 1
 2 \ ПРИМЕР
 3
 4 4 КЛАСС КЛАСС1  0 2  1 1  2 1  2 2  4 '  КЛАСС1 >КЛАСС
 5
 6 4 КЛАСС КЛАСС2  5 3  4 1  4 2  6 2  4 '  КЛАСС2 >КЛАСС
 7
 8 4 КЛАСС КЛАСС3  6 6  5 5  8 9  7 7  4 '  КЛАСС3 >КЛАСС
 9
10 3 КЛАССЫ КЛАСС1
11 ' КЛАСС1 ' КЛАСС2 ' КЛАСС3 3 >КЛАССЫ КЛАССЫ1
12
13 VARIABLE X 2 ALLOT
14
15
     95
 0 \ АЛГОРИТМЫ РАСПОЗНАВАНИЯ ОБРАЗОВ
 1
 2 ' КЛАСС1 ПРОТОТИП
 3 ' КЛАСС2 ПРОТОТИП
 4 ' КЛАСС3 ПРОТОТИП
 5
 6 4 X ! 4 X 2+ ! \ X = (4, 4)
 7
 8 X ' КЛАССЫ1 КЛАССИФИКАЦИЯ
 9
10
11
12
13
14
15
```

Экраны 93, 94, 95. Определения слов >КЛАСС и >КЛАССЫ

Слово ПРОТОТИП, изображенное на экране 91, выбирает из стека cfa слова-класса и вычисляет точку-прототип и w_{n+1} для нее, а также запоминает эти значения в соответствующие поля данных самого слова-класса. Слово D(X) является решающей функцией. Оно выбирает указатель на точку, подлежащую классификации, а также cfa класса, и возвращает в стек число, позволяющее судить о степени принадлежности данной точки к классу. Чем больше число, тем вероятнее принадлежность. Слово КЛАССЫ, показанное на экране 92, определяет классы в области образов. По его pfa располагается число классов, а затем по порядку адреса cfa каждого класса (класс 0 на первом месте). Наконец, слово КЛАССИФИКАЦИЯ выбирает указатель на точку данных X' и cfa области образов, а возвращает число назначенных классов.

На экране 93 представлены два слова, позволяющие упростить заполнение в словах КЛАСС и КЛАССЫ выделенной памяти данными. Их использование показано на экране 94, где класс 1 (КЛАСС1) определен четырьмя точками в области образов: (0,2), (1,1), (2,1) и (2,2). Экран 95 иллюстрирует вычисление точки-прототипа каждого класса и w_{n+1}. Далее классифицируемая точка помещается в X. На экране 95 X получает значение (4,4). Наконец, вызывается слово КЛАССИФИКАЦИЯ с параметрами X и КЛАССЫ1, где содержатся три класса. В результате мы имеем:

Класс	Точка-прототип	w_{n+1}
0	(1,1)	1
1	(4,2)	10
2	(6,7)	36

Слово КЛАССИФИКАЦИЯ относит точку (4,4) к классу 1.

АЛГОРИТМЫ КЛАССИФИКАЦИИ ОБРАЗОВ

Для классификатора, построенного с помощью критерия минимального расстояния, по набору точек с заданной классификацией вычисляются точки-прототипы. Затем могут быть классифицированы новые точки посредством решающих функций, построенных на базе этих прототипов. Таким образом, по точке-прототипу строится вектор весов W, используемый для определения решающей функции-гиперплоскости.

Теперь рассмотрим несколько алгоритмов автоматической классификации образов. Методы кластеризации с заранее определенными классами лежат в основе *неуправляемого* (*unsupervised*) обучения. Если же алгоритмы учатся правильно распознавать образы с использованием обратной связи по завершении процесса классификации, то такое обучение называется *управляемым* (*supervised*). В ЭС применяются оба вида обучения.

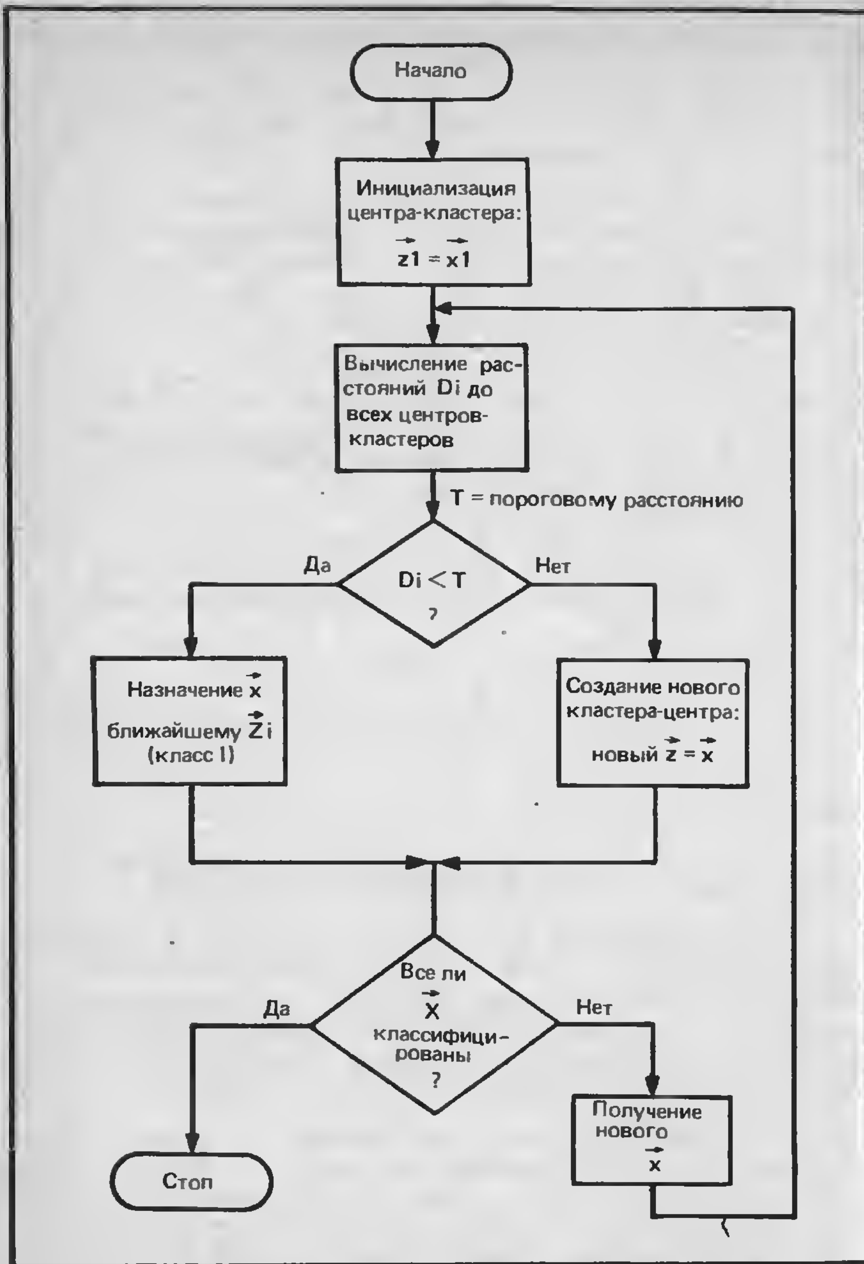


Рис. 11.9. Пороговый алгоритм классификации

На рис. 11.9 показан простой алгоритм кластеризации, известный как *пороговый алгоритм (threshold algoritm)*. Критерием для отнесения образа к какому-либо классу здесь служит пороговое расстояние T. Если образ находится в пределах расстояния T от некоторого прототипа, то он будет отнесен к данному кластеру. Если же рассматриваемый образ находится от любого прототипа дальше, чем на расстояние T, он становится новым прототипом. Первоначально точка-прототип определяется произвольно. Ею может быть первый образ X_1. Затем вычисляется расстояние от следующего образа до Z_1 (и до любого другого прототипа, если он имеется). Отнесение данного образа к тому или иному классу основывается на сравнении его расстояния до прототипа с T.

Пороговый алгоритм прост, но он не свободен от недостатков. Во-первых, существенное влияние на выбор прототипа оказывает порядок рассмотрения образов. Положение можно улучшить, если задать приблизительные значения прототипов. Во-вторых, выбор T влияет на разрешающую способность кластеризации. На рис. 11.10A представлено изменение числа кластеров с изменением T. "Нормальное" значение T должно находиться в том интервале, где данная зависимость линейна (между T_1 и T_2). В этом диапазоне алгоритм сравнительно не чувствителен к T и обеспечивает оптимальную кластеризацию. На рис.11.10B показана область образов применительно к нашей ситуации. Два кластера A и B будут четко разделены любым значением T между T_1 и T_2. В том случае, когда A и B перекрываются, ни одно из значений T не разделит их. Они действительно линейно неразделимы и здесь требуется применение статистических классификаторов.

Алгоритм, построенный по критерию максимального расстояния, обладает лучшими качествами. Как показано на рис. 11.11, он использует ту же меру длины, что и рассмотренный выше алгоритм. Здесь пороговое значение не фиксируется, а переопределяется после классификации всех образов, после чего осуществляется переклассификация. В целях коррекции T образец, наиболее удаленный от своего прототипа, сравнивается с другими по среднему расстоянию между прототипами. Если это расстояние превышает среднее на половину среднего расстояния, то создается новый прототип. Аналогичным образом происходит сравнение самых дальних точек для каждого прототипа. При образовании новых прототипов осуществляется повторная классификация.

Алгоритм, построенный по принципу максимального расстояния, имеет преимущество перед алгоритмом с пороговым значением, поскольку предусматривает коррекцию T. Однако он чувствителен к порядку рассмотрения образов и к ложным образам (помехи в передаче данных), так как самая дальняя точка вероятнее всего окажется шумовой. Поскольку для нахождения максимального расстояния вычисляется расстояние от каждой точки до ее прототипа, при большом числе образов на это уходит очень много

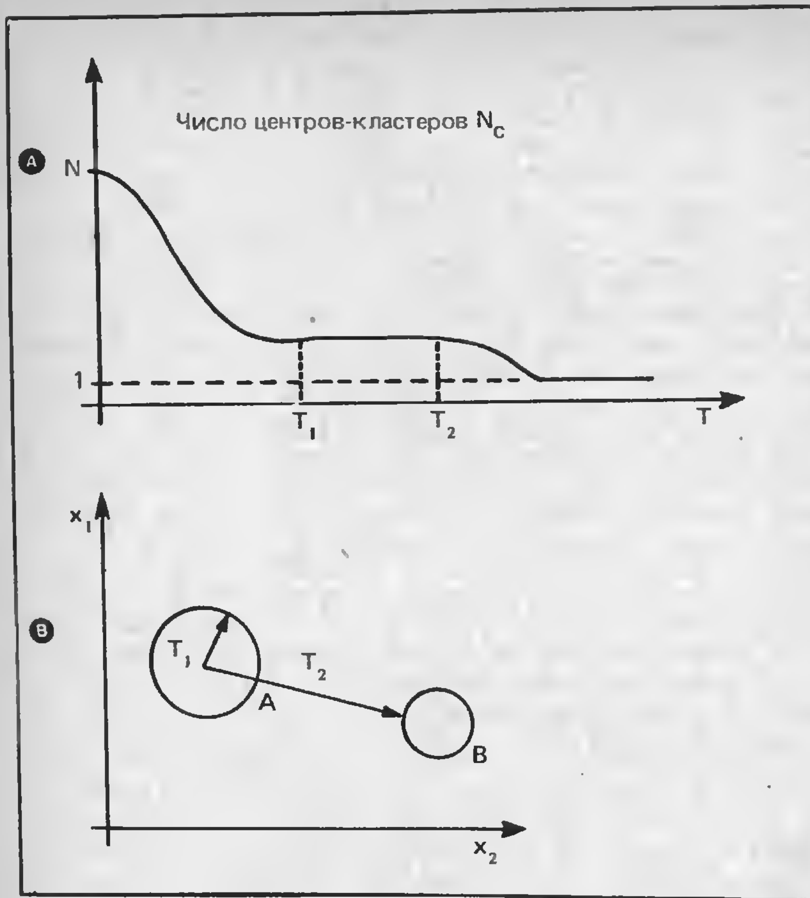


Рис. 11.10. Изменение числа кластеров с изменением T между T_1 и T_2 линейно (A), так как в этом интервале кластеры A и B разделены (B)

времени. Если в качестве нового прототипа выбрать самую отдаленную точку, то отдельный кластер может быть ошибочно слит с ближайшим к нему кластером, потому что на первых порах расстояние между кластерами велико. Наконец, алгоритм не будет функционировать при определенном распределении образов, например в том случае, когда классифицируемая точка находится посередине между двумя прототипами с равным числом точек.

Чувствительность алгоритма к порядку расположения образов можно уменьшить. Имеет смысл сначала изучить распределение образов и сделать предварительный выбор центров кластеров.

Начало

Выбор начального $\vec{x}_1$

Нахождение самой дальней точки от $\vec{z}_1$, положить ее равной $\vec{z}_2$

Выбор $\vec{x}$

Вычисление для $\vec{x}$ всех D

Определение принадлежности $\vec{x}$ к классу, ближайшему к $\vec{z}$

Все ли $\vec{x}$ выбраны? — Нет / Да

Вычисление среднего расстояния между всеми $\vec{z}$

Выбор $\vec{x}$ из класса Zi, являющийся самым дальним по отношению к $\vec{z}_i$

$\|\vec{x} - \vec{z}_i\| >$ D/2? — Да / Нет

Пусть новый $\vec{z} = \vec{x}$

Все ли классы рассмотрены? — Да / Нет

Были ли изменения кластеров-центров? — Нет

Конец

Рис. 11.11. Алгоритм классификации по критерию расстояния

Алгоритм усреднения по К (*K-means algorithm*) осуществляет выбор из (или ему задается) К прототипов. С помощью классификатора, построенного по критерию минимального расстояния, он относит каждую из оставшихся точек к одному из К заданных кластеров, после чего для каждого кластера вычисляется новый прототип с учетом вновь приписанных образов (путем нахождения средней точки, как мы это выполняли ранее). Классификация повторяется до тех пор, пока распределение образов по классам не завершится. Алгоритм усреднения по К практически перераспределяет прототипы по К заданным прототипам. Сам он выбирает исходные прототипы произвольным образом и поэтому остается чувствительным к порядку расположения образов. Если же аппроксимация К прототипов будет проведена до функционирования алгоритма, то он уже не может считаться "автоматическим" при выборе прототипов, как алгоритм, созданный по принципу максимального расстояния.

Более общий подход к кластеризации образов должен включать средства объединения кластеров, которые срослись (с использованием правил "отмирания"), а также средства генерации новых кластеров (с применением правил "порождения"). Поскольку образцы представляют собой данные измерений в некоторой области (или свойства, извлеченные из исходных данных), для описания этой области могут привлекаться правила из базы знаний, чтобы решения, принимаемые при классификации, были бы уже частично доказанными. При таком подходе сочетание численных и символьных преобразований позволило бы системе обучаться непосредственно на данных, получаемых с датчиков.

Так как процесс обучения требует новой информации об области, желательно иметь компьютер (или робот), который снимал бы эти данные непосредственно с датчиков, вместо того, чтобы вручную преобразовывать их в форму правил. В настоящее время в большинстве систем представления знаний принята соответствующая техника представления данных для обучения, которая является сенсорным расширением систем построения баз знаний и поставляет системе новую информацию вместо неэффективного человеко-машинного интерфейса. С применением роботов происходит объединение познавательных способностей систем, основанных на знаниях, и искусственных сенсомоторных систем, что ведет к сокращению такого интерфейса. Предполагается, что в будущем взаимодействие человека с системами управления базами знаний должно осуществляться через органы восприятия роботов, причем роботы могут быть как автономными, так и управляемыми человеком. Общение с ними будет подобно обычному человеческому общению, а способов общения станет гораздо больше, чем предоставляет нам сейчас клавиатура, дисплей и световое перо.

В одном из ранних проектов по обучающим системам был разработан алгоритм управляемого обучения - так называемый *алгоритм восприятия* (*perceptron algoritm*). В результате обучения по такому алгоритму создается вектор точных весов для классификации линейно разделимых кластеров. После обучения на образах известной классификации (*обучающем множестве - training set*) этот алгоритм может осуществлять классификацию образов из неизвестных классов (рис. 11.12). Данный метод легко расширить для классификации по нескольким классам.

Обучающее множество состоит из N образов в каждом из двух классов. Решающая функция для образов, принадлежащих классу 1, получает положительные числа:

$$X_{1i} * W > 0,\ i = 1,\ 2,\ \ldots,\ N$$

а для образов из класса 2 - отрицательные:

$$X_{2i} * W < 0,\ i = 1,\ 2,\ \ldots,\ N$$

или:

$$-X_{2i} * W > 0,\ i = 1,\ 2,\ \ldots,\ N$$

Заменив $-X_{2i}$ на X_{2i}, мы добились того, что функция $d(X)$ будет одинаково применяться к образам обоих классов. Алгоритм восприятия выбирает образ и определяет для него значение решающей функции. Если это значение положительно, образ классифицирован правильно, и корректировка W не требуется. В противном случае происходит корректировка W. Процедура повторяется до тех пор, пока все образы не будут правильно классифицированы по заданному вектору весов.

При отрицательном значении $d(X)$ W увеличивается на величину $c*X$, где c - положительное приращение коррекции. К-й шаг процедуры коррекции выглядит следующим образом:

$$W(k + 1) = W(k), \quad \text{если } W(k) * X > 0$$
$$W(k + 1) = W(k) + cX, \quad \text{если } W(k) * X <= 0$$

В результате коррекции W улучшается, поскольку значение функции теперь равно:

$$d(X) = W(k + 1) * X = (W(k) + cX) * X = W(k) * X + cX * X$$

Полученное новое значение $d(X)$ должно превышать предыдушее значение, так как оно составляет лишь первое слагаемое текущего значения - $W(k) * X$. Текущее значение превышает предыдущее на величину второго слагаемого $cX * X$, значение которого должно быть положительным, поскольку и c, и $X * X$ положительны (X не равен 0).

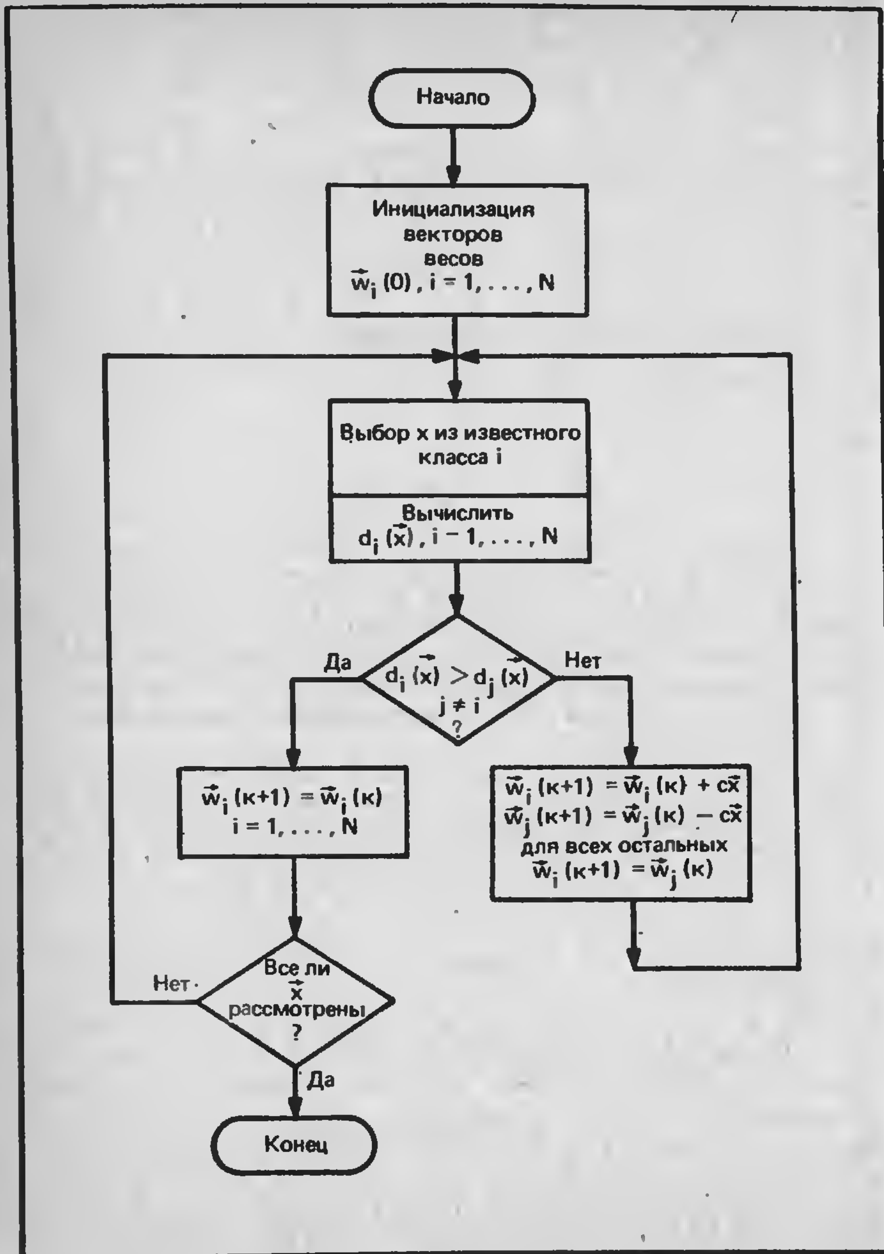


Рис. 11.12. Алгоритм восприятия или управляемого обучения

Рассмотрим выполнение приведенного алгоритма на примере обучающего множества, состоящего из двух образов в каждом классе в двумерном пространстве:

$$\text{класс 1: } X_1 = [3\ \ 1]^T, \quad X_2 = [1\ \ 2]^T$$
$$\text{класс 2: } -X_3 = [1\ -2]^T, \quad -X_4 = [2\ -1]^T$$

На рис. 11.13 показаны состояние обучающего множества и результат коррекции W на различных шагах выполнения алгоритма при заданных значениях c = 1 и W(0) = 0. Результирующий вектор весов $[4\ -1]^T$ перпендикулярен прямой, разделяющей два класса. X увеличивается на константный терм x_{n+1} (аналогично и W), как описано в разд. "Свойства гиперплоскости". Поскольку $d(X_1) = 0$, W подлежит изменению, в результате чего получается $W(1) = [3\ 1\ 1]^T$:

$$W(1) = W(0) + 1 * X_1 = [0\ 0\ 0]^T + [3\ 1\ 1]^T = [3\ 1\ 1]^T$$

Шаг	Класс	W	X	d(X)	Изменение W
0	1	(0, 0, 0)	(3, 1, 1)	0	да
1	1	(3 1 1)	(1, 2, 1)	6	нвт
2	2	(3, 1, 1)	(1, -2, -1)	0	да
3	2	(4, -1, 0)	(2, -1, -1)	9	нет
4	1	(4, -1, 0)	(3, 1, 1)	11	нет
5	1	(4, -1, 0)	(1, 2, 1)	2	нет
6	2	(4, -1, 0)	(1, -2, -1)	6	нет

Рис. 11.13. Пример выполнения алгоритма восприятия при нвчальном значении W = 0 и четырех образах в двумерном простргнстве при c=1

Далее:

$$d(X_2) = W * X_2 = [3\ 1\ 1][1\ 2\ 1]^T = 6$$

Результат положителен, W остается без изменений, или W(2)=W(1). Выполнение процедуры продолжается, как показано на рис. 11.11.

Что заставляет алгоритм восприятия выполняться? Из изложенного выше мы знаем, что W перпендикулярен поверхности решения решающей функции d(X) = 0. Для заданного образа X вектор W ориентирован в том же самом общем направлении, если W * X > 0. Угол между ними составляет меньше 90 градусов, пото-

му что скалярное произведение двух векторов равно:

$$W * X = |W| * |X| * \cos A$$

где A - угол между W и X.

Так как норма векторов всегда неотрицательна, знак скалярного произведения определяется ориентацией векторов. Следовательно, решающая функция является мерой степени направленности W и X в одну и ту же сторону. Алгоритм восприятия ищет такой вектор W, который указывает в том же направлении, что и векторы образов (образы класса 2 отрицательны). При добавлении c*X к W последний еще более ориентируется в направлении X. В результате коррекции W * X всегда улучшается. За счет добавления образов X_i к W тот выравнивается по отношению к X_i. Средняя величина вычисляется следующим образом:

$$M = (X_1+X_2+...+X_n)/N$$

В процессе выполнения алгоритма W приближается к M. По существу, это классификация на основе критерия минимального расстояния. Различие состоит лишь в том, что W не вычисляется по набору образов известной классификации, а обучается на данном наборе.

Можно оптимизировать коррекцию W путем дифференцированного подбора c, что и было сделано выше. Процесс будет сходиться гораздо быстрее, если мы вместо произвольной фиксации значения c, равного единице, присвоим c значение, необходимое для правильной классификации после коррекции весов. На первом шаге выполнения для заданного образца значение d(X) положительно. Определим оптимальное значение c на этом шаге, где W(k+1)*X > 0, или

$$W(k + 1) * X = (W(k) + cW) * X > 0$$

Разрешим это неравенство относительно c:

$$c > -W(k) * X/X * X = -W(k) * X/|X|^2$$

Так как X * X > 0 и W(k)* X < 0 (классификация была неудачной), имеем: -W(k)* X <= 0. Присваивая c найденное значение, называемое *абсолютным приращением коррекции* (*absolute-correction increment*), получаем *алгоритм восприятия с абсолютным приращением коррекции* (*absolute-correction perceptron algorithm*).

Несмотря на то что в алгоритмах восприятия, функционирующих с несколькими классами, может быть заложен любой из трех мультикатегорийных классификаторов, все-таки классификатор типа 3 допускает применение этого алгоритма в более общей форме. Для образа из класса i, если d_i не больше d_j (i не равен j), вектор

весов для d_i увеличивается, а для d_j уменьшается. Остальные векторы весов не претерпевают изменений.

Итак, мы рассмотрели область распознавания образов, причем довольно кратко, поскольку многие методы аналогичны методам, применяемым в ИИ с той лишь разницей, что в ИИ они основаны на символьных преобразованиях, а здесь на числовых. Нужно отметить, что мы не затронули такую важную тему РО, как *статистическое распознавание образов*, хотя в реальных программах требуется именно этот подход. Детерминированная классификация является тем базисом, без которого невозможно уяснить статистические методы.

УПРАЖНЕНИЯ

1. Расширьте набор слов Форта, предназначенных для выполнения классификации (экраны 90 - 95), с тем, чтобы векторы имели произвольную длину, в том числе и максимальную N. Такой более универсальный набор слов смог бы классифицировать образы в любом пространстве вплоть до N-мерного.

2. Напишите программу, реализующую пороговый алгоритм. Создайте несколько опытных образов и проверьте на них работу вашей программы.

3. Определите функцию $N_C(T)$, значение которой - число центров кластеров - определялось бы как функция от порогового значения T порогового алгоритма из упр. 2. Постройте график N_C.

4. Напишите программу алгоритма, созданного по принципу максимального расстояния, и испытайте ее на нескольких выбранных образах.

5. Напишите программу алгоритма усреднения по K и испытайте ее на выбранных образах. Пусть эта программа выберет прототипы случайным образом, а затем инициирует аппроксимацию кластерных центров по данным прототипам. Зафиксируйте различие в выполнении для двух отдельных случаев.

6. Напишите обучаемую программу восприятия для двух классов с образами в двумерном пространстве. Напечатайте табличное значение W и X, а также d(X) на каждом шаге при:

а) c = 1,

б) c = абсолютному приращению коррекции.

Приложение А

ИСХОДНЫЕ ТЕКСТЫ ПРОГРАММ

В настоящем приложении в полном объеме приводятся исходные тексты программ, реализующих интерпретатор Пролога, который был описан в гл.7, 8 и 9. Наберите содержимое приведенных ниже экранов, руководствуясь имеющимся у вас описанием стандарта Форт-83.

Если вы работаете на IBM PC в среде F83, созданной Перри и Лэксеном (поставляется как часть программного обеспечения на дискете), придерживайтесь правил инициализации Пролог-интерпретатора:

1. Скопируйте содержимое исходной дискеты на другую дискету или на жесткий диск.
2. Словом FORTH инициируйте Форт-систему.
3. Сделайте доступными блоки интерпретатора Пролога, введя текст:

```
OPEN PROLOG.BLK
```

Помните, что все литеры должны быть прописными.
4. Убедитесь в работоспособности Форт-системы, для чего выполните следующие действия:
а) распечатаите последовательность блоков с помощью слова SHOW, например, так:

```
37 74 SHOW
```

б) выведите любой блок словом LIST:

```
37 LIST
```

в) используя слово WORDS, просмотрите список доступных слов Форта;
г) словом SEE выполните декомпиляцию любого слова:

```
SEE LIST
```

5. Загрузка Пролог-интерпретатора осуществляется программой, приведенной на экрана 38:

```
37 LOAD
```


После окончания загрузки на экране дисплея появятся сообщения типа "...isn't unique". Это нормальное завершение, так как при загрузке были переопределены некоторые базовые слова Форта. В частности, теперь слово LIST входит в состав Пролога[1]. Для выдачи же блока следует обращаться к слову XLIST:

```
37 XLIST
```

6. После загрузки последнего экрана (экран 72) будут созданы списки (распечатайте экран 72 командой 72 XLIST). Просмотреть их можно с помощью следующих команд:

```
СП1 @ ВЫДАТЬ
СП2 @ ВЫДАТЬ
СП3 @ ВЫДАТЬ
```

7. Для проверки функции ЧТСП загрузите блок 50:

```
50 LOAD
I1 @ ВЫДАТЬ
I2 @ ВЫДАТЬ
I3 @ ВЫДАТЬ
```

8. Загрузите базу знаний и проверьте ее работоспособность:

```
70 LOAD
ЦЕЛИ @ ВЫДАТЬ
ПРЕДЛОЖЕНИЯ @ ВЫДАТЬ
СЛЕД
```

9. Повторите те же действия с экраном 71:

```
71 LOAD
ЦЕЛИ @ ВЫДАТЬ
ПРЕДЛОЖЕНИЯ @ ВЫДАТЬ
СЛЕД
```

Для лучшего уяснения приведенных ниже программ прочитайте главы с 6-й по 10-ю.

Примечание. Для сохранения в любой момент времени состояние памяти выполните команду SAVE-SYSTEM из стандарта Форт-83, указав имя файла:

```
SAVE-SYSTEM ANIMAL.COM
```

Файлы с расширением .COM аналогичны командам DOS, т.е. выполняются без загрузки.

---

[1] В данном переводе книги возможность конфликта между именами слов Форта и Пролога исключена - все слова, не входящие в стандартное ядро Форта, в том числе и слова, реализующие Пролог, приведены в русской транскрипции. В частности, слово Пролога LIST при переводе получило имя СПИСОК. - *Прим.перев.*

```
Экр # 37          C:PROLOG.BLK
 0 \ ЗАГРУЗКА ПРОЛОГА
 1 ONLY FORTH ALSO DEFINITIONS
 2 DECIMAL
 3 : XLIST LIST ;
 4 39 49 THRU
 5 51 52 THRU
 6 60 63 THRU
 7 66 68 THRU
 8 73 74 THRU
 9 69 LOAD
10 64 65 THRU
11 53 54 THRU
12 72 LOAD
13
14
15
```

```
Экр # 39          C:PROLOG.BLK
 0 \ МОДИФИЦИРОВАННЫЙ ОПЕРАТОР CASE ДЛЯ
   1 \ ФОРТА ПЕРРИ И  ЛЭКСЕНА
 2 : CASE 0 ; IMMEDIATE
 3
 4 : OF COMPILE OVER
 5      COMPILE =
 6      [COMPILE] IF
 7      COMPILE DROP ; IMMEDIATE
 8 : ENDOF [ COMPILE] ELSE ; IMMEDIATE
 9
10
11 : ENDCASE COMPILE DROP BEGIN ?DUP WHILE
12      [COMPILE] THEN REPEATE ; IMMEDIATE
13
14
15
```

```
Экр # 40          C:PROLOG.BLK
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
   1 \ ЛИСПА НА ФОРТЕ-83
 2 VARIABLE НУЛЬ НУЛЬ НУЛЬ ! \ ПУСТОЙ СПИСОК
 3
 4 ( #ЭЛЕМЕНТОВ -> )
 5 \ #ЭЛЕМЕНТОВ=МАКСИМ.ЧИСЛУ ЭЛЕМЕНТ. СПИСКА
 6 : НОВСПИСОК CREATE HERE 2+ , НУЛЬ , 2* ALLOT ;
 7 ( @СПИСОК -> @ПЕРВЫЙ)
 8 \ @ПЕРВЫЙ-УКАЗАТ.НА ПЕРВЫЙ ЭЛЕМ.СПИСКА
 9 : ПЕРВЫЙ @ ;
10  ( @СПИСОК | НУЛЬ -> ФЛАГ)
11 \ ФЛАГ = ИСТИНА, ЕСЛИ СПИСОК ПУСТОЙ
12 : НОЛЬ @ НУЛЬ = ;
13  ( @СПИСОК -> @ХВОСТ)
14 \ @ХВОСТ - УКАЗАТЕЛЬ НА ОСТАТОК  СПИСКА
15 : ХВОСТ DUP НОЛЬ IF @ ELSE 2- THEN ;
```



```
Экр # 41          C:PROLOG.BLK
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2  ( I -> ) \ СПИСОК ДЕЛАЕТСЯ НУЛЕВЫМ (ПУСТЫМ)
 3 : ПУСТОЙ DUP 2+ DUP ROT ! НУЛЬ SWAP ! ;
 4
 5  ( 1 @СПИСОК -> ) \ УСТАНОВКА ИМЕНИ СПИСКА I
 6 \ НА @СПИСОК
 7 : УСТАНОВИТЬ DUP НУЛЬ = IF DROP ПУСТОЙ
 8 \ ELSE  SWAP ! THEN ;
 9  ( @ЭЛЕМЕНТ I -> ) \ ДОБАВЛ. @ЭЛЕМЕНТ К ГОЛОВЕ
10 \ СПИСКА I
11  : СВЯЗЬ 2 OVER +! @ ! ;
12 \ СЛОВО, ВЫПОЛНЯЮЩЕЕ РЕКУРСИЮ
13 : РЕКУРСИЯ LAST @ NAME> , ; IMMEDIATE
14
15
```

```
Экр # 42          C:PROLOG.BLK
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2  ( НУЛЬ S1 S2 . . . SN I -> ) \ ПОСТРОЕНИЕ СПИСКА
 3                        \ С ИМЕНЕМ I
 4 : СПИСОК >R
 5   BEGIN DUP НОЛЬ NOT
 6   WHILE R@ СВЯЗЬ
 7   REPEATE R> 2DROP  ;
 8  ( @СПИСОК I -> ) \ РЕКУРСИВНОЕ ОПРЕДЕЛЕНИЕ
 9              \ 2СОЕД
10 : 2СОЕД OVER НОЛЬ
11   IF 2DROP
12   ELSE OVER ХВОСТ OVER РЕКУРСИЯ
13     SWAP ПЕРВЫЙ SWAP СВЯЗЬ
14   THEN
15 ;
```

```
Экр # 43          C:PROLOG.BLK
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2 ( @ -> ФЛАГ) \ ФЛАГ = ИСТИНА, ЕСЛИ @ ДАЕТ PFA
 3            \  ПЕРЕМЕННОЙ
 4 : ATOM? BODY> @ ['] НУЛЬ @ = ;
 5  ( @СПИСОК -> )
 6 : ВЫДАТЬСП CR ." ("
 7   BEGIN DUP ПЕРВЫЙ DUP ATOM?
 8     IF DUP НОЛЬ NOT
 9       IF BODY> >NAME .ID ELSE DROP THEN
10     ELSE РЕКУРСИЯ
11     THEN ХВОСТ DUP НОЛЬ
12   UNTIL 8 ( ЗАБОЙ ) EMIT ." ) " DROP
13 ;
14
15
```

```
Экр # 44        C:PROLOG.BLK
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2  ( @СПИСОК -> )
 3 : ВЫДАТЬ DUP @ НОЛЬ
 4    IF DROP CR ." НУЛЬ"
 5    ELSE DUP ATOM?
 6       IF BODY> >NAME .ID
 7       ELSE ВЫДАТЬСП
 8       THEN
 9    THEN
10 ;
11
12
13
14
15
```

```
Экр # 45        C:PROLOG.BLK
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2  ( -> C) \ ЧТЕНИЕ ОЧЕРЕДНОГО СИМВОЛА ИЗ
 3          \ ВХОДНОГО ПОТОКА
 4 : ЧТСИМ BEGIN ИСТОЧНИК >IN @ /STRING
 5          IF C@ 1 >IN +! TRUE
 6          ELSE DROP ['] ИСТОЧНИК >BODY @ [']
 7                      (ИСТОЧНИК) =
 8            IF QUERY ELSE 1  BLK +! 0 >IN ! THEN FALSE
 9          THEN
10          UNTIL  ;   ( -> CFA)
11 \ ЕСЛИ СЛОВА НЕТ В СЛОВАРЕ,
12 \ СОЗДАЕТСЯ ПЕРЕМЕННАЯ
13 : ?CREATE >IN @ DEFINED
14 IF NIP ELSE DROP >IN ! HERE VARIABLE 4 + NAME>
15 THEN ;
```

```
Экр # 46        C:PROLOG.BLK
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2  ( I -> )
 3 : ЧТСП >R
 4    BEGIN ЧТСИМ
 5      CASE BL
 6        OF FALSE ENDOF ASCII (
 7        OF НУЛЬ FALSE ENDOF ASCII )
 8        OF R@ СПИСОК TRUE ENDOF ASCII @
 9        OF @ FALSE ENDOF
10        -1 >IN +! ?CREATE EXECUTE FALSE ROT
11      ENDCASE
12    UNTIL R> DROP
13 ;
14
15
```



```
Экр # 47        C:PROLOG.BLK
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2   UNNEST
 3 ( @СПИСОК -> @ПОСЛЕДНИЙ) \ ВОЗВРАЩАЕТ
 4 \ УКАЗАТЕЛЬ НА  ПОСЛЕДНИЙ
 5 \ ЭЛЕМЕНТ СПИСКА С УКАЗАТЕЛЕМ @СПИСОК
 6 : ПОСЛЕДНИЙ  DUP ХВОСТ НОЛЬ NOT
 7    IF ХВОСТ РЕКУРСИЯ
 8    THEN
 9 ;
10 \ ИТЕРАТИВНЫЙ ВАРИАНТ СЛОВА ПОСЛЕДНИЙ
11 : ПОСЛЕДНИЙ
12    BEGIN DUP ХВОСТ НОЛЬ NOT
13    WHILE ХВОСТ
14    REPEAT
15 ;
```

```
Экр # 48        C:PROLOG.BLK
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2   ( @СПИСОК -> N) \ ВОЗВРАЩАЕТ ЧИСЛО
 3                \ ЭЛЕМЕНТОВ В СПИСКЕ
 4 : ДЛИНА DUP НОЛЬ
 5    IF DROP 0
 6    ELSE ХВОСТ РЕКУРСИЯ 1+
 7    THEN ;
 8
 9 UNNEST
10 \ ДЛИНА. ИТЕРАТИВНЫЙ ВАРИАНТ
11 : ДЛИНА 0
12    BEGIN OVER НОЛЬ NOT
13    WHILE 1+ SWAP ХВОСТ SWAP
14    REPEAT NIP
15 ;
```

```
Экр # 49        C:PROLOG.BLK
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2  ( ЭЛЕМЕНТ @СПИСОК -> @ХВОСТ) \ ЕСЛИ ЭЛЕМЕНТ В СПИСКЕ
 3 \ ПРИСУТСТВУЕТ, ТО
 4 \  @ХВОСТ БУДЕТ ОСТАТКОМ СПИСКА, НАЧИНАЮЩЕГОСЯ С
 5 \  ЭЛЕМЕНТА;
 6 \  ИНАЧЕ НУЛЬ
 7  : ЧЛ SWAP OVER НОЛЬ
 8    IF 2DROP НУЛЬ
 9    ELSE OVER ПЕРВЫЙ OVER =
10       IF DROP
11       ELSE SWAP ХВОСТ РЕКУРСИЯ
12       THEN
13    THEN
14 ;
15
```

```
Экр # 50       C:PROLOG.BLK
 0 \ ОТДЕЛЬНЫЕ СПИСКИ
 1
 2 20 НОВСПИСОК I1
 3 20 НОВСПИСОК I2
 4 20 НОВСПИСОК I3
 5
 6 VARIABLE S1
 7 VARIABLE S2
 8 VARIABLE S3
 9
10 I1 ЧТСП (S1 S2 S3 )
11 I2 ЧТСП (S1 I1 @ S2 S3 )
12 I3 ЧТСП (S3 )
13
14
15
```

```
Экр # 51       C:PROLOG.BLK
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2  ( С СП1 -> СП2)
 3 : АССОЦ DUP НОЛЬ
 4   IF NIP
 5   ELSE 2DUP ПЕРВЫЙ ПЕРВЫЙ =
 6      IF ПЕРВЫЙ NIP
 7      ELSE ХВОСТ РЕКУРСИЯ
 8      THEN
 9   THEN
10 ;
11
12
13
14
15
```

```
Экр # 52       C:PROLLOG.BLK
 0 \ ЛИСПОПОДОБНЫЕ СЛОВА ПОСТРОЕНИЯ СПИСКОВ
 1 \ НА ФОРТЕ-83
 2  ( СП1 N -> СП2)
 3 : NНЫЙ DUP 1 <
 4   IF 2DROP НУЛЬ
 5   ELSE
 6      BEGIN 1- DUP
 7      WHILE SWAP ХВОСТ SWAP
 8      REPEAT DROP
 9   THEN
10 ;
11
12
13
14
15
```

```
Экр # 53       C:PROLOG.BLK
 0 \ УНИФИКАЦИЯ ПРОЛОГА НА ФОРТЕ-83
 1
 2 ( C1 C2 -> F )
 3 : РАВНО 2DUP =
 4   IF 2DROP TRUE
 5   ELSE 2DUP ATOM? SWAP ATOM? OR
 6      IF 2DROP FALSE
 7      ELSE 2DUP ПЕРВЫЙ SWAP ПЕРВЫЙ SWAP
 8         РЕКУРСИЯ -ROT ХВОСТ SWAP ХВОСТ SWAP
 9         РЕКУРСИЯ AND
10      THEN
11   THEN
12 ;
13
14
15
```

```
Экр # 54       C:PROLOG.BLK
 0 \ ПРИМЕР РЕКУРСИИ: ВЫЧИСЛЕНИЕ ФАКТОРИАЛА
 1
 2  ( N M -> )
 3 : *ФАКТОРИАЛ OVER 0=
 4   IF NIP
 5   ELSE OVER * SWAP 1- SWAP РЕКУРСИЯ
 6   THEN
 7 ;
 8
 9  ( M -> M!)
10 : ФАКТОРИАЛ 1 *ФАКТОРИАЛ ;
11
12
13
14
15
```

```
Экр # 57       C:PROLOG.BLK
 0 \ ИНТЕРПРЕТАТОР ПРАВИЛ ПРОЛОГА
 1
 2 ( -> ФЛАГ ИСТИНА | ЛОЖЬ)
 3 \  ФЛАГ = ИСТИНА => УСПЕХ
 4 : ХПОИСК
 5   ЦЕЛИ @ НОЛЬ NOT
 6
 7   IF ПОЛУЧИТЬ-ЦЕЛЬ @ НАЙТИ-ПРЕДЛОЖЕНИЕ?
 8      IF ДОБАВИТЬ-ЦЕЛИ FALSE
 9      ELSE ВОЗВРАТ
10         IF FALSE ELSE FALSE TRUE THEN
11      THEN
12   ELSE TRUE DUP
13   THEN
14 ;
15
```

```
Экр # 60        C:PROLOG.BLK
 0 \ ИНТЕРПРЕТАТОР ПРАВИЛ ПРОЛОГА
 1
 2 20  НОВСПИСОК ЦЕЛИ
 3 20  НОВСПИСОК РЕШЕНИЯ
 4 200 НОВСПИСОК ПРЕДЛОЖЕНИЯ
 5  ( -> ЦЕЛЬ)
 6 : ПОЛУЧИТЬ-ЦЕЛЬ ЦЕЛИ @ DUP ПЕРВЫЙ SWAP
 7 ХВОСТ ЦЕЛИ  SWAP УСТАНОВИТЬ ;
 8 ( ЦЕЛЬ @ПРЕДЛОЖЕНИЯ1 -> ЦЕЛЬ
 9   @ПРЕДЛОЖЕНИЯ2)
10 : НАЙТИ-ПРЕДЛОЖЕНИЕ
11    BEGIN 2DUP ПЕРВЫЙ ПЕРВЫЙ DUP >R = R>
12       НОЛЬ OR NOT
13    WHILE ХВОСТ
14    REPEAT
15 ;
```

```
Экр # 61        C:PROLOG.BLK
 0 \ ИНТЕРПРЕТАТОР ПРАВИЛ ПРОЛОГА
 1  ( ЦЕЛЬ @ ПРЕДЛОЖЕНИЕ -> ФЛАГ)
 2 \ ПОИСК  ПОДХ.ПРЕДЛОЖЕНИЯ И ПОМЕЩ. В РЕШ.
 3 \ ФЛАГ ИСТИНЕН, ЕСЛИ ПРЕДЛОЖЕНИЕ НАЙДЕНО
 4 : НАЙТИ-ПРЕДЛОЖЕНИЕ? НАЙТИ-ПРЕДЛОЖЕНИЕ
 5   DUP НОЛЬ  DUP >R
 6     IF 2DROP
 7     ELSE РЕШЕНИЯ СВЯЗЬ РЕШЕНИЯ СВЯЗЬ
 8     THEN R> NOT ;
 9 : ДОБАВИТЬ-ЦЕЛИ РЕШЕНИЯ @ ХВОСТ ПЕРВЫЙ
10   ПЕРВЫЙ ХВОСТ  ЦЕЛИ 2СОЕД ;  ( -> ФЛАГ)
11 \ ФЛАГ = 'ЛОЖЬ', ЕСЛИ СПИСОК РЕШЕНИЯ ПУСТ
12 : ВОЗВРАТ РЕШЕНИЯ @ DUP ПЕРВЫЙ SWAP ХВОСТ
13 ПЕРВЫЙ  ХВОСТ РЕШЕНИЯ DUP @ ХВОСТ ХВОСТ
14 УСТАНОВИТЬ НАЙТИ-ПРЕДЛОЖЕНИЕ? DUP
15 IF ДОБАВИТЬ-ЦЕЛИ THEN ;
```

```
Экр # 62        C:PROLOG.BLK
 0 \ ИНТЕРПРЕТАТОР ПРАВИЛ ПРОЛОГА
 1  ( -> ФЛАГ ИСТИНА | ЛОЖЬ)
 2 \ ФЛАГ = 'ИСТИНА' =>  УСПЕШНОЕ ЗАВЕРШЕНИЕ
 3 : (ПОИСК)
 4   ЦЕЛИ @ НОЛЬ NOT
 5   IF ПОЛУЧИТЬ-ЦЕЛЬ ПРЕДЛОЖЕНИЯ
 6              @ НАЙТИ-ПРЕДЛОЖЕНИЕ?
 7      IF ДОБАВИТЬ-ЦЕЛИ FALSE
 8      ELSE ОТКАТ
 9        IF FALSE ELSE FALSE TRUE THEN
10      THEN
11   ELSE TRUE DUP
12   THEN
13 ;
14
15
```



```
Экр # 63        C:PROLOG.BLK
 0 \ ИНТЕРПРЕТАТОР ПРАВИЛ ПРОЛОГА
 1  ( -> ФЛАГ)
 2 \ ФЛАГ = 'ИСТИНА' => УСПЕХ
 3 : ПОИСК
 4   BEGIN (ПОИСК)
 5    UNTIL
 6 ;
 7 DEFER ВЫВОД ' ПОИСК IS ВЫВОД
 8 : ?- РЕШЕНИЯ НУЛЬ УСТАНОВИТЬ ЦЕЛИ DUP НУЛЬ
 9        УСТАНОВИТЬ
10 ЧТСП ВЫВОД CR
11    ['] ВЫВОД >BODY @ ['] ПОИСК = \ УСТАНОВЛЕН
12   ЛИ ВЫВОД НА ПОИСК ?
13   IF
14      IF ." УСПЕХ" ELSE ." НЕУДАЧА" THEN
15   THEN  ;
```

```
Экр # 64        C:PROLOG.BLK
 0 \ ИНТЕРПРЕТАТОР ПРАВИЛ ПРОЛОГА
 1
 2 \ МАКСИМУМ ЦЕЛЕЙ В ПРАВИЛЕ = 4
 3 : ПРАВИЛО: CREATE HERE DUP 2+ ,
 4 НУЛЬ , 10 ALLOT ЧТСП ;
 5
 6 : .КЛ ПРЕДЛОЖЕНИЯ @ ВЫДАТЬ ;
 7 : .ЦЕЛИ ЦЕЛИ @ ВЫДАТЬ
 8 : КАК? РЕШЕНИЯ @ DUP НОЛЬ
 9   IF ВЫДАТЬ
10   ELSE
11      BEGIN DUP НОЛЬ NOT
12      WHILE DUP ХВОСТ ПЕРВЫЙ ПЕРВЫЙ ВЫДАТЬ
13          ХВОСТ ХВОСТ
14      REPEAT DROP
15   THEN ;
```

```
Экр # 65        C:PROLOG.BLK
 0 \ ИНТЕРПРЕТАТОР ПРАВИЛ ПРОЛОГА
 1 : ДОСТАТОЧНО? KEY?
 2         IF KEY DROP KEY 13 =
 3           ELSE FALSE
 4         THEN
 5 ;
 6
 7 \ СЛЕДЫ ПОИСКА
 8 : СЛЕД
 9   BEGIN CR ." ЦЕЛИ:" ЦЕЛИ @ ВЫДАТЬ
10       CR ." РЕШЕНИЯ:" КАК? CR (ПОИСК) DUP >R
11       IF
12         IF ." УСПЕХ" ELSE ." НЕУДАЧА" CR THEN
13       THEN R> ДОСТАТОЧНО? OR
14   UNTIL
15 ;
```

```
Экр # 66         C:PROLOG.BLK
 0 \ АЛГОРИТМ УНИФИКАЦИИ
 1
 2
 3
 4 \ ПЕРСП ОПРЕДЕЛЯЕТ ЛОГИЧЕСКИЕ ПЕРЕМЕННЫЕ
 5  ( -> PFA+2)
 6 : ПЕРСП CREATE HERE 2+ , 0 , DOES>
 7     2+ ;
 8
 9
10 ПЕРСП FOO ' FOO @ FORGET FOO
11
12
13   ( C -> F) \ F ИСТИННО, ЕСЛИ  ПЕРСП
14 : ПЕР? DUP @ 0= SWAP 4 - @ [ DUP ] LITERAL = AND
15 DROP
```

```
Экр # 67         C:PROLOG.BLK
 0 \ АЛГОРИТМ УНИФИКАЦИИ
 1  ( C СП1 -> СП2)
 2 \ 2АССОЦ-СПИСОК СОДЕРЖИТ ПАРЫ ПЕРСПИС- ЗНАЧЕНИЕ
 3 : 2АССОЦ DUP НОЛЬ
 4   IF NIP
 5   ELSE 2DUP ПЕРВЫЙ =
 6      IF NIP
 7      ELSE ХВОСТ ХВОСТ РЕКУРСИЯ
 8      THEN
 9   THEN
10 ;
11
12
13
14
15
```

```
Экр # 68         C:PROLOG.BLK
 0 \ АЛГОРИТМ УНИФИКАЦИИ
 1
 2  ( C1 @O -> СП2)
 3 : ЗНАЧЕНИЕ OVER ПЕР?
 4   IF 2DUP 2АССОЦ DUP НОЛЬ
 5      IF 2DROP
 6      ELSE ROT DROP ХВОСТ ПЕРВЫЙ SWAP РЕКУРСИЯ
 7      THEN
 8   ELSE DROP
 9   THEN
10 ;
11
12
13
14
15
```



```
Экр # 69         C:PROLOG.BLK
 0 \ АЛГОРИТМ УНИФИКАЦИИ
 1  ( C1 C2 O -> F)
 2 \ F = 'ИСТИНА',ЕСЛИ C1 и C2 УНИФИЦИРУЕМЫ; СВЯЗКИ В О
 3 : УНИФИКАЦИЯ DUP >R @ ROT OVER ЗНАЧЕНИЕ
 4 -ROT ЗНАЧЕНИЕ OVER ПЕР?
 5   IF НУЛЬ -ROT R> СПИСОК TRUE  ELSE DUP ПЕР?
 6      IF SWAP НУЛЬ -ROT R> СПИСОК TRUE
 7      ELSE 2DUP ATOM? SWAP ATOM? OR NOT
 8         IF 2DUP ПЕРВЫЙ SWAP ПЕРВЫЙ SWAP R@ РЕКУРСИЯ
 9            IF ХВОСТ SWAP ХВОСТ SWAP R> РЕКУРСИЯ
10             ELSE 2DROP R> DROP FALSE
11            THEN
12         ELSE R> DROP =
13         THEN
14      THEN
15   THEN ;
```

```
Экр # 70         C:PROLOG.BLK
 0 \ БАЗА ЗНАНИЙ
 1
 2 : МАРКЕР ;
 3 ПРАВИЛО: ДАЕТ-МОЛОКО    (ДАЕТ-МОЛОКО )
 4 ПРАВИЛО: ИМЕЕТ-ВОЛОС-ПОКРОВ  (ИМЕЕТ-ВОЛОС-ПОКРОВ )
 5 ПРАВИЛО: ИММЕТ-РОГА     (ИМЕЕТ-РОГА )
 6 ПРАВИЛО: МЛЕКОПИТАЮЩЕЕ (МЛЕКОПИТАЮЩЕЕ ДАЕТ-МОЛОКО
 7 ИМЕЕТ-ВОЛОС-ПОКРОВ)
 8 ПРАВИЛО: КОЗЕЛ1       (КОЗЕЛ БОРЬКА )
 9 ПРАВИЛО: КОЗЕЛ2       (КОЗЕЛ МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-РОГА )
10 ПРЕДЛОЖЕНИЯ НУЛЬ УСТАНОВИТЬ ПРЕДЛОЖЕНИЯ ЧТСП
11 (ДАЕТ-МОЛОКО @ ИМЕЕТ-ВОЛОС-ПОКРОВ @ ИМЕЕТ-РОГА
12 @ МЛЕКОПИТАЮЩЕЕ @ КОЗЕЛ1  @ КОЗЕЛ2 @ )
13 РЕШЕНИЯ НУЛЬ УСТАНОВИТЬ
14 ЦЕЛИ НУЛЬ УСТАНОВИТЬ
15 НУЛЬ КОЗЕЛ ЦЕЛИ СПИСОК
```

```
Экр # 71         C:PROLOG.BLK
 0 \ БАЗА ЗНАНИЙ
 1 : МАРКЕР ; ПРАВИЛО: ДАЕТ-МОЛОКО   (ДАЕТ-МОЛОКО )
 2 ПРАВИЛО: ИМЕЕТ-ВОЛОС-ПОКРОВ1
 3 (ИМЕЕТ-ВОЛОС-ПОКРОВ ГРУБИЯН )
 4 ПРАВИЛО: ИМЕЕТ-ВОЛОС-ПОКРОВ2 (ИМЕЕТ-ВОЛОС-ПОКРОВ )
 5 ПРАВИЛО: ИМЕЕТ-РОГА (ИМЕЕТ-РОГА )
 6 ПРАВИЛО: МЛЕКОПИТАЮЩЕЕ
 7 (МЛЕКОПИТАЮЩЕЕ ДАЕТ-МОЛОКО ИМЕЕТ-ВОЛОС-ПОКРОВ)
 8 ПРАВИЛО: КОЗЕЛ1       (КОЗЕЛ БОРЬКА )
 9 ПРАВИЛО: КОЗЕЛ2       (КОЗЕЛ ГРУБИЯН )
10 ПРАВИЛО: КОЗЕЛ3       (КОЗЕЛ МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-РОГА )
11 ПРЕДЛОЖЕНИЯ НУЛЬ УСТАНОВИТЬ ПРЕДЛОЖЕНИЯ ЧТСП
12 (ДАЕТ-МОЛОКО @ ИМЕЕТ-ВОЛОС-ПОКРОВ1 @ ИМЕЕТ-ВОЛОС
13 ПОКРОВ2 @  ИМЕЕТ-РОГА @ МЛЕКОПИТАЮЩЕЕ @ КОЗЕЛ1 @
14 КОЗЕЛ2 @ КОЗЕЛ3 @ ) РЕШЕНИЯ НУЛЬ УСТАНОВИТЬ ЦЕЛИ НУЛЬ 15
УСТАНОВИТЬ НУЛЬ КОЗЕЛ ЦЕЛИ СПИСОК
```

```
Экр # 72         C:PROLOG.BLK
 0 \ ПЕРЕМЕННЫЕ-СПИСКИ И СПИСОК ENV
 1
 2 ПЕРСП X1
 3 ПЕРСП X2
 4 ПЕРСП X3
 5
 6 VARIABLE A1
 7 VARIABLE A2
 8 VARIABLE A3
 9 20 НОВСПИСОК ENV
10 20 НОВСПИСОК СП1 НУЛЬ A2 X3 СП1 СПИСОК
11 20 НОВСПИСОК СП2 НУЛЬ A1 X3 СП1 @ СП2 СПИСОК
12 20 НОВСПИСОК СП3 НУЛЬ A1 X2 СП3 СПИСОК
13
14
15
```

```
Экр # 73         C:PROLOG.BLK
 0 \ ВЫВОД РАСШИРЕННОГО ПЕРСП
 1 ( @ -> ФЛАГ)
 2 \ ФЛАГ=ИСТИНЕ, ЕСЛИ @ ЯВЛЯЕТСЯ PFA ПЕРЕМЕННОЙ
 3 : ATOM? BODY> @ ['] НУЛЬ @ = ;
 4
 5 ( @СПИСОК -> )
 6 : ВЫДАТЬСП CR ." ("
 7   BEGIN DUP ПЕРВЫЙ DUP ATOM?
 8     IF DUP НОЛЬ
 9       IF DROP ELSE BODY> >NAME .ID THEN
10     ELSE DUP ПЕР?
11       IF 2- BODY> >NAME .ID ELSE РЕКУРСИЯ THEN
12     THEN ХВОСТ DUP НОЛЬ
13   UNTIL 8 ( ЗАБОЙ) EMIT ." ) " DROP
14 ;
15
```

```
Экр # 74         C:PROLOG.BLK
 0 \ ВЫВОД РАСШИРЕННОГО ПЕРСП
 1
 2 ( @СПИСОК -> )
 3 : ВЫДАТЬ DUP @ НОЛЬ
 4   IF DROP CR ." НУЛЬ"
 5   ELSE DUP ATOM?
 6     IF BODY> >NAME .ID
 7     ELSE DUP ПЕР?
 8       IF 2- BODY> >NAME CR ." ПЕРСП " .ID
 9       ELSE ВЫДАТЬСП
10       THEN
11     THEN
12   THEN
13 ;
14
15
```



```
Экр # 90         C:PROLOG.BLK
 0 \ АЛГОРИТМЫ РАСПОЗНАВАНИЯ ОБРАЗОВ
 1 \ КЛАСС - ОПРЕДЕЛЯЮЩЕЕ КЛАССЫ СЛОВО;
 2 \ СОДЕРЖИТ: # ТОЧЕК В КЛАССЕ;
 3 \ ТОЧКУ-ПРОТОТИП; W(N+1); ТОЧКИ КЛАСТЕРА (X, Y)
 4
 5 ( N -> ) \ # ОБУЧАЮЩИХ ТОЧЕК В КЛАССЕ
 6 : КЛАСС CREATE DUP , 6 ALLOT 4 * ALLOT
 7 ( N -> @) \ @ УКАЗЫВАЕТ НА ТОЧКУ (X, Y)
 8   DOES> DUP -ROT @ MIN 4 * + 8 +
 9 ;
10
11 \ СКАЛЯРНОЕ ПРОИЗВЕДЕНИЕ ВЕКТОРОВ
12 ( @V1 @V2 -> N)
13 : ТОЧКА 2DUP @ SWAP @ * -ROT 2+ @ SWAP 2+ @ * + ;
14
15
```

```
Экр # 91         C:PROLOG.BLK
 0 \ АЛГОРИТМЫ РАСПОЗНАВАНИЯ ОБРАЗОВ
 1 ( 'КЛАСС -> )
 2 \ УСРЕДНЕННЫЙ ВЕКТОР - 'КЛАСС ЭТО CFA КЛАССА
 3 : ПРОТОТИП DUP >BODY DUP 2+ 6 0 FILL @ DUP >R 0
 4   DO DUP >BODY 2+ OVER I SWAP EXECUTE
 5     2DUP @ SWAP +! 2+ @ SWAP 2+ +!
 6   LOOP >BODY 2+ DUP DUP @ R@ / OVER ! 2+ DUP @ R> / SWAP ;
 7   DUP 4 + SWAP DUP ТОЧКА 2/ SWAP ! \ ВЫЧИСЛЕНИЕ W(N+1)
 8 ;
 9 \ РЕШАЮЩАЯ ФУНКЦИЯ ПО ПРИНЦИПУ МИНИМАЛЬНОГО
10 \ РАССТОЯНИЯ
11 ( 'КЛАСС @X -> N)
12 : D(X) SWAP >BODY 2+ DUP 4 + @ -ROT ТОЧКА SWAP - ;
13
14
15
```

```
Экр # 92         C:PROLOG.BLK
 0 \ АЛГОРИТМЫ РАСПОЗНАВАНИЯ ОБРАЗОВ
 1
 2 ( N -> )
 3 : КЛАССЫ CREATE DUP , 2* ALLOT
 4 ( N -> 'КЛАССN)
 5   DOES> DUP @ ROT MIN 2* + 2+ @
 6 ;
 7
 8 ( @X 'КЛАССЫ -> N) КЛАССИФИКАЦИЯ ТОЧКИ (X, Y) В @X
 9 : КЛАССИФИКАЦИЯ DUP 0 0 ROT >BODY @ 0
10   DO I 3 PICK EXECUTE 4 PICK D(X) DUP 3 PICK >
11     IF -ROT 2DROP I ELSE DROP THEN
12   LOOP NIP NIP NIP
13 ;
14
15
```

```
Экр # 93        C:PROLOG.BLK
 0 \ АЛГОРИТМЫ РАСПОЗНАВАНИЯ ОБРАЗОВ
 1
 2  ( X1 Y1 Y2 ... XN YN N 'КЛАСС -> )
 3 : >КЛАСС >BODY DUP >R 8 + DUP ROT DUP R> ! 4 * + 2-
 4    DO I ! -2 +LOOP
 5 ;
 6
 7  ( 'КЛАСС1 'КЛАСС2 ... 'КЛАССN -> )
 8 : >КЛАССЫ ' >BODY DUP >R 2+ DUP ROT DUP R> ! 2* + 2-
 9    DO I ! -2 +LOOP
10 ;
11
12
13
14
15
```

```
Экр # 94        C:PROLOG.BLK
 0 \ АЛГОРИТМЫ РАСПОЗНАВАНИЯ ОБРАЗОВ
 1
 2 \ ПРИМЕР
 3
 4 4 КЛАСС КЛАСС1  0 2  1 1  2 1  2 2  4 '  КЛАСС1 >КЛАСС
 5
 6 4 КЛАСС КЛАСС2  5 3  4 1  4 2  6 2  4 ' КЛАСС2 >КЛАСС
 7
 8 4 КЛАСС КЛАСС3  6 6  5 5  8 9  7 7  4 '  КЛАСС3 >КЛАСС
 9
10 3 КЛАССЫ КЛАСС1
11 ' КЛАСС1 ' КЛАСС2 ' КЛАСС3 3 >КЛАССЫ КЛАССЫ1
12
13 VARIABLE X 2 ALLOT
14
15
```

```
Экр # 95        C:PROLOG.BLK
 0 \ АЛГОРИТМЫ РАСПОЗНАВАНИЯ ОБРАЗОВ
 1
 2 ' КЛАСС1 ПРОТОТИП
 3 ' КЛАСС2 ПРОТОТИП
 4 ' КЛАСС3 ПРОТОТИП
 5
 6 4 X ! 4 X 2+ ! \ X = (4, 4)
 7
 8 X ' КЛАССЫ1 КЛАССИФИКАЦИЯ
 9
10
11
12
13
14
15
```

Приложение Б

ПОДПРОГРАММЫ ДИАГНОСТИКИ

Добавив экраны, приведенные в настоящем разделе, к описанному в приложении А прологоподобному расширению Форта, вы получите систему, которая выполняет интерпретацию правил, ведет диалог с пользователем и поддерживает функционирование рабочей памяти.

Реализация новых подпрограмм дополнительно потребовала введения следующих новых списков:

1. ГИПОТЕЗЫ. Сюда заносятся гипотезы, подлежащие проверке системой. Среди них могут оказаться и конечные заключения (см. экран 80).

2. ЗАКЛЮЧЕНИЯ. Короткий список, содержащий конечные заключения.

3. ФАКТЫ. Этот список используется как рабочая память или база данных для истинных утверждений.

В приводимом расширении "главным" является слово ДИАГНОЗ. Оно пытается доказать гипотезы, поочередно помещая каждую в список ЦЕЛИ и выполняя ее сопоставление с имеющимися предложениямн. Доказательство проводится обратным методом, реализованным словом (ПОИСК). Если гипотеза оказывается верной, процедура ДИАГНОЗ завершает свою работу и выдает конечное заключение.

Здесь мы имеем дело с модифицированным вариантом слова (ПОИСК), допускающим диалог с пользователем и применение рабочей памяти.

Каждую цель процедура (ПОИСК) в первую очередь старается найти в списке ФАКТЫ (как уже доказанную) с помощью слова НАЙТИ-ФАКТ?, которое представляет собой несколько модифицированный алгоритм слова НАЙТИ-ПРЕДЛОЖЕНИЕ? . Если цель в списке ФАКТЫ не обнаружена, подпрограмма (ПОИСК) пытается подыскать правило, используя слово НАЙТИ-ПРЕДЛОЖЕНИЕ? . Если и этот поиск завершается неудачей, пользователю выдается запрос на продолжение верификации. При получении на запрос ответа "Y" ("Да") цель добавляется к списку ФАКТЫ как истинное утверждение. В том случае, когда выдается ответ "N" ("Нет"), программа (ПОИСК) возвращается по списку РЕШЕНИЯ на один уровень выше текущего и продолжает функционирование.

Итак, в результате внесенных изменений имеем:

(ПОИСК) модифицировано
СЛЕД модифицировано
ПОИСК модифицировано
ПОЛУЧ-ГИП получение гипотезы (аналогично слову ПОЛУЧИТЬ-ЦЕЛЬ)
НАЙТИ-ФАКТ? попытка обнаружения факта в списке ФАКТЫ аналогично

слову НАЙТИ-ПРЕДЛОЖЕНИЕ?

ДИАГНОЗ вновь определенное слово высокого уровня

ДИАЛОГ получает ответ пользователя

Примечание. Проще добавлять доказанные факты к предложениям (а не к списку **ФАКТЫ**), но в нашей системе это невозможно. **ПРЕДЛОЖЕНИЯ** - это список списков, и вы не можете внести цель в данный список без создания для нее отдельного списка с последующим его присоединением к списку предложений.

Наша система не позволяет создавать списки в динамике. Вы всегда можете определить с помощью команды **ВЫДАТЬ**, является ли некий объект атомом или списком. Если выданный объект заключен в круглые скобки, он является списком, в противном случае - атомом. Применяя выражение **"ПОЛУЧИТЬ-ЦЕЛЬ ВЫДАТЬ"**, мы получаем атом - верхний элемент списка **ЦЕЛИ**. С помощью же выражения **"ПРЕДЛОЖЕНИЯ @ ВЫДАТЬ"** мы получаем список.

Поскольку наша система диагностики считается базовой, может возникнуть потребность расширить ее следующим образом:

1. Добавить слово, которое при включении факта в список **ФАКТЫ** выдавало бы след проведенного рассуждения для проверки, появились ли вследствие добавления этого факта новые доказанные заключения, и если они появились, присоединяло бы их к списку **ФАКТЫ**. Например, если доказано утверждение **ИМЕЕТ-ОПЕРЕНИЕ**, то заключение **ЭТО-ПТИЦА** должно быть внесено в список фактов, причем все вхождения выражения **"ФАКТЫ СВЯЗЬ"** в программу **(ПОИСК)** должны быть заменены вводимым словом.
2. Добавить новый список **ФАКТЫ**, в котором запоминались бы неистинные факты. В прежней реализации системы в список **ФАКТЫ** заносились истинные факты, что не давало возможности задавать один и тот же вопрос при проверке следующей гипотезы. Если же факт признавался ложным, он не сохранялся и вопрос повторно задавался во время обоснования других гипотез. Требуется модификация программы **(ПОИСК)** с тем, чтобы она проверяла и новый список.

Можно изменить наполнение базы знаний, приведенной на экранах с 80 по 83. Например, вы можете реализовать базу знаний для диагностики неисправности автомобиля. Каковы при этом конечные гипотезы? Какие симптомы неисправностей нужно учитывать? Что выбрать в качестве промежуточных целей?

Особенности работы с Фортом

Загрузите эти экраны по аналогии с загрузкой экспертной системы на Прологе, описанной в приложении А, руководствуясь описанием вашей конкретной системы.

Если вы пользуетесь Фортом Ф83 Лэксена и Перри (поставляемым как часть программного обеспечения на дискетах), выполните следующие действия:

1. Загрузите базовую систему согласно инструкции, приведенной в приложении А.
2. Загрузите новые экраны:

```
75 83 THRU
```


3. Переключите след:

```
' СЛЕД IS ВЫВОД
```

4. Теперь вы можете пользоваться системой, запуская ее на выполнение словом ДИАГНОЗ. Чтобы повторно войти в систему с помощью слова ДИАГНОЗ, вы должны снова загрузить базу знаний, введя выражение 80 83 THRU, и не забыть переназначить СЛЕД.

Вы можете запомнить состояние системы после загрузки с помощью выражения "SAVE-SYSTEM ANIMAL.COM". Убедитесь,что у вас для сохраняемого варианта имеется доствточный объем памяти (т.е. не менее 32К), иначе произойдет разрушение системы Форт. Если же вариант хранится в памяти, для запуска системы вам достаточно набрать в среде DOS слово ЖИВОТНОЕ и не перечислять номера экранов. Не забудьте перед сохранением варианта переключить вывод (' СЛЕД IS ВЫВОД).

```
Экр # 75          C:PROLOG.BLK
 0 \ ПРОЛОГ. ПОДПРОГРАММЫ ДИАГНОСТИКИ
 1 : ДИАЛОГ CR ." Это соответствует истине (Y/N)?
 2   KEY DUP EMIT CR ASCII Y =
 3 ;
 4
 5 : НАЗАД
 6     ВОЗВРАТ
 7     IF FALSE ELSE FALSE TRUE THEN
 8 ;
 9 20 НОВСПИСОК ГИПОТЕЗЫ
10 20 НОВСПИСОК ФАКТЫ
11 20 НОВСПИСОК ЗАКЛЮЧЕНИЯ
12 VARIABLE ФЛГКОН
13
14 : ПОЛУЧ-ГИП ГИПОТЕЗЫ @ DUP ПЕРВЫЙ
15       SWAP ХВОСТ ГИПОТЕЗЫ SWAP УСТАНОВИТЬ ;
```

```
Экр # 76          C:PROLOG.BLK
 0 \ ПРОЛОГ. ПОДПРОГРАММЫ ДИАГНОСТИКИ
 1 ( ЦЕЛИ @ФАКТЫ1 -> ЦЕЛИ @ФАКТЫ2 )
 2 : НАЙТИ-ФАКТ
 3   BEGIN 2DUP ПЕРВЫЙ DUP >R = R> НОЛЬ OR NOT
 4     WHILE ХВОСТ
 5     REPEAT
 6 ;
 7
 8 ( -> ФЛАГ ) \ ФЛАГ ИСТИНЕН, ЕСЛИ ФАКТ НАЙДЕН
 9 : НАЙТИ-ФАКТ? НАЙТИ-ФАКТ DUP НОЛЬ DUP >R
10     IF 2DROP
11     ELSE R> РЕШЕНИЯ СВЯЗЬ РЕШЕНИЯ СВЯЗЬ
12     THEN R> NOT
13 ;
14
15
```

```
Экр # 77          C:PROLOG.BLK
 0 \ ПРОЛОГ. ПОДПРОГРАММЫ ДИАГНОСТИКИ
 1 \ ПРОЛОГ. ИНТЕРПРЕТАТОР ПРАВИЛ
 2 ( -> ФЛАГ ИСТИНА | ЛОЖЬ )
 3 \ ФЛАГ = 'ИСТИНА' => УСПЕШН.ЗАВЕРШ.
 4 :(ПОИСК) ЦЕЛИ @ НОЛЬ NOT
 5   IF ПОЛУЧИТЬ-ЦЕЛЬ DUP ФАКТЫ @ НАЙТИ-ФАКТ?
 6   IF DROP FALSE ELSE DUP
 7    ПРЕДЛОЖЕНИЯ @ НАЙТИ-ПРЕДЛОЖЕНИЕ?
 8         IF ДОБАВИТЬ-ЦЕЛИ DROP FALSE
 9         ELSE DUP CR ВЫДАТЬ ДИАЛОГ
10            IF ФАКТЫ СВЯЗЬ FALSE
11            ELSE DROP НАЗАД
12            THEN
13          THEN
14       THEN
15   ELSE TRUE DUP  THEN ;
```



```
Экр # 78          C:PROLOG.BLK
 0 \ ПРОЛОГ. ПОДПРОГРАММЫ ДИАГНОСТИКИ
 1 : ПОИСК
 2   BEGIN
 3     (ПОИСК) DUP >R
 4     IF IF TRUE ФЛГКОН ! ELSE FALSE ФЛГКОН ! THEN
 5     THEN R>
 6   UNTIL ;
 7 \ СЛЕД ПОИСКА
 8 : СЛЕД
 9   BEGIN CR ." ЦЕЛИ:" ЦЕЛИ @ ВЫДАТЬ
10       CR ." РЕШЕНИЯ:" КАК? CR (ПОИСК) DUP >R
11       IF
12         IF TRUE ФЛГКОН ! ELSE FALSE ФЛГКОН ! THEN
13       THEN R> ДОСТАТОЧНО? OR
14   UNTIL
15 ;
```

```
Экр # 79          C:PROLOG.BLK
 0 \ ПРОЛОГ. ПОДПРОГРАММЫ ДИАГНОСТИКИ
 1 : ДИАГНОЗ РЕШЕНИЯ НУЛЬ УСТАНОВИТЬ ФАКТЫ НУЛЬ
 2   УСТАНОВИТЬ
 3   BEGIN ЗАКЛЮЧЕНИЯ НУЛЬ УСТАНОВИТЬ ПОЛУЧ-ГИП DUP
 4   НОЛЬ NOT
 5     IF DUP ЦЕЛИ НУЛЬ УСТАНОВИТЬ НУЛЬ SWAP ЦЕЛИ СПИСОК
 6     НУЛЬ SWAP ЗАКЛЮЧЕНИЯ СПИСОК ВЫВОД
 7     ФЛГКОН @
 8     THEN
 9   UNTIL
10   CR ." ЗАКЛЮЧЕНИЕ:" ЗАКЛЮЧЕНИЯ @ ВЫДАТЬ CR ;
11 \ ПРИМЕР КЛАССИФИКАЦИИ ЖИВОТНЫХ
12 : МАРКЕР ;
13 ГИПОТЕЗЫ НУЛЬ УСТАНОВИТЬ ГИПОТЕЗЫ ЧТСП
14 (ЭТО-АЛЬБАТРОС ЭТО-ПИНГВИН ЭТО-СТРАУС ЭТО-ЗЕБРА
15 ЭТО-ЖИРАФ ЭТО-ТИГР ЭТО-ГЕПАРД )
```

```
Экр # 81          C:PROLOG.BLK
 0 \ ПРИМЕР КЛАССИФИКАЦИИ ЖИВОТНЫХ
 1 ПРАВИЛО: ЭТО-МЛЕКОПИТАЮЩЕЕ1? (ЭТО-МЛЕКОПИТАЮЩЕЕ
 2 ИМЕЕТ-ВОЛОС-ПОКРОВ) ПРАВИЛО: ЭТО-МЛЕКОПИТАЮЩЕЕ2?
 3 (ЭТО-МЛЕКОПИТАЮЩЕЕ ДАЕТ-МОЛОКО )
 4 ПРАВИЛО: ЭТО-ПТИЦА1? (ЭТО-ПТИЦА ИМЕЕТ ОПЕРЕНИЕ )
 5 ПРАВИЛО: ЭТО-ПТИЦА2? (ЭТО-ПТИЦА ЛЕТАЕТ ОТКЛ-ЯЙЦА )
 6 ПРАВИЛО: ЭТО-ПЛОТОЯДНОЕ1? (ЭТО-ПЛОТОЯДНОЕ
 7 ПИТАЕТСЯ-МЯСОМ )  ПРАВИЛО: ЭТО-ПЛОТОЯДНОЕ2?
 8 (ЭТО-ПЛОТОЯДНОЕ ИМЕЕТ-ОСТРЫЕ-ЗУБЫ
 9 ИМЕЕТ-КОГТИ ИМЕЕТ-СМОТРЯЩИЕ-ВПЕРЕД-ГЛАЗА )
 8 ПРАВИЛО: ЭТО-КОПЫТНОЕ1? (ЭТО-КОПЫТНОЕ
 9 ЭТО-МЛЕКОПИТАЮЩЕЕ ИМЕЕТ-КОПЫТА )
10 ПРАВИЛО: ЭТО-КОПЫТНОЕ2? (ЭТО-КОПЫТНОЕ
11 ЭТО-МЛЕКОПИТАЮЩЕЕ ЖВАЧНОЕ )  ПРАВИЛО: ЭТО-ГЕПАРД?
12 (ЭТО-ГЕПАРД ЭТО-МЛЕКОПИТАЮЩЕЕ  ЭТО-ПЛОТОЯДНОЕ
13 ИМЕЕТ-РЫЖЕ-КОРИЧ-ОКРАС ИМЕЕТ-ТЕМНЫЕ-ПЯТНА )
14
15
```

```
Экр # 82         C:PROLOG.BLK
 0 \ ПРИМЕР КЛАССИФИКАЦИИ ЖИВОТНЫХ
 1 ПРАВИЛО: ЭТО-ЗЕБРА? (ЭТО-ЗЕБРА ЭТО-КОПЫТНОЕ
 2 ИМЕЕТ-ЧЕРНЫЕ-ПОЛОСЫ )  ПРАВИЛО: ЭТО-СТРАУС?
 3 (ЭТО-СТРАУС ЭТО-ПТИЦА НЕ-ЛЕТАЕТ ИМЕЕТ-ДЛИННУЮ-ШЕЮ
 4 ИМЕЕТ-ДЛИННЫЕ-НОГИ ) ПРАВИЛО: ЭТО ПИНГВИН?
 5 (ЭТО-ПИНГВИН ЭТО-ПТИЦА НЕ-ЛЕТАЕТ ПЛАВАЕТ
 6 ИМЕЕТ-ЧЕРНО-БЕЛ-ОКРАС ) ПРАВИЛО. ЭТО-АЛЬБАТРОС?
 7 (ЭТО-АЛЬБАТРОС ЭТО-ПТИЦА ЛЕТАЕТ-ХОРОШО )
 8  ПРАВИЛО: ЭТО ТИГР? (ЭТО-ТИГР ЭТО-МЛЕКОПИТАЮЩЕЕ
 9      ЭТО-ПЛОТОЯДНОЕ
10      ИМЕЕТ-РЫЖЕ-КОРИЧ-ОКРАС ИМЕЕТ-ЧЕРНЫЕ-ПОЛОСЫ )
11  ПРАВИЛО: ЭТО-ЖИРАФ? (ЭТО-ЖИРАФ
12      ИМЕЕТ-ДЛИННУЮ-ШЕЮ
13      ИМЕЕТ-ДЛИННЫЕ-НОГИ ИМЕЕТ ТЕМНЫЕ-ПЯТНА )
14
15 ПРЕДЛОЖЕНИЯ НУЛЬ УСТАНОВИТЬ
```

```
Экр # 83         C:PROLOG.BLK
 0 \ ПРИМЕР КЛАССИФИКАЦИИ ЖИВОТНЫХ
 1 ПРЕДЛОЖЕНИЯ ЧТСП
 2 (ЭТО-ЖИВОТНОЕ1? @ ЭТО-ЖИВОТНОЕ2? @ ЭТО-ПТИЦА1? @
 3  ЭТО ПТИЦА2? @ ЭТО-ПЛОТОЯДНОЕ1? @ ЭТО-ПЛОТОЯДНОЕ2? @
 4  ЭТО-КОПЫТНОЕ1? @ ЭТО-КОПЫТНОЕ2? @ ЭТО-ГЕПАРД? @
 5  ЭТО-ТИГР? @ ЭТО-ЖИРАФ? @ ЭТО-ЗЕБРА? @
 6  ЭТО-СТРАУС? @ ЭТО-ПИНГВИН? @ ЭТО-АЛЬБАТРОС? @ )
 7
 8
 9
10
11
12
13 РЕШЕНИЯ НУЛЬ УСТАНОВИТЬ
14 ЦЕЛИ НУЛЬ УСТАНОВИТЬ
15 НУЛЬ ЭТО-АЛЬБАТРОС ЦЕЛИ СПИСОК
```

Приложение В

СЛОВАРЬ ФОРТА

!	(n a ->)	Занесение n по адресу a.
'	(-> cfa)	Занесение в вершину стека cfa следующего слова из входного потока
(	->)	Пропуск символов из входного потока до правой круглой скобки. Используется для записи комментария.
*	(n1 n2 -> n3)	Умножение n1 на n2 с получением произведения n3.
*/	(n1 n2 n3 -> n4)	Умножение n1 на n2 с получением результата двойной длины, который затем делится на n3. Используется при масштабировании чисел с фиксированной точкой.
*/MOD	(n1 n2 n3 -> r q)	То же, что и */, но в качестве результата в вершину стека заносит частное q и остаток r.
+	(n1 n2 -> n3)	Сложение n1 и n2, результатом является n3.
+!	(n a ->)	Добавление n к содержимому по адресу a.
+LOOP	(n ->)	Добавление n к счетчику цикла. Если в результате полученное значение счетчика выходит за границу, цикл принудительно завершается, в противном случае управление передается на фрагмент, расположенный за DO.
,	(n ->)	Компиляция n в словарь путем выделения памяти и занесения.
-	(n1 n2 -> n3)	Вычитание n2 из n1 с получением разности n3.
.	(n ->)	Вывод числа n.
."	(->)	Используется во время компиляции Компилирует строковый литерал, начинающийся с символа, следующего за пробелом после .", и завершающийся символом " . Скомпилированный текст выводится при исполнении слова ."

/	(n1 n2 -> n3)	Деление n1 на n2 с получением частного n3.
/MOD	(n1 n2 -> r q)	То же, что и / , но в качестве результата в вершину стека заносится частное q и остаток r
0<	(n -> f)	Флаг истинен, если n < 0.
0=	(n -> f)	Флаг истинен, если n = 0.
0>	(n -> f)	Флаг истинен, если n > 0
1+	(n1 -> n2)	n2 = n1 + 1.
1-	(n1 -> n2)	n2 = n1 - 1.
2+	(n1 -> n2)	n2 = n1 + 2
2-	(n1 -> n2)	n2 = n1 - 2 .
2/	(n1 -> n2)	n2 = n1/2 . Выполняется арифметическим сдвигом вправо на один бит.
2*	(n1 -> n2)	n2 = 2 * n1 . Выполняется сдвигом влево на один бит.
:	(->)	Определение слова Форта путем создания заголовка словарной статьи и компиляции текста из входного потока до символа ; .
;	(->)	Завершение определения через двоеточие и перевод Форта в режим интерпретации.
<	(n1 n2 -> f)	Флаг истинен, если n1 < n2
=	(n1 n2 -> f)	Флаг истинен, если n1 = n2.
>	(n1 n2 -> f)	Флаг истинен, если n1 > n2 >.
>BODY	(cfa -> pfa)	Переход от cfa к pfa, т.е. к телу словарной статьи.
>R	(n ->)	Занесение n в стек возвратов.
?DUP	(n1 -> [n1] n2)	Если n1 не ноль, выполнение операции DUP.
@	(a -> n)	Занесение в вершину стека содержимого по адресу a.
ABS	(n1 -> n2)	Занесение в вершину стека абсолютного значения n1.
ALLOT	(n1 ->)	Выделение n байт памяти в словаре
AND	(n1 n2 -> n3)	Выполнение поразрядной конъюнкции над n1 и n2 с получением n3.
BASE	(-> pfa)	Системная переменная, содержащая текущее основание системы счисления для вводимых чисел.

BEGIN	(->)	Начало цикла как со счетчиком (UNTIL), так и с условием (WHILE-REPEAT) .
C!	(n a ->)	Занесение младших разрядов (байта) n по адресу a.
C@	(a -> n)	Выборка младших разрядов (байта) содержимого по адресу a. Старшие разряды результата обнуляются.
CONSTANT	период-компиляции: (n ->) период-выполнения: (-> n)	Определяющее слово, которое в период компиляции создает слово, заносящее при своем исполнении (в период выполнения) число n в вершину стека.
COUNT	(a -> a+1 u)	Преобразование указателя строки со счетчиком в адрес первой литеры и число литер строки.
CR	(->)	Вывод символа возврата каретки (ASCII 13).
CREATE	(->)	Определяющее слово, создающее заголовок словарной статьи Форта. Выбирает из входного потока имя и строит поля: имени, связи и кода. В cfa помещается ссылка на программу периода выполнения для VARIABLE.
DECIMAL	(->)	Установка основания системы счисления путем занесения числа 10 в переменную BASE равным десяти.
DEFINITIONS	(->)	Установка контекстного словаря компиляции (CURRENT) на текущий словарь поиска (CONTEXT).
DO	(*граница счетчик* ->)	Компилирующее слово. Служит началом цикла DO. Исходным значением счетчика цикла является *счетчик*. Цикл завершается при переходе значения счетчика через *границу* (в обоих направлениях). Цикл всегда исполняется по крайней мере один раз.
DOES>	период-компиляции: (->) период-выполнения: (-> pfa)	Путем использования определяющего слова DOES> в словах Форта задаются действия периода выполнения. Во время выполнения на стек заносится pfa слова, подлежащего исполнению.
DROP	(n ->)	Удаление верхнего элемента стека.
DUP	(n n ->)	Занесение копии верхнего элемента стека в вершину последнего.

Слово	Стек	Описание
ELSE		Компилирующее слово. Используется в конструкциях IF-ELSE-THEN. Слова,расположенные между ELSE и THEN, выполняются в том случае, если флаг для IF ложен.
EMIT	(n ->)	Вывод n в выходной поток в символьном виде.
EXECUTE	(cfa ->)	Выполнение слова, pfa которого находится на стеке.
FLUSH	(->)	Запись обновленных дисковых буферов во внешнюю память и их освобождение.
FORGET	(->)	Выбор очередного слова из входного потока. Забываются все слова, определенные перед этим словом. Если слово [illegible], выдается аварийное сообщение.
FORTH	(->)	Контекстный словарь, содержащий стандартные слова Форта.
HERE	(-> a)	Занесение в вершину стека очередного доступного адреса (первого свободного адреса в словаре).
I	(-> n)	Используется в конструкциях DO-LOOP или DO-+LOOP для занесения в вершину стека текущего значения счетчика.
IF	(f ->)	Компилирующее слово, которое в случае истинности флага f заставит в период-выполнения исполняться слова, следующие непосредственно за IF. В противном случае будут исполняться слова, следующие за ELSE или THEN (если конструкция ELSE отсутствует).
IMMEDIATE	(->)	Отмечает последнее определенное слово как компилирующее, которое при режиме компиляции будет не компилироваться, а исполняться.
J	(-> n)	Применяется во вложенных циклах типа DO. Заносит в вершину стека значение счетчика внешнего цикла (см. I).
KEY	(-> n)	Занесение на стек символа из входного потока.
LATEST	(-> nfa)	Занесение на стек nfa последней созданной статьи.
LITERAL	(n ->)	Компилирующее слово, которое в период выполнения вызывает занесение на стек n.
LOAD	(n ->)	Слово для работы с диском. Вызывает интерпретацию блока n.
LOOP		Компилирующее слово, которое в период выполнения увеличивает значение счетчика DO-цикла на единицу. Если полученное значение счетчика выходит за границу, выполнение цикла завершается и начинают исполняться слова, следующие за словом LOOP. В противном случае исполняются слова, следующие за DO.
MAX	(n1 n2 -> n3)	Занесение на стек большего из чисел n1 и n2.
MIN	(n1 n2 -> n3)	Занесение на стек меньшего из чисел n1 и n2.
NEGATE	(n -> -n)	Занесение на стек дополнения до двух от n или отрицательной величины n.
NOT	(n1 -> n2)	Поразрядное инвертирование n1.
OR	(n1 n2 -> n3)	Выполнение поразрядной дизъюнкции над n1 и n2 с получением n3.
OVER	(n1 n2 -> n1 n2 n1)	Занесение на стек копии второго элемента стека.
R>	(-> n)	Перенесение верхнего элемента стека возвратов на стек данных.
R@	(-> n)	Копирование верхнего элемента стека возвратов на стек данных.
REPEAT	(->)	Компилирующее слово. Применяется в циклах типа BEGIN-WHILE-REPEAT для передачи управления на слово, следующее за BEGIN.
ROT	(n1 n2 n3 -> n2 n3 n1)	Перемещение третьего элемента стека в его вершину.
SAVE-BUFFERS	(->)	Запись на диск всех обновленных дисковых буферов.
SPACE	(->)	Вывод символа пробела (ASCII 32).

STATE	(-> pfa)	Системная переменная, содержащая текущее состояние Форта. Ноль - компиляция, в противном случае - интерпретация.
SWAP	(n1 n2 -> n2 n1)	Поменять местами два верхних элемента стека.
THEN	(->)	Компилирующее слово,применяемое в конструкциях IF-THEN или IF-ELSE-THEN (см. IF и ELSE).
TYPE	(a u ->)	Вывод строки, заданной двумя верхними элементами стека.
UNTIL	(f ->)	Компилирующее слово, применяемое в конструкциях BEGIN-UNTIL. Если в период выполнения флаг f истинен, исполнение цикла завершается и начинают выполняться слова, следующие за UNTIL. В противном случае выполняются слова, следующие за BEGIN.
VARIABLE	(->)	Определяющее слово для создания переменных. В период выполнения на стек помещается pfa значения этой переменной.
VOCABULARY	(n ->)	Определяющее слово для создания контекстных словарей.
WHILE	(f ->)	Компилирующее слово, применяемое в конструкциях BEGIN-WHILE-REPEAT. Если в период выполнения флаг f истинен, исполняются слова, следующие за WHILE. В противном случае выполняются слова, следующие за REPEAT.
XOR	(n1 n2 -> n3)	Выполнение поразрядного "исключающего или" над n1 и n2 с получением n3.
[	(->)	Компилирующее слово. Переход к состоянию интерпретации.
[']	Период-компиляции: (->) период-выполнения: (-> cfa)	Компилирующее слово, применяемое внутри определения через двоеточие для компиляции cfa очередного слова из входного потока. В период выполнения этот cfa заносится на стек. Данное слово при интерпретации выполняется как слово ' .
]	(->)	Переход к состоянию компиляции.

Приложение Г

ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА ПОСТРОЕНИЯ ЭКСПЕРТНЫХ СИСТЕМ

Инструментальные средства (ИС) представляют собой пакеты программ, облегчающие процесс построения экспертных систем. Они включают в себя блок логического вывода, пользовательский интерфейс и программу приобретения знаний, позволяющую пользователю пополнять систему знаниями.

Каждое инструментальное средство имеет свои особенности, что позволяет наиболее эффективно его применять только в какой-то одной прикладной области. К выбору инструментального средства для построения конкретной экспертной системы нужно подходить очень внимательно.

В настоящем приложении перечислены наиболее распространенные инструментальные средства построения экспертных систем.

Название ИС (разработчик)	Модель представ-ления знаний	Язык реализации (тип компьютера)	Область приложений	Цена (дол.)
1	2	3	4	5
Advisor (Ultimate Media Inc.)	Правила	Ассемблер (Apple, Commodore 64)	Обучение	99.50
Advice Language/X (Univ. of Edinburgh,. Scotl.)	Правила	Паскаль (Apple)	Распределение аудиторий	----
Duck (Smart System Technology)	Правила, немоно-тонный вывод	---- (Apollo, Symbolics, VAX)	Организационное управление	6000
ESP/Advisor (Expert Systems Intern.)	Правила	Пролог (IBM PC)	Разработка небольших ЭС	895
EXPERT (Rutgers University)	Правила	Фортран	Фармакология, диагности-ка ревматич. заболеваний	----
EXPERT-2 (Mountain View Press)	Правила	Форт (IBM PC, Apple)	Разработка небольших ЭС	100

1	2	3	4	5
Expert-Ease (Jeffrey-Perrone & Assoc.)	Примеры решений	---- (IBM PC, DEC, Victor 9000)	Разработка небольших ЭС	695
ExperOPS V (ExperTelligence)	Правила	ExpertLISP (Macintosn)	Разработка небольших ЭС	325
EXSYS (Exsys Inc.)	Правила	---- (IBM PC)	Разработка небольших ЭС	295
INSIGHT-1 (Level 5 Research)	Правила	Паскаль (IBM PC, DEC, Victor 9000)	Обучение	95
INSIGHT-2 (Level 5 Research)	Правила	Паскаль (IBM PC, DEC, Victor 9000)	Обучение	485
KDS Development System (KDS)	Правила	Ассемблер (IBM PC)	Разработка небольших ЭС	795
Knowledge Craft (Carnegie Group Inc.)	Фреймы и праaила	Common LISP (DEC VAX)	Управление производством	50000

1	2	3	4	5
Knowledge Engineering Environment - KEE (IntelliCorp)	Фреймы	Лисп (Symbolics 3600, Xerox 1100)	Генная инженерия и др.	60000
Knowledge Engineering System - KES (Software A & E)	Правила	IQLISP (IBM PC, DEC VAX)	----	4000
LOOPS (Xerox Palo Alto Research)	Фреймы	INTERLISP (Xerox 1100)	Демонстрацисный прототип	300
M.1 (Teknowledge Inc.)	Правила	Пролог (IBM PC)	Разработка больших ЭС	5000
MicroExpert (McGraw Hill Book Co.)	Правила	Паскаль (IBM PC, Apple)	Разработка небольших ЭС, обучение	49.50
Nexpert (Neuron Data Inc.)	Правила	---- (Macintosh)	----	5000
OPS5 (Carnegie-Mellon Univ.)	Правилв	Лисп (VAX 11/780)	Определение конфигурации ЭВМ	----
OPS5e (Verbs Inc., San Diego)	Прввила	ZetaLISP (Symbolics 3600)	----	3000

1	2	3	4	5
OPS5+ (Artelligence Inc.)	Правила	---- (IBM PC)	Разработка больших ЭС	3000
Personal Consultant (Texas Instruments)	Правила	IQLISP (TI Professional & MS-DOS)	Разработка больших ЭС	950
Personal Consultant Plus (Texas Instruments)	Фреймы и правила	IQLISP (TI Professional)	Разработка небольших ЭС	3000
Rule-Master (Radian Corporation)	Правила	Си (IBM XT/AT, VAX, ЭВМ с ОС UNIX)	Диагностика и управление, военное дело, страхование	5000-25000
SeRIES-PC (SRI International)	Правила	IQLISP (IBM PC)	----	5000
S.1 (Teknowledge Inc.)	Правила	Лисп (DEC VAX, Xerox)	----	70000
TIMM (General Research Corp.)	Правила	Фортран (IBM, DEC VAX, IBM PC)	Помощь пилоту вертолета во время боя	39500

Приложение Д

ЭКСПЕРТНЫЕ СИСТЕМЫ

В настоящее время существует ряд экспертных систем, успешно применяемых при решении таких задач, которые ранее могли быть решены только экспертами. В данном приложении перечислены системы, представляющие наибольший интерес. Каждая из приведенных систем включает как оболочку, так и базу знаний. Следует иметь в виду, что это лишь небольшая часть работоспособных систем из тех, которые в настоящее время имеются на рынке программных средств.

Название ЭС (разработчик)	Инструментальное средство (представление знаний)	Язык реализации (тип компьютера)	Область приложений	Год выпуска
1	2	3	4	5
DELTA-CAIS (General Electric)	---- (Правила)	Форт	Ремонт локомотивов	1981
DENDRAL (Stanford University)	----	Лисп	Масспектрометрия в органической химии	1970
DRILLING ADVISOR (Teknowledge)	KS 300	----	Диагностика технических систем	----
GENESIS (Intellicorp)	MOLGEN (фреймы)	IBM 370, DEC VAX	Генная инженерия	1981
HEARSAY-2 (Carnegie-Mellon University)	AGE и HEARSAY-3 (Архитектура доски объявлений)	SAIL	Понимание речи	1975
INTERNIST/CADUCEUS (University of Pittsburgh)	--- (Правила)	Лисп	Медицинская диагностика	1970

1	2	3	4	5
MACSYMA (MIT)	----	Лисп (KL-10)	Символьные преобразования математических выражений	1980
PROSPECTOR (SRI International)	KAS (Семантическая сеть с правилами)	Лисп	Геология	1978
PUFF (Stanford University)	EMYCIN (Правила)	Лисп (PDP-11)	Легочные заболевания	1981
MYCIN (Stanford University)	EMYCIN (Правила)	Лисп (PDP-11)	Лечение менингита; диагностика инфекционных заболеваний	1975
XCON, XSEL (Carnegie-Mellon Univ. & DEC)	OPS5 (Правила)	--- (DEC VAX)	Определение конфигурации компьютерных систем типа VAX	1978
(Rutgers University & Helena Laboratories)	EXPERT (правила)	Фортран	Анализ результатов электрофореза	1980

ГЛОССАРИЙ

Алгоритм. Эффективная конечная процедура.

Альфа-бета алгоритм. Стратегия поиска решения с сокращением перебора за счет отбрасывания заведомо бесперспективных ветвей.

Ассоциативный список. Представление списковой структуры, при которой в ячейках связи хранятся пары атрибут-значение (А-З - пары), называемые также *А-списками*.

Атом. Основной элемент данных в Лиспе. Используется для представления объектов. Как атом, так и списки являются символьными выражениями.

Атрибут. Некоторое свойство объекта. Например, в факте "Цвет слона - серый" понятие *слон* - это объект, *цвет* - атрибут, а *серый* - значение.

Атрибут-значение (А-З). Способ представления фактуального знания, при котором атрибутам могут присваиваться значения. Например, в факте "Животное имеет волосяной покров" атрибутом служит *животное*, а его значением является *имеет_волосяной_покров*.

База данных. В системе, основанной на знаниях, этот термин, как правило, означает рабочую память (см. *рабочая память*).

База знаний. Часть системы, основанной на знаниях, которая состоит из фактов и правил. В продукционной системе она состоит из базы правил и рабочей памяти.

Булевские операторы. Операторы исчисления предикатов. К ним относятся операторы И, ИЛИ, НЕ.

Висячие ссылки. Ситуация в Лиспе при распределении памяти, когда некоторая ячейка считается свободной, а на самом деле занята. При таком положении возможны "висячие" указатели из активных ячеек (см. *сборка мусора*).

Внешний интерпретатор. Интерпретатор Форта, взаимодействующий с пользователем.

Внутренний интерпретатор. Часть виртуальной Форт-машины, интерпретирующая шитый код.

Возврат. Переход при выводе на несколько шагов назад для анализа или трассировки хода рассуждения, а также в целях поиска альтернативного пути.

Вывод. Процесс рассуждения, во время которого из известных фактов выводятся новые факты.

Глубинные знания. Основные знания о некоторой области (см. также *поверхностные знания*).

Демоны. Скрытые или виртуальные процедуры в системах, основанных на знаниях, активизируемые данными.

Дерево вывода. Графическое представление возможных путей поиска решения в Прологе.

Диагностические системы. Тип систем, основанных на знаниях, которые применяются для нахождения причин неполадок в технических системах или заболеваний у человека.

Динамическое распределение. Распределение памяти в период выполнения программы.

Долговременная память. Часть человеческой памяти, предназначенная для хранения сведений, необходимость в которых может возникнуть не только в момент их появления, но и некоторое время спустя. Аналогична внешней памяти компьютера или базе правил в системе, основанной на знаниях.

Домен. Определенная часть знаний о некоторой области. Информатика представляет собой весьма обширный домен, в то время как когнитивное моделирование - более узкий.

Доска объявлений. Глобальная структура данных в экспертной системе, через которую осуществляется взаимодействие различных источников знаний.

Знания. Совокупность фактов, закономерностей и эвристических правил, с помощью которых можно решить поставленную задачу.

Идентификатор списка. Переменная Форта, значение которой указывает на именуемый ею список.

Иерархия. Отношение подчиненности между понятиями или объектами.

Инженер знаний. Специалист, которому знакомы содержательная сторона задачи и методы структурирования знаний в экспертных системах. Эта область деятельности предполагает наличие навыков в теории познания, информатике и системах, основанных на знаниях.

Интерпретатор. Процедура, производящая разбор входного потока данных, преобразование его в исполнимую форму и немедленно его выполняющая.

Интерпретирующая система. Тип системы, основанной на знаниях, применяемых для вывода заключений по наблюдаемым данным.

Искусственный интеллект (ИИ). Одна из ветвей информатики. Основной проблемой ИИ является разработка методов представления знаний и решения неформализуемых задач.

Когнитивное моделирование. Область науки, целью которой является разработка теории и моделей человеческого мышления и его функций.

Компилируемые знания. Знания, полученные от эксперта или из другого источника и организованные в форме, позволяющей включить их в систему, основанную на знаниях. Обычно знания каким-либо образом структурируются, например, путем выделения чанков и установления взаимосвязей между ними.

Компилятор. Программа, анализирующая входной поток и запоминающая переработанную информацию в форме, наиболее приемлемой для дальнейшего эффективного использования.

Компонент управления. Процедура вывода или интерпретатор правил, определяющие последовательность применения правил.

Консультационный режим. Один из интерактивных режимов в ЭС, при котором пользователь продвигается к решению задачи, задавая системе вопросы.

Лисп. Программная среда для решения символьных задач.

Логика первого порядка. Расширение пропозиционального исчисления кванторами всеобщности и существования для переменных.

Логические связки. Булевские операции, применяемые в логике первого порядка для выражения отношений между термами, например, СЛЕДУЕТ или ЭКВИВАЛЕНТНО (см. табл. 4.1).

Механизм вывода. Часть продукционной системы, которая выводит новые факты из имеющихся в базе знаний.

Модус поненс. Правило вывода вида ЕСЛИ (А и ЕСЛИ А ТО В) ТО В.

Монотонный вывод. Вид рассуждения в ЭС, при котором факты из рабочей памяти не удаляются.

Мусор. Вид ошибочного распределения памяти в Лиспе, при котором ячейка может считаться занятой, будучи фактически свободной (см. *висячие ссылки*).

Наследование. Процесс получения объектом значений для своих атрибутов от класса объектов, находящегося выше него в родовидовой иерархии.

Немонотонный вывод. Вид рассуждения, при котором факты из рабочей памяти могут удаляться.

Область списков. Часть машинной памяти, используемая для хранения списков.

Обратный вывод. Стратегия вывода, при которой вывод производится путем подбора подходящих фактов под имеющееся заключение.

Объект. Элемент системы, основанной на знаниях, который описывается одним или несколькими атрибутами.

Объект-атрибут-значение (А-О-З). Способ выражения фактуального знания, при котором объект представляется в виде набора свойств с их значениями. Например, в факте "Брат пациента - Джон" понятие ПАЦИЕНТ является объектом, понятие БРАТ - атрибутом, а ДЖОН - значением (см. *атрибут-значение*).

Определяющие слова. Слова Форта, применяющиеся для определения других слов.

Отметить и удалить. Схема управления списками в Лиспе, при которой все активные ячейки помечаются, а оставшиеся свободными заносятся в список свободных ячеек.

Параллельная архитектура. Вид архитектуры компьютера, включающего несколько одновременно функционирующих процессоров.

ПЕРВЫЙ. См. CAR.

Плоское представление. Представление знаний, вид которого отличен от иерархии.

Поиск в глубину. Стратегия поиска, при которой исследование очередной альтернативы продолжается до тех пор, пока дальнейшее продвижение в глубь дерева решений окажется невозможным.

Поиск в ширину. Стратегия поиска решения, при которой сначала просматриваются все узлы данного уровня, а затем происходит спуск на более детальный уровень пространства альтернатив.

Понимание естественного языка. Ветвь искусственного интеллекта, целью которой является разработка компьютерных программ, выполняющих перевод с одного языка на другой или способных взаимодействовать с человеком на естественном языке.

Правило типа IF-THEN. Оператор, реализующий отношение между посылкой и следствием, то же, что и *продукция*.

Предложение. Конъюнктивно соединяемый терм в Прологе.

Представление знаний. Метод структурирования фактов и отношений для включения их в базу знаний.

Прямой вывод. Стратегия вывода, при которой правила применяются к фактам для получения заключений, из них вытекающих.

Разрешение конфликта. В продукционных системах это процесс выбора нужного правила из нескольких путем сопоставления с заданными фактами в рабочей памяти.

Расширенная переменная. Переменная Форта с дополнительно выделенной памятью.

Сборка мусора. Восстановление неиспользуемых ячеек в Лиспе.

Система управления сетевыми базами данных. Тип систем управления базами данных, в которых данные представлены в виде иерархии и запись-"потомок" может иметь больше информации, чем ее владелец, т.е. запись-"предок".

Системы, основанные на знаниях. Класс компьютерных программ, использующих для решения задач знания и процедуры вывода.

Скобочная нотация. Вид записи списков в Лиспе, при которой элементы списка заключаются в круглые скобки, например (A B).

Словарь. Список слов Форта.

Список. Упорядоченное множество связных ячеек памяти в Лиспе (см. *атом*).

Стек параметров. Стек Форта, используемый для передачи данных между словами.

Структура данных. Способ организации данных и определение методов доступа к ним.

Тело. Цели некоторого предложения в языке Пролог.

Указатель списка. Адрес головы списка.

Факт. Утверждение или посылка, которые являются истинными. Факт может состоять из атрибута и соответствующего значения.

Фактор достоверности (ФД). Мера доверия к факту или отношению. Отличается от понятия вероятности.

Фасеты. Ограничения на значения, хранимые в слотах при представлении знаний в виде фреймов.

Фрейм. Структура для представления знаний, которая состоит из одного или нескольких атрибутов, описывающих объект. Значение каждого атрибута хранится в слоте, таким образом, фрейм представляет собой множество слотов, связанных с объектом.

Функциональное программирование. Метод программирования, основанный на передаче параметров между функциями.

Чанк. Фрагмент знаний, хранимый и используемый как единое целое.

Эвристические правила. Неформальные знания, используемые в целях повышения эффективности поиска в данной предметной области.

Экспертные системы. См. *системы, основанные на знаниях.*

Экспертиза. Эвристические и точные знания, которыми обладают люди в определенной предметной области. Экспертиза производится путем сбора больших объемов знаний в некоторой предметной области и организации их в соответствующие иерархические структуры для дальнейшего применения в процессе решения прикладных задач.

Ядро. Слова машинного уровня в Форте, с помощью которых определяются остальные слова Форта.

Ячейки связи. Тип данных Лиспа, состоящий из двух компонент - ГОЛОВЫ и ХВОСТА (см. CAR и CDR).

CAR. Адресный указатель на элемент списка в Лиспе, альтернативные имена - ГОЛОВА или ПЕРВЫЙ.

CDR. Адресный указатель на следующую ячейку связи списка в Лиспе, альтернативные имена - ХВОСТ или ОСТАТОК.

cfa. Адрес поля кода в Форте - участка, где находится указатель на процедуру периода-выполнения данной словарной статьи.

nfa. Адрес поля имени в Форте, в котором располагается имя словарной статьи.

NIL (НУЛЬ). Атом Лиспа, являющийся одновременно и пустым списком.

ЛИТЕРАТУРА

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

1. Barr A., Feigenbaum E. The Handbook of Artificial Intelligence. - William Kaufmann Inc., 1982.
2. Feigenbaum E., McCorduck P. The Fifth Generation: Artificial Intelligence and Japan's Computer Challenge to the World. - Reading, Mass.: Addison-Wesley Publ., 1983.
3. Winston P. Artificial Intelligence. - Reading, Mass.: Addison-Wesley Publ., 1984.
4. Winston P. The AI Buseness: Commercial Uses of Artificial Intelligence. - Cambridge,Mass.: The MIT Press, 1984.
5. BYTE. - Vol.6, N 9, 1981.
6. BYTE. - Vol.10, N 4, 1985.

ЭКСПЕРТНЫЕ СИСТЕМЫ

7. Forsith R. Expert Systems: Principles and Case Studies. - London: Chapman and Hall, 1984.
8. Freiling A. Starting a Knowledge Engineering Project. - AI Magazine, vol.6, N 3, 1985.
9. Genesereth M. Fundamentals of Artificial Intelligence. - Palo Alto, Calif.: Teknowledge, 1982.
10. Gevarter W. An Overview of Expert Systems. - Washington D.C.: National Bureau of Standarts, 1982.
11. Hayes-Roth F., Waterman D. Building Expert Systems. - Reading, Mass.: Addison-Wesley Publ., 1983.

Harmon P. Expert Systems: Artificial Intelligence in Business. - N.-Y.: John Wiley & Sons Inc., 1985.

12. Stefik M. The Organisation of Expert Systems. - Palo Alto, Calif.: Xerox Palo Alto Center, 1982.
13. Communication of the ACM. - Vol.28, N 5, 1985.
14. BYTE. - Vol.4, N 8, 1979.

ЭКСПЕРТНЫЕ СИСТЕМЫ, РЕАЛИЗОВАННЫЕ НА ФОРТЕ

15. Johnson H. Expert Systems for Diesel Locomotive Repair. - Journal of Forth Application and Research, September, 1983.

16. Redington D. Outline of a Forth Oriented Real-time Expert System for Sleep Staging. - Forth Modification Laboratory: Sixth FORML conf., November, 1984.
17. Park J. MVP-FORTH Expert System Toolkit. - Mountain View, Calif.: Mountain View Press, 1984.

ЯЗЫКИ ПРОГРАММИРОВАНИЯ

18. Brodie L. Starting Forth. - Englewood Clifs: Prentice-Hall, 1981.
19. Meehan J. The New UCI LISP Manual. - Hillsdale, N.J., 1979.
20. Winston P., Horn B. LISP. - Reading, Mass.: Addison-Wesley Publ., 1984.
21. BYTE. - Vol.10, N 8, 1985.
22. BYTE. - Vol.6, N 8, 1981.

ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА

23. Goldenberg J. Experts on Calls. - PC World, September, 1985.
24. PC Magazine. - Vol.4, N 8, 1985.

ПРЕДСТАВЛЕНИЕ ЗНАНИЙ

25. Communication of the ACM. - Vol.28, N 9, 1985.

ОГЛАВЛЕНИЕ

Предисловие к русскому изданию 5

Предисловие 14

ЧАСТЬ I. ЭКСПЕРТНЫЕ СИСТЕМЫ - СИСТЕМЫ, ОСНОВАННЫЕ НА ЗНАНИЯХ 17

1. Введение в искусственный интеллект 18
   - Что такое мышление? 18
   - Процесс мышления 19
   - Организация хранения информации в человеческой памяти 23
   - Мир искусственного интеллекта 25
     Представление знаний * Решение задач * Экспертные системы * Средства общения с ЭВМ на естественном языке * Обучение * Когнитивное моделирование * Обработка визуальной информации и робототехника
   - Современный уровень развития экспертных систем 33
   - Упражнения 34

2. Системы, основанные на знаниях 36
   - Что такое система, основанная на знаниях? 36
   - Отличительные особенности 39
   - Области применения 39
     Медицинская диагностика * Прогнозирование * Планирование * Интерпретация * Контроль и управление * Диагностика неисправностей в механических и электрических устройствах * Обучение
   - Критерии использования 42
   - Ограничения 44
   - Преимущества 45
   - Возможности реализации на персональных компьютерах 45
   - Эвристические и алгоритмические методы решения задач 47
   - Символьная и числовая обработка данных 49
   - Рассуждения с расширяющимся и уменьшающимся множеством заключений .. 50
   - Выводы 51
   - Упражнения 51

[illegible] Структура систем, основанных н[illegible]
   - Прод[illegible]
     Память д[illegible] хранения п[illegible] * Р[illegible] (база да[illegible]х) * Механиз[illegible] [illegible]крет[illegible] приобре[illegible] [illegible]ний * Сред[illegible] общения на [illegible] Под[illegible]ге[illegible]
   - Стр[illegible]
     Прямо[illegible]
   - Коммерческие пр[illegible] си[illegible] [illegible]
   - Инструментальные средства . [illegible]
   - Языки программирования с[illegible] [illegible]снова[illegible]ных на знаниях [illegible]
   - Продукционная система - [illegible] процессов мышления человека ... 7[illegible]
   - Упражнения ... 74

[illegible] Представление знаний 75
   - Игра "Отгадай животн[illegible]" .... 7[illegible]
   - Представление знаний в [illegible] правил .... 7[illegible]
   - Фреймы 81
   - Представление знаний в виде семантичес[illegible] [illegible]ти 85
   - Представление знаний средствами логики п[illegible]ка 87
     Основные понят[illegible] логики предикатов * Исчисл[illegible] предикатов с кванторами (логик[illegible] предикато[illegible] * Синтаксис языка логики предикатов
   - Системы, использующие принцип доски об[illegible] 92
   - Формирование базы знаний по примерам. [illegible]
   - Построение демонстрационной базы знаний [illegible]
   - Упражнения. ... [illegible]

[illegible] Инженерия знаний .... [illegible]
   - Описание предметной области .... [illegible]
     Определение характера решаемых задач * Выявлени[illegible] предметной области * Установление взаимосвязей м[illegible] * Формализация знаний * Вы[illegible] специфи[illegible] предме[illegible] обла[illegible] * Выбор мо[illegible] реа[illegible] [illegible] Структурная иерархия * Причинно-следс[illegible]нная и[illegible] поведения системы * Функциональная иерархи[illegible] * [illegible] поверхностные знания * Сопоставление мето[illegible] реализации знани[illegible]
   - Приобретение знаний. .... [illegible]
   - Упражнения .... [illegible]

ЧАСТЬ 2. ПОСТРОЕНИЕ СИСТЕМ, ОСНОВАННЫХ НА ЗНАНИЯХ 112

6. Язык Форт - мощное средство построения экспертных систем 113
- Слова и словарь языка Форт 114
- Передача данных через стек 116
- Иерархическая декомпозиция и разбиение на модули 117
- Выполнение арифметических операций 119
- Манипулирование элементами стека 123
- Доступ к данным 125
- Флаги, логические операторы и сравнение чисел 126
- Управляющие конструкции 128
- Стек возвратов 135
- Обработка строк 136
- Потоки текстов 137
- Структура слов Форта 138
- Управление словарем 141
- Виртуальная Форт-машина 141
- Определяющие и компилирующие слова 144
- Слова для управления внешней памятью 152
- Упражнения 153

7. Обработка списков 154
- Для чего нужно эмулировать Лисп? 155
- Статическое и динамическое управление памятью 156
- Что такое список? 157
- Простейшие операции над списками 159
- Идентификатор и указатель списка 161
- Вывод списков на печать 164
- Ввод списков 164
- Типы данных при работе со списками 167
- Что такое НУЛЬ? 168
- Списки свойств 168
- Ассоциативные списки 170
- Функции РАВНО и РАВ 171
- Операции преобразования списков 172
- Другие функции работы со списками 173
- Упражнения 176

8. Методы программирования 178
- Рекурсия 178
- Сборка мусора 188
- Реализация функций преобразования списков 190
- Функции непосредственного преобразования списков и учет ссылок 192
- Упражнения 193

9. Пролог - язык разработки систем, основанных на знаниях 194
- Логическое программирование на Прологе 194
- Интерпретатор Пролога 199
- Реализация поиска 203
- Деревья вывода 211
- Поиск в ширину и эвристический поиск 212
- Унификация 213
- Упражнения 223

10. Дополнительные возможности 225
- Встроенные предикаты Пролога 225
- Процедурное дополнение и вызов по образцу 226
- Немонотонные рассуждения 227
- Объектно-ориентированное программирование 229
- Метарассуждения: управление выводом 233
- Неопределенность и достоверность 235
- Контекстные словари Форта 237
- Параллельные вычисления 241
- Упражнения 243

11. Обучение и распознавание образов 244
- Обучение 244
- Распознавание образов 247
  - Свойства гиперплоскости * Классификаторы, построенные по критерию минимального расстояния
- Алгоритмы классификации образов 261
- Упражнения 271

Приложение А. Исходные тексты программ 272

Приложение Б. Подпрограммы диагностики 287

Приложение В. Словарь Форта 293

Приложение Г. Инструментальные средства построения экспертных систем 299

Приложение Д. Экспертные системы 304

Глоссарий 307

Литература 314

Научное издание

Таунсенд Карл, Фохт Деннис

ПРОЕКТИРОВАНИЕ И ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ЭКСПЕРТНЫХ СИСТЕМ НА ПЕРСОНАЛЬНЫХ ЭВМ

Книга одобрена на заседании секции редсовета по электронной обработке данных в экономике 28.10.86 г.

Зав. редакцией К.В. Коробов
Редактор Н.К. Логинова
Худож. редактор Ю.А. Артюхов
Техн. редактор Г.А. Полякова
Корректоры Г.А. Башарина, Г.В. Хлопцева
Переплет художника Е.К. Самойлова

ИБ N 2307

Оригинал-макет книги подготовлен к печати с помощью текстового процессора Microsoft Word 5.0

Подписано в печать 20.09.90.
Формат 60 x 88 1/16 Бум. офсетная. Гарнитура "Литературная"
Печать офсетная. Усл.п.л. 19,6. Усл. кр.-отт. 19,6. Уч.-изд. л. 19,11.
Тираж 30 000 экз. Заказ 299. Цена 4 р. 50 к.

Издательство "Финансы и статистика",
101000, Москва, ул. Чернышевского, 7.

Отпечатано в типографии им. Е.Котлякова
издательства "Финансы и статистика"
Государственного комитета СССР по печати.
195273, Ленинград, ул. Руставели, 13.