СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

И.Г. СИДОРКИНА

УЧЕБНОЕ ПОСОБИЕ



КНОРУС

И.Г. Сидоркина

СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Допущено УМО вузов
по университетскому политехническому образованию
в качестве **учебного пособия**
для студентов высших учебных заведений,
обучающихся по направлению
«Информатика и вычислительная техника»

КНОРУС • МОСКВА • 2015

Knorus media
электронные версии книг

УДК 004.8(075.8)
ББК 32.813я73
С34

Рецензенты:
Е.К. Пугачёв, доц. кафедры «Компьютерные системы и сети» МГТУ им. Н.Э. Баумана, канд. техн. наук,
В.В. Кошкин, доц. кафедры «Проектирование и производство электронно-вычислительных систем» МарГТУ, канд. техн. наук,
Ю.А. Гатчин, заведующий кафедрой СПбГУ ИТМО, д-р техн. наук, проф.

Сидоркина И.Г.

С34 Системы искусственного интеллекта : учебное пособие / И.Г. Сидоркина. — М. : КНОРУС, 2015. — 248 с.

ISBN 978-5-406-04417-9

Рассмотрены теоретические аспекты проектирования систем искусственного интеллекта, даны модели представления знаний, изложены материалы информационного, справочного, консультирующего характера по использованию инструментальных средств, а также по технологическим особенностям разработки систем данного класса.

Для студентов, обучающихся по направлению «Информатика и вычислительная техника».

УДК 004.8(075.8)
ББК 32.813я73

Сидоркина Ирина Геннадьевна
СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Сертификат соответствия № РОСС RU. АЕ51. Н 16604 от 07.07.2014.

Изд. № 8699. Формат 60×90/16.
Гарнитура «NewtonC». Печать офсетная.
Усл. печ. л. 15,5. Уч.-изд. л. 13,0.

ООО «Издательство «КноРус».
117218, г. Москва, ул. Кедрова, д. 14, корп. 2.
Тел.: 8-495-741-46-28.
E-mail: office@knorus.ru http://www.knorus.ru

Отпечатано в ООО «Контакт».
107150, г. Москва, проезд Подбельского 4-й, д. 3.

ISBN 978-5-406-04417-9

© Сидоркина И.Г., 2015
© ООО «Издательство «КноРус», 2015

ОГЛАВЛЕНИЕ

Предисловие 6

Список сокращений 8

Введение 9

1. Базы данных, ориентированные на искусственный интеллект 13

1.1. Экспертные системы и их особенности 13

1.2. Основные типы задач, решаемых с помощью экспертных систем . . 18

1.3. Особенности разработки экспертных систем 21

1.4. Виды экспертных систем 23

1.5. Представление знаний в системах искусственного интеллекта 24

2. Формализация знаний о проблемной области 29

2.1. Таксономическая классификационная схема 29

2.2. Онтологический подход к представлению проблемной информации 33

2.3. Модели представления знаний 41

3. Инструментальные средства логического программирования 53

3.1. Язык логического программирования Пролог 53

3.2. Основные разделы программы 55

3.3. Рекурсивные вычисления в Пролог-программе 57

3.4. Процесс реализации вывода 58

3.5. Предикаты 60

3.6. Списковые структуры 68

3.7. Вызов внешних функций из Пролог-программы и интерфейс с программами на других языках программирования 70

3.8. Пример реализации экспертной системы на языке Пролог 72

3.9. Диалекты и языки, используемые для задач искусственного интеллекта 75

4. Организация принятия решений в экспертных системах 77

4.1. Организация логического вывода в экспертных системах 77

4.2. Правила 78

4.3. Поиск решений 80

4.4. Управляющая структура 81

4.5. Технологии принятия решений в системах с базами знаний 83

4.6. Методы поиска, реализованные в экспертных системах 83

4.7. Использование процедур 86
4.8. Представление неопределенности в информационных приложениях с базами знаний 86

5. Интеллектуальные технологии обработки информации 90
5.1. Интеллектуальные системы, основанные на нечеткой логике 90
5.2. Нейронные сети 96
5.3. Эволюционные вычисления 109
5.4. Комплексный подход к проектированию систем искусственного интеллекта 120
5.5. Инструментальные средства представления знаний 121

6. Система моделирования ЭО КАРРА 124
6.1. Представление знаний в ЭО КАРРА. 124
6.2. Начало работы с ЭО КАРРА 125
6.3. Окно иерархии объектов (Object Browser) 127
6.4. Окно инструментов (Knowledge Tools) и редакторы знаний 128
6.5. Окно интерпретатора (KAL Interpreter) 134
6.6. Окно сеанса (Session) 135
6.7. Окно связи правил (Rule Relations) 137
6.8. Окно трассировки правил (Rule Trace) 138
6.9. Окно просмотра иерархии выводов (Inference Browser) 139
6.10. Средство объяснений ЭО КАРРА. 140

7. Стандартные функции ЭО КАРРА 142
7.1. Функции манипулирования знаниями 142
7.2. Математические функции. 148
7.3. Функции работы со строками 148
7.4. Функции работы со списками 149
7.5. Логические функции. 149
7.6. Функции работы с файлами 149
7.7. Функции управления 150
7.8. Функции работы с окнами 150
7.9. Функции работы с компонентами 151
7.10. Функции, определенные пользователем. 151

8. Работа с правилами в ЭО КАРРА 153
8.1. Создание и редактирование правил 153
8.2. Формирование списка правил 154
8.3. Создание и редактирование цели 154
8.4. Рассуждения в прямом направлении 155
8.5. Рассуждения в обратном направлении 161

9. Создание интерфейса пользователя в ЭО КАРРА 164
9.1. Стандартные компоненты интерфейса пользователя 165
9.2. Особенности русификации ЭО КАРРА 174
10. Инструментальная оболочка разработки ЭС — CLIPS 176
10.1. Общие сведения об ИО CLIPS 176
10.2. Программирование в ИО CLIPS 176
10.3. Интерфейс ИО CLIPS 186
10.4. Организация работы в ИО CLIPS 191
11. Разработка экспертной системы в ИО CLIPS 204
11.1. Подготовка исходных данных 204
11.2. Выделение сущностей 206
11.3. Сбор информации 208
11.4. Диагностические правила 209
11.5. Листинг программы 215
11.6. Выполнение программы 222
12. Создание проекта онтологии с помощью ИС Protégé 225
12.1. Создание нового проекта 225
12.2. Структура проекта 226
12.3. Работа с классами 227
12.4. Работа со слотами 231
12.5. Сохранение проекта в формате RDF 233
12.6. Экспорт онтологии в формат ЭО CLIPS 235
Заключение 237
Глоссарий 239
Список литературы 244

ПРЕДИСЛОВИЕ

Развитие современных средств вычислительной техники стимулирует развитие методов интеллектуальной обработки информации, проектирование систем и средств, базирующихся на технологиях искусственного интеллекта и баз знаний.

Искусственный интеллект предоставляет средства для решения задач, которые сложно или нерационально решать с помощью других методов. Эти задачи включают в себя эвристический поиск в локальной и распределенной средах (семантические Web), планирование, системы распознавания, робототехнику и др. Особую значимость приобретает создание экспертных систем, производство которых уже вышло на промышленный и коммерческий уровень.

В настоящем издании внимание сконцентрировано на тех концепциях искусственного интеллекта, которые положены в основу технологии проектирования экспертных систем.

Острая нехватка методической и специальной литературы по теории и практике проектирования различных классов систем искусственного интеллекта, недостаток квалифицированных специалистов в области инженерии знаний, незаполненный отечественный рынок программных продуктов для разработчиков экспертных систем затрудняют развитие и эффективное внедрение в сферу профессиональной деятельности общества разработок в области искусственного интеллекта. В результате на современном рынке программных продуктов и в учебно-методической литературе представлены многочисленные диалоговые пакеты и интерактивные инструментальные средства, которые не могут претендовать на вхождение в класс экспертных систем (в классическом понимании этого понятия).

По определению, предметом интереса в области искусственного интеллекта являются информационные проблемы, которые не могут быть решены с помощью традиционных технологий. Думаю, что данная тенденция сохранится и в обозримом будущем.

Процесс разработки экспертной системы имеет существенные отличия от разработки традиционных программных продуктов и требует участия высококвалифицированных специалистов в инженерии знаний. Одной из основных целей издания настоящего учебного пособия является некоторое восполнение недостатка литературы и учебно-методических разработок по вопросам создания систем искусственно-

го интеллекта, которые сейчас также называют системами, основанными на знаниях.

Данное учебное пособие, подготовленное в соответствии с требованиями государственного образовательного стандарта по направлению 230100 «Информатика и вычислительная техника», продолжает серию учебно-методической литературы, предназначенной для теоретической проработки и практической поддержки образовательных программ, ориентированных на подготовку специалистов по программированию и информационным технологиям, в рамках которых читаются курсы «Системы искусственного интеллекта», «Новые информационные технологии».

Задачей курсов является разработка студентами под руководством преподавателя демонстрационного прототипа системы, содержащей базы знаний по соответствующей заданию проблемной области. Реализация системы осуществляется на основе изученных методов, технологий искусственного интеллекта и инструментальных средств за счет времени, отведенного на лабораторные занятия (частично) и самостоятельную работу.

Настоящее учебное пособие, на мой взгляд, дает достаточное представление о теоретических аспектах проектирования систем, основанных на знаниях, и о материалах информационного, справочного, консультирующего характера, необходимых при выполнении лабораторных и расчетно-графических работ.

Автор выражает благодарность за оказанную помощь в подготовке данного издания проф. К.С. Хомякову (МГТУ им. Н.Э. Баумана).

Автор выражает глубокую признательность рецензентам — проф. кафедры компьютерных систем СПбГУИТМО А.Г. Коробейникову и доц. кафедры «Компьютерные системы и сети» МГТУ им. Н.Э. Баумана Е.К. Пугачёву.

Автор будет признателен всем читателям за замечания по содержанию учебного пособия, которые можно направить по адресам: igs@marstu.mari.ru и igs592000@mail.ru.

СПИСОК СОКРАЩЕНИЙ

БД	—	база данных
БЗ	—	база знаний
БПР	—	бизнес-процесс реинжиниринга
ГА	—	генетические алгоритмы
ЗУ	—	запоминающее устройство
ИАС	—	интеллектуальная автоматизированная система
ИИ	—	искусственный интеллект
ИнС	—	интеллектуальные системы
ИО	—	инструментальная оболочка
ИП	—	информационное приложение
ИС	—	инструментальная среда
НС	—	нейронная сеть
ООП	—	объектно-ориентированное программирование
ОС	—	операционная система
ПО	—	предметная область
СБИС	—	сверхбольшая интегральная микросхема
СОЗ	—	система, основанная на знаниях
ТС	—	техническое состояние
ЭВ	—	эволюционные вычисления
ЭВМ	—	электронно-вычислительная машина
ЭвС	—	эволюционные стратегии
ЭП	—	эволюционное программирование
ЭО	—	экспертные оболочки
ЭС	—	экспертная система
BC	—	Borland C
GUI	—	Graphical User Interface (графический интерфейс пользователя)
NP	—	неопределенные полиномиальные задачи и алгоритмы
RDF	—	Resource Description Framework
SW	—	Semantic Web

ВВЕДЕНИЕ

Спектр исследований в области искусственного интеллекта определился в США на знаменитом Дармутском семинаре 1956 г., положившем начало дисциплине ИИ. Практически одновременно зародились и два полярных подхода к данной проблеме: нейробионический и логический (символьный).

С первым подходом связаны имена У. МакКаллока, В. Питтса, Д. Хебба, Ф. Розенблата [17] — ученых, заложивших принципы «восходящего проектирования» и локальной самоорганизации в искусственном интеллекте (ИИ). В СССР развитие этого подхода в 1960—1970-е гг. было отражено в первую очередь в работах И.А. Амосова и его школы.

Символьный подход предполагает моделирование мышления в виде операций над символами (сравнение, поиск, модификация). Исходными элементами являются вербализуемые понятия предметной области, а направлением исследований — движение внутрь человеческой психики с выделением глубинных (подсознательных, интуитивных и неявных) структур опыта и мышления. Согласно гипотезе А. Ньюэлла [32] необходимое и достаточное условие, для того чтобы некоторая физическая система могла проявлять интеллектуальность, состоит в том, чтобы она являлась универсальной системой формальных манипуляций над конкретными символами.

В логическом подходе традиционно выделяют три направления [17].

Часть специалистов считает, что нужно найти свой способ решения задач на ЭВМ, который даст либо результат, подобный человеческому, либо даже лучше него. Специалисты такого типа неоднократно демонстрировали свое искусство по созданию программ такого рода.

Другие специалисты полагают, что поскольку естественный интеллект человека определяется главным образом его способностью обучаться при необходимости тому или иному виду творческой деятельности, то и искусственный интеллект должен не просто имитировать решение отдельных (пусть и творческих) задач, а ориентироваться на автоматическое создание программ для решения задач подобного рода.

Третья часть специалистов — это программисты, чьими руками реализуются программы решения задач ИИ. Они склонны рассматривать свою деятельность как новый виток в развитии программирования.

Начиная с 80-х гг. XX в., на основе исследований в области ИИ была сформирована новая отрасль компьютерной индустрии — разработка интеллектуальных систем (ИнС), областью применения которых яв-

ляются сложные из области NP [15] задачи, где логическая обработка информации превалирует над вычислительной. Это и задачи, решаемые в агрессивных средах, где необходимо принятие компетентного решения иногда в ограниченном временно́м диапазоне.

«Искусственный интеллект — это область информатики, которая занимается разработкой интеллектуальных компьютерных систем, т.е. систем, обладающих возможностями, которые мы традиционно связываем с человеческим разумом, — понимание языка, обучение, способность рассуждать, решать проблемы и т.д.» [19]. Данное определение продолжает оставаться актуальным и в настоящее время. При этом разработка программ, реализующих сложную статистическую обработку данных, не будет относиться к исследованиям в области искусственного интеллекта, какие бы сложные алгоритмы в них ни использовались. А вот создание программы порождения и проверки гипотез относится именно к этой области.

Современные ИнС подразделяются на два класса [4]: системы общего назначения и специализированные. Технология использования систем общего назначения состоит в следующем. Пользователь (эксперт) формирует знания (данные и правила), описывающие выбранную проблемную область. Далее на основе этих знаний, заданной цели, исходных данных и метапроцедур система генерирует и исполняет решение задачи. Данную технологию называют технологией систем, основанных на знаниях, или технологией инженерии знаний. Она позволяет специалисту, не обладающему профессиональными знаниями в области программирования, разрабатывать информационные прикладные приложения. В настоящее время инструментальными средствами общего назначения являются экспертные оболочки [12] и языки обработки знаний [2].

До недавнего времени при разработке специализированных ИнС использовалась технология традиционного процедурного программирования. Это позволяло обеспечивать их достаточно высокую эффективность. Однако использование традиционных программных средств разработки ИнС ограничивает их способность к изменению поведения при изменяющихся условиях, что достаточно важно при решении интеллектуальных задач. Использование технологии инженерии знаний при разработке интеллектуальных приложений позволило устранить данный недостаток.

А наиболее распространенным классом ИнС как специализированных, так и общего назначения, являются экспертные системы (ЭС).

Экспертные системы — это класс компьютерных систем, ориентированных на тиражирование опыта высококвалифицированных

специалистов-экспертов в таких слабоформализованных областях, где качество принятия решений традиционно зависит от уровня экспертизы (например, медицина, экономика, юриспруденция, геология, военное дело и т.д.). Напомним, что частным случаем понятия ЭС является понятие «система, основанная на знаниях» (СОЗ) как наиболее часто употребляемый в зарубежной литературе термин [7, 15].

Одним из направлений исследований в области ИИ является интеллектуальное программирование. Выбор инструментария разработки или инструментальной среды определяет успех создания проекта в целом.

Предметом рассмотрения в настоящем учебном пособии являются инструментальные средства создания ИнС и технологии работы с ними. Описываются язык и методология использования языка логического программирования Пролог и инструментальных оболочек различного назначения в рамках задач ИИ.

Экспертные оболочки KAPPA и CLIPS предназначены для построения СОЗ, иногда называемых базами знаний. Они могут служить иллюстрацией базовых концепций ИИ, используемых при создании СОЗ. При создании баз знаний (БЗ) в их среде происходит построение модели реальной системы. Студенты, только приступающие к освоению материала об экспертных системах и СОЗ, найдут в описании этих компьютерных программ все необходимое для разработки программ в более современных системах.

Иногда оболочки ЭС могут быть ориентированы (как следует из их описания) на работу с пользователями, не являющимися профессионалами в области программирования. Основным свойством таких оболочек является то, что они содержат все компоненты ЭС в готовом виде, и их использование не предполагает программирования, а сводится лишь к вводу формализованных знаний о проблемной области. Каждая оболочка характеризуется фиксированным способом представления знаний, организации вывода и функционирования компонентов, которые будут использоваться во всех приложениях, где она будет применяться.

Желание предоставить разработчику ЭС разнообразные средства для учета особенностей приложения привело к объединению в рамках одной компьютерной системы различных методов решения задач, представления и интерпретации знаний. В их состав могут входить средства модификации функционирования оболочки, набор компонентов, позволяющих конструировать собственные оболочки, средства комплексирования компонентов в виде встроенного языка, раз-

витый интерфейс пользователя. Подобные средства называют средствами автоматизации разработки (проектирования) ЭС. Они обычно позволяют представлять предметную область в виде сложной структуры, состоящей из произвольного числа понятий.

Необходимость выбора инструментальных средств, анализа их возможностей, модификации их функционирования требует от пользователя специальных знаний в области ИИ независимо от того, является он программистом или нет.

Среди производителей и разработчиков оболочек экспертных систем в группу лидеров попадают фирмы: Gensym (G2, Rethink), Intellicorp (KAPPA), Exsys (Exsys Professional), Национальное аэрокосмическое Агентство США (CLIPS — C Language Integrated Production System).

В настоящее время наблюдается постоянный рост интереса к исследованиям в области искусственного интеллекта. В первую очередь это связано с увеличением объемов обрабатываемой информации в расширяющемся информационном пространстве. Получают внедрение новые методы, разработанные в области технологии символических вычислений и теории представления знаний, хорошо сочетающиеся со статистическим и вероятностным подходами.

Развитие Интернет-приложений, в частности интеллектуальных агентов, облегчает извлечение информации при работе с различными электронными проблемными областями. Результаты исследований в области ИИ сегодня в значительной степени зависят от того, как исследователи преодолеют влияние традиционных концепций и сосредоточат усилия на задачах, решаемых в распределенной информационной среде. Например, задача многоагентной обработки информации в условиях пространственного распределения знаний о проблемной области является сложной. Использование онтологий для представления знаний позволяет установить семантические связи между элементами проблемной области и, как следствие, использовать известные методы ИИ.

Изучение инструментальной среды Protégé позволяет наглядно представить процесс создания онтологий. Система реализует объектно-ориентированный подход, имеет дополнительные возможности экспортирования онтологии в различные форматы, такие как CLIPS, HTML и т.п. Это является важным фактором при выборе системы для построения онтологий специалистами, занимающимися проблемами ИИ, семантического Web, разработкой программного обеспечения, системами разбора естественного языка, экспертными системами.

1. БАЗЫ ДАННЫХ, ОРИЕНТИРОВАННЫЕ НА ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

Термин «искусственный интеллект», безусловно, привлекает внимание каждого, хотя бы поверхностно интересующегося вычислительной техникой. Пока самым большим достижением в области ИИ можно считать «экспертные системы». В этой главе мы остановимся на некоторых особенностях их разработки.

В России исследования и разработки в области ЭС включены в ряд государственных и отраслевых научно-технических программ. Системы с базами знаний не только стали находить практическое применение в бизнесе и в решении серьезных информационных задач, но и продемонстрировали ощутимый эффект от их использования. Например, чрезвычайно эффективными с точки зрения применения ЭС оказались системы поддержки организационного управления и планирования распределения ресурсов. Основными областями их применения являются: медицина, электроника, вычислительная техника, геология, математика, космос, сельское хозяйство, управление, финансы, юриспруденция и т.д.

1.1. Экспертные системы и их особенности

Эксперты на основе собственного опыта или глубокого изучения проблемы осмысливают большое число факторов и устанавливают правила, их объясняющие. Выработанные ими правила упорядочиваются и хранятся в памяти ЭВМ. К ЭВМ обращаются за консультацией другие специалисты.

Но возможности экспертной системы шире. Подобно базе данных, она позволяет проводить поиск по ряду признаков одновременно и дает возможность оценить вероятность событий, которые могут использоваться в качестве условий поиска. Такие системы применяются на практике; например, ЭС MYCIN (середина 1970-х гг., Стэнфордский университет) ставила диагноз при инфекционных заболеваниях крови.

Приведем примеры других известных прикладных ЭС, ставших сегодня классическими:

- DENDRAL (середина 1960-х гг., Стэнфордский университет) — ЭС расшифровки данных масс-спектрографического анализа;
- PROSPECTOR (1974—1983 гг., Стэнфордский университет) — ЭС обнаружения полезных ископаемых;
- SOPHIE — ЭС обучения диагностированию электрических цепей;
- XCON — ЭС конфигурирования оборудования системы VAX;
- PALLADIO — ЭС проектирования и тестирования СБИС;
- JUDITH — ЭС оказания помощи специалистам по гражданским делам и юристам, предлагающая различные варианты подходов к разрешению дела на основе его фактических и юридических предпосылок;
- LRS — ЭС оказания помощи в подборе и анализе информации о судебных решениях и правовых актах в области кредитно-денежного законодательства, связанного с использованием векселей и чеков;
- «Ущерб» — созданная на основе российского трудового законодательства ЭС, обеспечивающая юридический анализ ситуации привлечения рабочих и служащих к материальной ответственности при нанесении предприятию материального ущерба.

Многие из систем сегодня получили развитие, например HEARSAY, HEARSAY-2, HEARSAY-3, AGE [18]. Первые две системы этого ряда являются развитием интеллектуальной системы распознавания слитной человеческой речи, слова которой берутся из заданного словаря. Они отличаются оригинальной структурой, основанной на использовании доски объявлений — глобальной базы данных, содержащей текущие результаты работы системы. В дальнейшем на основе этих систем были созданы инструментальные системы HEARSAY-3 и AGE (Attempt to Generalize — попытка общения) для построения ЭС.

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

Среди современных коммерческих систем хочется выделить экспертную систему-оболочку G2 американской фирмы Gensym (США) [16] как непревзойденную экспертную коммерческую систему для работы с динамическими объектами. Работа в реальном масштабе вре-

мени с малыми интервалами ответа необходима при анализе критических ситуаций, возникающих в корпоративных информационных сетях, на атомных реакторах, в космических полетах и других задачах, требующих принятия решения в течение миллисекунд с момента их возникновения.

Многие предприятия используют ЭС для принятия решений в таких областях, как торги на фондовой бирже, автоматическое понимание новостей, кредитный анализ, управление рисками, построение портфелей кредитов и инвестиций, оценка рейтинга банков, автоматизация аудита, предсказание изменений на финансовом рынке и др.

Приложением теории экспертных систем в экономике является использование их для организации бизнес-процесса реинжиниринга (БПР).

БПР — это анализ деятельности компании в целях выявления ее слабых мест и создание на основе его результатов максимально эффективной модели функционирования предприятия.

В практике построения экспертных систем используются следующие технологии искусственного интеллекта:

- набор инструментов программирования для представления знаний;
- стратегии обработки знаний, т.е. их преобразования, представления и управления их применением в процессе решения задач в рамках проблемной области;
- методологии проектирования, обеспечивающие создание экспертных систем, «прозрачных» как для пользователя, так и для инженеров по знаниям, сопровождающих систему в процессе эксплуатации.

Эти технологии оформлены в виде блоков-компонентов, из которых формируется архитектура экспертной системы (рис. 1.1).

Если данные существуют на жестком носителе в БД, то знания существуют только в рабочей области памяти в момент функционирования системы. В базе знаний представлены описания знаний (каркасы знаний) — модели, которые, попав в рабочую область памяти, заполняются текущими данными из БД прикладной системы или информационного приложения (ИП). Таким образом осуществляется наполнение блока логического вывода для формирования рекомендаций и комментариев пользователю.

Современные экспертные системы основываются на знаниях экспертов, специалистов в заданной проблемной области. Программа работает с базой данных, где собраны различные факты, статистические

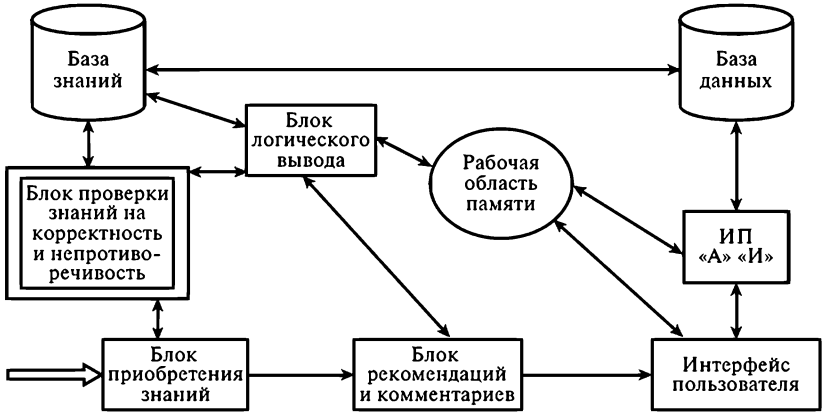


Рис. 1.1. Архитектура экспертной системы

данные, и извлекает из информации, хранимой в базе знаний, набор правил. Правила могут быть сформулированы в виде суждений на языке, близком к естественному, на основе которых можно принять решение или выработать его автоматически.

Например, можно ожидать, что база знаний, в которой хранятся сведения о торговых операциях фирмы, просмотрит записи о клиентах и попытается выявить характеристики потенциальных задолжников. А с помощью базы знаний, ориентированной на медицинские исследования, можно попытаться прогнозировать исход сердечных приступов или причины острых инфекционных заболеваний.

Во многих случаях базы знаний основываются на информации, собираемой для решения обычных задач. Вывод правил на основе этой информации проводится параллельно обработке данных. В других случаях данные будут собираться в процессе научного исследования. Подходы к проектированию БЗ и ЭС включают ряд альтернатив: от классического статистического анализа, байесовского логического анализа, в котором используются не только ответы типа «да» и «нет», «истинно» или «ложно», до теории вероятности с генерацией правил вывода; нечеткая логика; нейросетевые и эволюционные технологии.

Таким образом, экспертные системы, или системы, основанные на знаниях, предназначены для решения плохо или слабо формализованных задач. Трудно формализуемые задачи обладают ошибочностью, неполнотой, неоднозначностью и противоречивостью как исходных данных, так и знаний о решаемой задаче.

ЭС используются там, где существует враждебная человеку среда, отсутствует алгоритм решения задачи или для решения задачи требуется достаточно много времени (машинного) и трудоемкий алгоритм либо имеется недостаток в числе экспертов для решения поставленной задачи.

Эксперт — это человек, являющийся профессионалом высокой квалификации в проблемной области, для которой предназначена разработка экспертной системы. Его знания лежат в основе системы.

ЭС разрабатывается в том случае, если ее разработка, во-первых, необходима, во-вторых, оправдана и задача, которую предполагается решать с ее помощью, вполне под силу эксперту-человеку.

Чаще всего экспертные системы используются, как правило, для решения так называемых NP-задач [15]. ***NP-задачи*** — это недетерминированные полиномиальные задачи, которые могут не сойтись при конечном количестве итераций. К ним относятся слабо формализованные или плохо структурированные задачи, а также задачи, для которых может не существовать точного решения. Подобные задачи призваны решать проблемы в условиях неполной, нечеткой или недостоверной информации, а также при достаточно большом объеме обрабатываемых данных, т.е. при угрозе комбинаторного взрыва.

Работающий совместно с экспертом специалист по инженерии знаний, выявляющий и формализующий экспертные знания, называется когнитологом. Часто когнитолога называют также инженером по знаниям.

Дополним определение.

Экспертная система — это система, основанная на знаниях о заданной проблемной области, в которой знания слабо структурированы, решаются сложные NP-задачи, осуществляется взаимодействие с естественным языком на основе рассуждений и комментирования своих действий с целью обучения пользователя при самообучении системы.

Структура экспертной системы должна быть «прозрачна» для конечного пользователя. Конечными пользователями экспертной системы могут являться и необученный пользователь, и эксперт в заданной предметной области, и прикладной программист, и когнитолог.

Отличительными особенностями разработки экспертных систем в настоящее время являются использование естественного языка и объектно-ориентированное представление информации.

1.2. Основные типы задач, решаемых с помощью экспертных систем

Области применения систем, основанных на знаниях, могут быть сведены к нескольким основным типам.

Интерпретация — это процедура анализа данных с целью определения их смысла. Интерпретатор должен быть в состоянии обрабатывать информацию, представленную частично, и выдвигать гипотезы о доверии данным. При ненадежных данных интерпретация также будет ненадежной, поэтому для достижения доверия необходимо определить, какая информация была неточной или неопределенной. Так как цепочки рассуждений в ИнС могут быть достаточно длинными, интерпретатору необходимо располагать средствами объяснения того, как интерпретация обусловлена имеющимися данными.

Интерпретирующие системы обладают способностью получать определенные заключения на основе результатов наблюдения. Например, система PROSPECTOR — одна из наиболее известных систем интерпретирующего типа — объединяет знания девяти экспертов. Используя сочетания девяти методов экспертизы, система обнаружила залежи руды стоимостью около 1 млн дол., причем наличие этих залежей не предполагал ни один из девяти экспертов. Другая интерпретирующая система — HASP/SIAP — позволяет определять местоположение и типы судов в Тихом океане по данным акустических систем слежения.

Планирование, или процедура составления планов. Планирующие системы предназначены для достижения конкретных целей при решении задач с большим числом переменных. Планирование должно быть условным, т.е. иметь возможность коррекции при поступлении новых сведений. Процедура планирования должна предусматривать возможность совершать пробные шаги, сравнивать различные варианты, планы, а также уметь сосредотачивать внимание на наиболее важных гипотезах и работать в условиях неопределенности.

В качестве примеров таких систем можно привести: ЭС составления перспективного плана капиталовложения компании; ЭС планирования потребности материалов и комплектующих для основного производства предприятия на определенный период, а также упомянутую выше экспертную систему XCON для конфигурирования компьютерных систем типа VAX.

Прогнозирование — это определение хода событий в будущем на основании модели прошлого и настоящего. Прогнозирующие системы пред-

сказывают возможные результаты или события на основе данных о текущем состоянии объекта. Прогнозирование действий осуществляется на определенный (заданный) промежуток времени. Ключевой проблемой задачи является требование соединения в единое целое неполной имеющейся информации. Прогнозирование должно рассматривать различные варианты будущего и указывать их чувствительность к изменению входных данных. Решаемая задача прогнозирования должна носить условный характер, поскольку вероятность определенных событий в будущем будет зависеть от более близких, но не предсказуемых событий.

В то же время, например, прогнозирование местоположения Юпитера через два года после ближайшего четверга в случае имеющейся полной информации к задаче ИИ не относится.

Мониторинг — это непрерывное оповещение о состоянии системы или процесса. Это непрерывная интерпретация сигналов и выдача оповещений при возникновении ситуаций, требующих вмешательства.

Проектирование — это использование экспертной системы для исключения профессионала из задачи проектирования или выполнение рутинных действий по обработке информации в конкретной прикладной системе.

Ключевыми проблемами проектирования являются:

- отсутствие исчерпывающей информации, позволяющей увязать ограничения проектирования с принимаемыми решениями;
- взаимодействие подзадач (одна оказывает влияние на другую);
- умение видеть картину в интересах ухода из локально-оптимальных точек в пространстве проекта;
- оценивание последствий принимаемых решений.

В таких ЭС структуры представления информации, отражающие сведения о круге проектируемых объектов, представляются в виде морфологических таблиц и семантических сетей, имеющих в частном случае вид и (или) деревьев, гиперграфов и наборов продукций.

Диагностика — это процесс поиска неисправностей в системе или определение стадии заболевания в медицине, основанный на интерпретации данных (возможно, зашумленных). Такие системы используются для установления связи между нарушениями деятельности в системе или организме и их возможными причинами. К основным проблемам, возникающим при решении подобных задач, можно отнести: недоступность или малодоступность некоторых данных; сочетание не вполне совместимых частных моделей объектов или процессов; наложение симптомов других неисправностей (в некоторых ИП этой проблемой пренебрегают).

Наиболее известна диагностическая система MYCIN, которая предназначена для диагностики и наблюдения за состоянием больного при бактериальных инфекциях. Ее первая версия была разработана в Стэнфордском университете в середине 1970-х гг. 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. Диагностические ЭС незаменимы при ремонте механических и электрических машин и при устранении неисправностей и ошибок в аппаратном и программном обеспечении компьютеров.

Обучение рассматривается в ЭС в двух аспектах: обучение пользователя и самообучение системы, причем как на этапе приобретения знаний, так и в процессе работы ИП.

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. Одним из примеров обучающих ЭС является разработанная Д. Ленатом система EURISCO [30], которая использует простые эвристики.

Большинство ЭС 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.

Контроль и управление — это системы, основанные на знаниях, которые могут применяться в качестве контролирующих и обеспечивающих поддержку принятия решений, анализируя данные, поступающие от нескольких источников. Системы подобного типа широко используются на атомных электростанциях, в управлении воздушным движением и медицинском контроле. Также они могут быть полезны при регулировании финансовой деятельности предприятия и в процессе выработки решений в критических ситуациях. Такие системы обычно называют системами поддержки принятия решений [2]. В большей степени в них используются статистические методы обработки информации и интеллектуальный анализ данных.

В целом для задач, решаемых в среде ИнС, характерны большие пространства представления и поиска решений, условные рассуждения, изменяющиеся во времени, и зашумленные данные.

1.3. Особенности разработки экспертных систем

В настоящее время сложилась определенная последовательность разработки ЭС, которая включает следующие этапы: приобретение знаний, представление знаний, реализацию.

1.3.1. Приобретение знаний

Приобретение знаний для системы, основанной на знаниях, осуществляется у эксперта в заданной прикладной области когнитологом (инженером по знаниям).

Известны следующие способы приобретения знаний:

- наблюдение за работой эксперта;
- опрос эксперта;
- интервьюирование и др.

Приобретение знаний может осуществляться на основе:

- четко представленной информации в виде конкретных значений данных и описывающих их понятий;
- информации, представленной в виде таблиц, графиков, гистограмм;
- слабо структурированной информации, представленной в виде эвристик эксперта в заданной проблемной области.

Функции когнитолога могут выполнять человек, программа распознавания естественного языка или экспертная система.

На этапе приобретения знаний осуществляются функции:

- описание (данных, ситуаций и т.д.);
- концептуализация — описание объектов, свойств, рассуждений;
- формализация — представление информации в заданных структурах;
- реализация — разработка информационного приложения, осуществляемая с проверкой корректности и непротиворечивости поступающей информации.

1.3.2. Представление знаний

Представление знаний заключается в последовательной реализации следующих этапов.

На первом этапе осуществляется описание проблемной области с помощью таксономий или онтологий [1].

Затем необходимо определить факты, правила и отношения для заданной проблемной области. Под фактами понимаются описания значений данных, правила служат для интеграции фактов в каузальной форме в целях получения решения, а отношения представляют собой совокупность семантических зависимостей, используемую для интерпретации правил в стратегиях поиска решений.

На следующем этапе на основании определенных ранее фактов, правил и отношений осуществляется идентификация знаний, причем:

- факты и правила составляют главные компоненты знания;
- факты, правила и отношения определяют стратегию поиска решения;
- декларативные и процедурные знания выражаются в виде цепочки логических рассуждений в стратегиях поиска решения и лежат в основе организации логического вывода.

Очередной этап связан с разработкой моделей представления знаний на основе одного из известных подходов:

- фреймы;
- семантические сети;
- исчисление предикатов первого порядка;
- продукции;
- комбинированные модели на основе правил продукции.

Процедура лингвистической обработки проблемной информации реализует еще один этап представления знаний и заключается в формулировании предложений (совокупности фактов) и утверждений (совокупности предложений).

Следующим этапом является организация логического вывода, определяемого в виде результата совокупного применения стратегий и цепочек логических рассуждений. Цепочки логических рассуждений представляют собой интерпретацию фактов предметной области, причем может быть использована как прямая, так и обратная цепочка логического вывода. На основе цепочек логических рассуждений строится соответствующая стратегия поиска решения.

На заключительном этапе осуществляется оценка неопределенности в системах ИИ, которая может быть реализована на основе:

- коэффициентов неопределенности;
- вероятностных характеристик событий;
- факторов логической необходимости и логической достаточности событий;
- формулы Байеса для оценки вероятности событий;

- лингвистической переменной;
- переменной неопределенности.

1.3.3. Реализация

Реализация ИнС предполагает выбор инструментальных средств. При этом уже определены характеристики задачи, решаемой проектируемой экспертной системой, имеются структура, пространство поиска и пространство формируемых решений, требуется обеспечить необходимые функциональные возможности инструментального комплекса или среды разработки.

Для реализации ЭС используются как традиционные языки программирования, языки обработки списков, логического программирования, процедурные языки, так и программные оболочки (среды) для создания экспертных систем, экспертные системы и системы приобретения знаний.

Применяются следующие технологии:

- объектно-ориентированный подход к разработке экспертных систем;
- системы с распределенной архитектурой (клиент-сервер);
- технология реинжиниринга [16] или моделирования процессов.

1.4. Виды экспертных систем

По степени соответствия реальному масштабу времени различают статические и динамические экспертные системы (см. рис. 1.1).

Статические системы используют постоянную информацию из базы данных и неизменные правила в базе знаний. Данные и знания не могут быть добавлены в процессе работы ИП. ЭС данного типа разрабатываются, как правило, для приложений, в которых нет необходимости учитывать влияние окружающей среды.

Динамические системы, напротив, ориентированы на работу с постоянно меняющейся информацией в БД и в БЗ системы или ИП.

Современные ЭС обычно сочетают в себе оба указанных класса, реализованных в виде соответствующих режимов работы. Например, в зависимости от вида работы, архитектура экспертной системы будет просто включать различные функциональные элементы.

Так, в статическом режиме система использует (см. рис. 1.1):

- рабочую область памяти;

- блок логического вывода, осуществляющий сопоставление данных и знаний, которые требуются лишь на заданном промежутке времени;
- базу знаний, содержащую правила логического вывода, не меняющиеся на заданном промежутке времени и представленные в установленных для системы формализмах (фреймах, семантических сетях, предикатах первого порядка и др.);
- базу данных, содержащую постоянные, не меняющиеся на заданном промежутке времени данные, необходимые для работы БЗ;
- блок приобретения знаний, предназначенный для работы с экспертом и когнитологом;
- блок рекомендаций и комментариев;
- интерфейс пользователя, предназначенный для взаимодействия с пользователем и отображающий состояние рабочей памяти на основе рекомендаций и комментариев;
- блок проверки знаний на корректность и противоречивость, выполняющий согласование вновь поступающих в базу правил с уже имеющимися в ней.

Динамический режим работы системы предполагает использование:

- рабочей памяти;
- блока логического вывода;
- базы данных;
- базы знаний;
- блока рекомендаций и комментариев;
- интерфейса пользователя;
- информационного приложения, работающего в автоматическом («А») или интерактивном («И») режиме.

Кроме этого, имеется система датчиков взаимодействия с внешней средой и информационная база ИП.

1.5. Представление знаний в системах искусственного интеллекта

1.5.1. Данные и знания

Данные представляют собой описания и значения свойств объектов, представленных в заданной форме, например в виде значений констант и переменных.

Знания — это представленная на естественном языке в соответствии с предметной областью совокупность данных, фактов и правил вывода, включающая информацию о свойствах объектов, закономерностях процессов и явлений, а также правилах использования этой информации в целях констатации фактов или принятия решений и реализующая свойства означивания, активности, интерпретируемости.

Правила, соединенные в тексте программы логическими операторами & (и), or (или), not (не), образуют предложения.

Совокупность предложений, не связанных операторами &, or, not, называется утверждением и характеризуется пространством утверждения (рис. 1.2).

Утверждения задают процедуры обработки. Имя утверждения является идентификатором процедура обработки. Следует различать декларативные («знать, что») и процедурные («знать, как») знания. Декларативное знание — это констатация факта, комментирование события, а процедурное — порядок выполнения процедуры, определяемый располагаемыми данными.

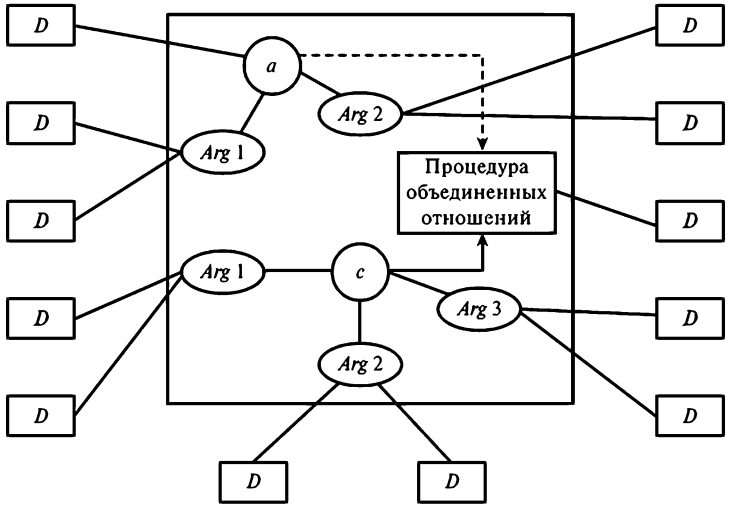


Рис. 1.2. Структура пространства утверждения:
a, *c* — имена отношений; *D* — действие

Декларативные знания — это факты, правила, отношения.

Процедурные знания — это правила, отношения и процедуры.

База знаний — это вместилище знаний, используемых для получения решений и представленных на основе одной или несколь-

ких моделей представления знаний, которые используются в цепочках логического вывода прикладных систем искусственного интеллекта.

Означивание — это свойство знаний, характеризующее задержанный процесс, инициирующий организацию логического вывода.

Отношение — это наличие семантической зависимости между конкретными значениями данных с параметрами процедуры.

Интерпретация — это свойство знаний, характеризующее соответствие фактов предметной области правилам логического вывода. Кроме того, интерпретация — это наиболее общий анализ данных в целях определения их смысла.

Факты — это значения данных или описание понятий предметной области, предназначенные для интерпретации в правилах логического вывода.

Правила — это отношения каузальной (причинно-следственной) зависимости между фактами предметной области и получаемыми решениями в прямой или обратной цепочках логического вывода.

Таксономическая схема — это иерархическая классификационная схема понятий проблемной области, предназначенная для описания состава информационных единиц разрабатываемого приложения.

Онтология — это совокупность формальных описаний, включающая:

- классы — понятия и отношения между ними в рассматриваемой предметной области;
- слоты — свойства каждого понятия, описывающие различные атрибуты понятия (роли или свойства);
- фацеты — ограничения, накладываемые на слоты (ограничения ролей).

Онтология вместе с набором индивидуальных экземпляров классов образует базу знаний. В действительности провести четкую границу между онтологией и базой знаний сложно.

Последовательность разработки информационного приложения с базой знаний можно представить следующим образом:

- определение назначения системы и цель разработки;
- определение функций ИИ, которые необходимо реализовать;
- формирование решения о способе представления необходимой проблемной информации (словесно или графически);
- определение фактов, данных, свойств, отношений и процессов в структуре таксономической схемы;
- выбор модели представления знаний;

- организация логического вывода;
- представление и выявление неопределенностей;
- выбор инструментальных средств разработки приложения.

1.5.2. Представление знаний в рабочей памяти ЭВМ

Представление знаний в рабочей памяти складывается из двух компонентов:

- текущих значений данных;
- агенды (списка активных конфликтующих в данный момент правил).

В соответствии с текущей ситуацией утверждение на основе значений данных (пространства утверждения) настраивается на текущую ситуацию. В соответствии с этим приобретают смысл отношения пространства утверждения (см. рис. 1.2), формирующие агенду.

Пространство утверждения содержит совокупность релевантных (соответствующих) правил для выполнения заданной в утверждении процедуры, которая реализуется с помощью внешних ссылок пространства на значениях данных.

1.5.3. Представление знаний в базе знаний

Интеллектуальность экспертной системы оценивается на основе релевантности агенды. Релевантность заключается в выборе необходимых правил для принятия решения.

Релевантность обеспечивается:

- используемой технологией. Например, при использовании ООП вся совокупность объектов подразделяется на классы, а выбор знаний из базы осуществляется на основе определения требуемых объектов и происходит в два этапа:
  - на первом этапе искомый объект сопоставляется с объектами, хранящимися в базе знаний (например, по имени утверждения — см. рис. 1.2);
  - на втором этапе выбор объекта уточняется на основе значений данных, содержащихся в рабочей памяти и во внешних ссылках пространства утверждения. Если значения данных совпадают, то выбранный объект может быть использован для данного утверждения:
- механизмом доступа к базе знаний;
- механизмом сопоставления с образцами, хранящимися в БЗ.

Механизм сопоставления может быть:

- синтаксическим, сопоставляющим выделенные значения данных со значениями данных пространства утверждения по написанию (значению);
- параметрическим, сопоставляющим выделенные значения данных со значениями пространства утверждения по параметрам (понятиям) предметной области;
- функциональным, сопоставляющим заданные значения данных по выполняемым функциям;
- принудительным, при котором механизм сопоставления назначается разработчиком экспертной системы.

Контрольные вопросы

1. Дайте определение системы искусственного интеллекта.
2. В каких случаях оправдано создание системы искусственного интеллекта?
3. Охарактеризуйте понятие «знание».
4. В чем заключаются различия между декларативными и процедурными знаниями?
5. Назовите архитектурные компоненты экспертных систем.
6. Сравните возможности инструментальных средств разных классов — оболочек, языков программирования высокого уровня и многофункциональных сред — применительно к проектированию ЭС для задач планирования, проектирования и интерпретации.

2. ФОРМАЛИЗАЦИЯ ЗНАНИЙ О ПРОБЛЕМНОЙ ОБЛАСТИ

2.1. Таксономическая классификационная схема

Описание проблемной области начинается с представления информации обо всех объектах, классах, объединенных объектах и понятиях, их определяющих, а также данных, которые могут потребоваться в разрабатываемом приложении. Для формализованного представления этой информации используется таксономическая классификационная схема понятий, или таксономия (от греч. *taxis* — порядок, чередование, следование и *nomos* — закон).

Таксономия используется для упорядочивания информации в ИнС. Она имеет вид иерархического дерева (рис. 2.1), ветвями которого являются обобщенные понятия, соответствующие рассматриваемой проблемной области. В корне таксономической схемы лежит основное понятие, описывающее проблемную область. Основное понятие таксономической схемы не должно иметь предшествующих понятий.

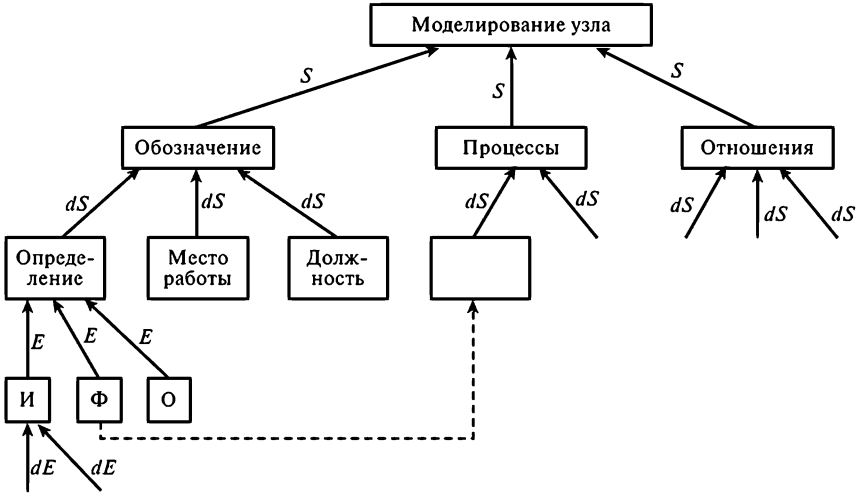


Рис. 2.1. Таксономия

Перечень понятий утверждается разработчиком системы или когнитологом. Таксономия является структурой из трех компонентов:

1) элементов;
2) дуг;
3) подсетей/узлов.

Кроме этого, каждая дуга в таксономической схеме определяет назначение понятия. Дуга S является множеством. Дуга E является составным элементом, dS является подмножеством, а dE —составной частью элемента.

При описании предметной области с помощью таксономической схемы должны быть представлены все понятия (рис. 2.2), формирующие пространство утверждения для стереотипных ситуаций. Описания ситуаций с помощью одних и тех же понятий образуют класс.

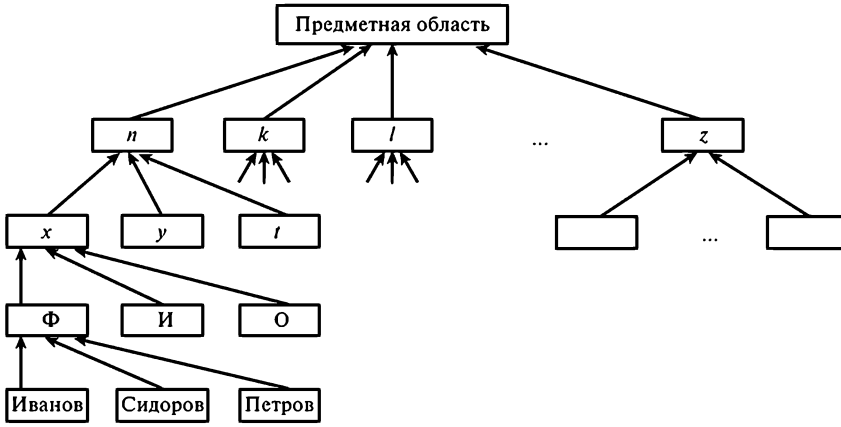


Рис. 2.2. Фрагмент таксономической схемы

Каждый узел или каждая подсеть могут функционировать в таксономической области с описывающими их понятиями. На основе выделенных в таксономической схеме отношений, ориентированных на заданную предметную область, строятся правила вывода (продукции). Каждый узел таксономической схемы представляет собой понятие, которое конкретизируется, уточняется и дополняется понятиями нижних уровней, причем каждое понятие более низкого уровня наследует все свойства понятий верхнего уровня и добавляет свои. Узел таксономической схемы отражает отношение между объектами и понятиями (рис. 2.3).

Основные виды отношений, используемых в системе ИИ, также могут быть представлены в виде таксономии:

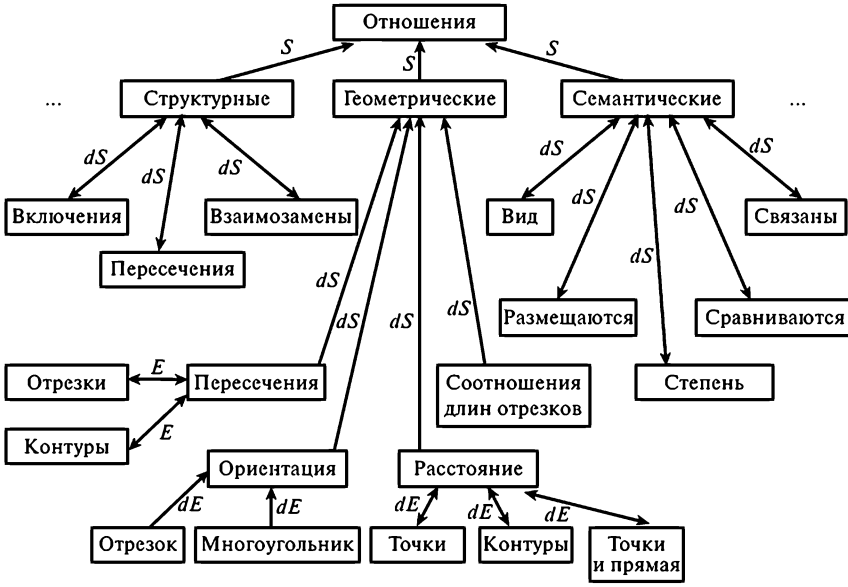


Рис. 2.3. Фрагмент таксономии отношений

1) каузальные:
   а) причина и следствие (прямая цепочка логических рассуждений);
   б) следствие и причина (обратная цепочка логических рассуждений);
2) логико-арифметические:
   а) арифметические;
   б) логические;
3) теоретико-множественные:
   а) отношение принадлежности (часть целого или целое части);
   б) классификационные:
      - вид → род (быть подмножеством);
      - род → вид (супермножество);
4) характеристические (присущие всем БД):
   а) атрибутивные:
      - иметь свойство;
      - быть свойством;
   б) идентифицирующие:
      - быть именем;
      - иметь имя;

5) квантифицирующие:
   а) иметь количественное значение;
   б) иметь лингвистическое значение;
6) динамические:
   а) изменить положение;
   б) изменить ориентацию (на 90°);
7) временные:
   а) быть раньше;
   б) быть позже;
   в) быть одновременно;
8) пространственные:
   а) быть расположенным;
   б) расстояние между;
9) инструментальные:
   а) выполняться посредством;
   б) выполняться с помощью.

Корнем таксономической схемы является основной узел. Таксономическая классификационная схема определяет совокупность узлов подсетей и элементов. Уточним назначения ее дуг. Дуги имеют заданные значения.

Дуги типа S/dE определяют назначение текущего узла, как «является множеством» /«состоит из элементов». Дуги dS/E определяют назначение узла, как «является подмножеством» /«является составным элементом».

Таким образом, каждая ветвь таксономической схемы может представлять собой самостоятельную таксономию в зависимости от детализации области представления информации. При этом в корень нового образующего дерева ставятся участвующие в экспертизе понятия.

Каждый объект представляется как в таксономии, так и в утверждении двумя группами характеристик:

- группа I определяет внутреннюю структуру объекта, т.е. отношения со своими аргументами в пространстве утверждения;
- группа II определяет понятие пространства утверждения на основании внешних ссылок пространства, служащих для экспертизы.

Соответственно изначально выбор и сопоставление осуществляется на основе внутренней структуры объектов, а затем на основе внешних ссылок пространства.

2.2. Онтологический подход к представлению проблемной информации

В настоящее время онтологии являются центральными компонентами большинства крупных компьютерных информационных приложений.

Онтология — это формальное описание понятий предметной области и отношений между ними в рассматриваемой предметной области, свойств каждого понятия, описывающих различные атрибуты понятия, а также ограничений, наложенных на слоты.

Онтология вместе с набором индивидуальных экземпляров классов образует базу знаний. В центре большинства онтологий находятся классы. Классы описывают понятия предметной области.

Онтологии в сети варьируются от больших таксономий, категоризирующих веб-сайты (как на сайте Yahoo!), до категоризаций продаваемых товаров и их характеристик (как на сайте Amazon.com). Консорциум WWW разрабатывает специальный формат представления информации RDF (Resource Description Framework, Brickley & Guha, 1999) и язык кодирования знаний на веб-страницах, чтобы сделать их понятными для электронных агентов, осуществляющих поиск информации. Совокупность разработанных онтологий представлена на сайте swoogle.com.

Разрабатываемые в настоящее время стандартные онтологии могут использоваться экспертами по предметным областям для совместного использования и аннотирования информации в своей области. Например, в области медицины созданы большие стандартные структурированные словари.

Онтология определяет общий словарь для ученых, которым необходимо совместно использовать информацию о предметной области. Она включает машинно-интерпретируемые формулировки основных понятий предметной области и отношения между ними.

Распространение онтологического подхода к представлению знаний оказало влияние на создание разнообразных языков представления онтологии и инструментальных средств, предназначенных для их редактирования и анализа.

Существуют традиционные языки спецификации онтологий [23, 24]: Ontolingua, CycL, а также языки, основанные на дескриптивных логиках (LOOM) и фреймах (OKBC, OCML, Flogic). Более поздние языки основаны на Web-стандартах (XOL, SHOE, UPML). Специально для обмена онтологиями через Web были созданы языки RDF (S), DAML, OIL, OWL. Semantic Web базируется на модели данных Resource Description Framework (RDF) [29].

2.2.1. Цели разработки онтологий

Часто онтология предметной области сама по себе не является целью. Разработка онтологии соответствует определению набора данных и их структуры для использования другими программами. Методы решения задач, доменно-независимые приложения и программные агенты используют в качестве данных онтологии и базы знаний, построенные на их основе.

Онтологии разрабатываются в целях:

- совместного использования людьми или программными агентами для общего понимания структуры информации;
- обеспечения возможности повторного использования знаний в предметной области;
- возможности явных допущений в предметной области;
- отделения знаний в предметной области от оперативных знаний;
- анализа знаний в предметной области.

При необходимости создания большой онтологии можно интегрировать несколько уже существующих онтологий (рис. 2.4).

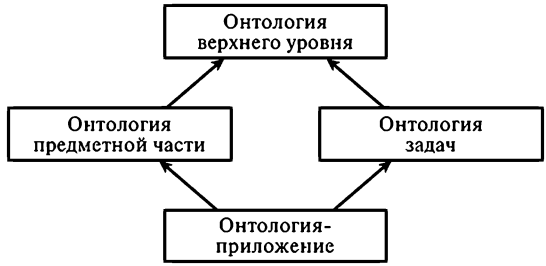


Рис. 2.4. Многоуровневая схема отношений между онтологиями

Разработка онтологий отличается от проектирования классов и отношений в объектно-ориентированном программировании. Объектно-ориентированное программирование сосредоточено главным образом на методах классов. При этом программист принимает проектные решения, основанные на операторных свойствах класса, тогда как разработчик онтологии принимает эти решения, опираясь на структурные свойства класса. В результате структура класса и отношения между классами в онтологии отличаются от структуры подобной предметной области в объектно-ориентированной программе.

Единой методологии разработки онтологий на сегодняшний день не существует. На практике такая разработка включает:

- определение классов в онтологии;
- расположение классов в таксономической иерархии (подкласс — надкласс);
- определение слотов и описание допускаемых значений этих слотов;
- заполнение значений слотов-экземпляров.

После этого можно создать базу знаний, определив отдельные экземпляры этих классов, введя в определенный слот значение и дополнительные ограничения для слота.

2.2.2. Фундаментальные правила разработки онтологии

Перечислим фундаментальные правила разработки онтологии.

1. Не существует единственно правильного способа моделирования предметной области — всегда найдется жизнеспособная альтернатива. Лучшее решение почти всегда зависит от предполагаемого приложения и ожидаемых расширений.

2. Разработка онтологии — это обязательно итеративный процесс.

3. Понятия в онтологии должны быть близки к объектам (физическим или логическим) и отношениям в интересующей вас предметной области. Скорее всего, это существительные (объекты) или глаголы (отношения) в предложениях, которые описывают вашу предметную область, т.е. знание того, для чего вы собираетесь использовать онтологию и насколько детальной или общей она будет, повлияет на многие решения, касающиеся моделирования. Среди нескольких жизнеспособных альтернатив нам нужно определить, какая поможет лучше решить поставленную задачу и будет более наглядной, более расширяемой и более простой в обслуживании.

Необходимо также помнить, что онтология — это модель реального мира и понятия в ней должны отражать эту реальность. После того как мы определим начальную версию, можем оценить и отладить ее, используя ее в приложениях или в методах решения задач и (или) обсудив ее с экспертами предметной области. Как правило, в результате начальная версия онтологии будет пересмотрена. Этот процесс итеративного проектирования, вероятно, будет продолжаться в течение всего жизненного цикла онтологического описания предметной области.

2.2.3. Определение области и масштаба онтологии

Разработку онтологии следует начинать с определения ее области и масштаба. При этом необходимо ответить на основные вопросы.

1. Какую область будет охватывать онтология?

2. Для чего будет использоваться данная онтология?

3. На какие типы вопросов должна давать ответы информация в онтологии?

4. Кто будет использовать и поддерживать онтологию?

Ответы на эти вопросы могут измениться во время процесса проектирования онтологии, но в любой заданный момент времени они помогают ограничить масштаб модели.

Одним из способов определения масштаба онтологии является составление списка вопросов, на которые должна ответить база знаний, основанная на онтологии, т.е. так называемые вопросы для проверки компетентности.

Эти вопросы позволят определить: содержит ли онтология достаточно информации для ответа на эти типы вопросов; требуется ли для ответов особый уровень детализации или представление определенной области. Вопросы для проверки компетентности являются формальными и не должны быть исчерпывающими в обязательном порядке.

2.2.4. Рассмотрение вариантов повторного использования существующих онтологий

Почти всегда стоит учитывать опыт уже имеющихся наработок в интересующей нас конкретной предметной области и попытаться улучшить и расширить ее описание в целях решения поставленных задач. Повторное использование существующих онтологий может быть необходимым, если вновь разрабатываемой системе необходимо будет взаимодействовать с другими приложениями, которые уже вошли в отдельные онтологии или контролируемые словари. Многие онтологии уже доступны в электронном виде и могут быть импортированы в используемую среду проектирования новой онтологии. Формализм онтологии часто не имеет значения, так как многие системы представления знаний могут импортировать и экспортировать онтологии. Даже если конкретная система представления знаний не может работать напрямую с отдельным формализмом, то задача перевода онтологии из одного формализма в другой обычно не является сложной.

2.2.5. Перечисление важных терминов в онтологии

Полезно составить список всех терминов, о которых мы хотели бы сказать что-либо или которые хотели бы объяснить пользователю. Какие термины мы хотели бы рассмотреть? Какие свойства имеют эти термины? Что мы хотели бы сказать об этих терминах?

В первую очередь важно получить полный список терминов, не беспокоясь о пересечении понятий, которые они представляют, об отношениях между терминами, о возможных свойствах понятий или о том, чем являются понятия — классами или слотами.

Следующие два шага — разработка иерархии классов и определение свойств понятий (слотов) — тесно переплетены между собой, и выполнить их в отдельности сложно. Обычно в иерархии дается несколько формулировок понятий, а затем описываются свойства этих понятий и т.д. Эти два шага являются самыми важными в процессе проектирования онтологии. В этом разделе они будут описаны лишь вкратце, а более сложные проблемы, которые необходимо принять во внимание, будут рассмотрены далее.

2.2.6. Определение классов и их иерархии

Существует несколько возможных подходов для разработки иерархии классов.

Процесс нисходящей разработки начинается с определения самых общих понятий предметной области с последующей конкретизацией понятий. Например, мы можем начать с создания классов для общих понятий «Вино» и «Еда». Затем конкретизируем класс «Вино», создавая его подклассы: «Белое вино», «Красное вино», «Розовое вино». Мы можем еще дальше категоризировать класс «Красное Вино», например: Syrah, Red Burgundy, Cabernet Sauvignon и т.д.

Процесс восходящей разработки начинается с определения самых конкретных классов, листьев иерархии, с последующей группировкой этих классов в более общие понятия. Например, сначала можно определить классы для вин Pauillac и Margaux, а затем создать общий надкласс для двух этих классов — Medoc, который в свою очередь будет подклассом Bordeaux.

Процесс комбинированной разработки представляет собой сочетание нисходящего и восходящего подходов и подразумевает определение в первую очередь наиболее значимых понятий, которые затем соответствующим образом обобщаются и ограничиваются.

Ни один из перечисленных методов не лучше других. Выбор подхода в большей степени зависит от личного взгляда на предметную область. Если разработчик склонен к рассмотрению предметной области сверху вниз, то ему, возможно, больше подойдет нисходящий метод. Для многих разработчиков онтологий самым простым является комбинированный метод, так как понятия, находящиеся «посередине», обычно являются самыми наглядными.

Какой бы ни был избран метод, обычно процесс разработки начинается с определения классов. Из списка, составленного в подразделе 2.2.5, прежде всего выбираются термины, описывающие независимые объекты. В онтологии эти термины станут классами и будут точками привязки в иерархии [1]. Организация классов в иерархическую таксономию осуществляется как ответ на следующий вопрос: если объект является экземпляром одного класса, будет ли он обязательно (т.е. по определению) экземпляром некоторого другого класса?

В более формализованном виде эту процедуру можно выразить так: если класс *A* — это надкласс класса *B*, то каждый экземпляр *B* также является экземпляром *A*. Другими словами, класс *B* представляет собой понятие, являющееся «разновидностью» *A*.

2.2.7. Определение свойств слотов

Классы сами по себе не предоставляют достаточно информации для ответа на вопросы проверки компетентности (см. подраздел 2.2.3). После определения некоторого количества классов необходимо описать внутреннюю структуру понятий. Для каждого свойства из списка следует определить, какой класс оно описывает. Эти свойства станут слотами, привязанными к классам.

В онтологии слотами могут стать несколько типов свойств объектов:

- «внутренние» свойства;
- «внешние» свойства;
- части (как физические, так и абстрактные), если объект имеет структуру;
- отношения с другими индивидными концептами (между отдельными членами класса и другими элементами).

Все подклассы наследуют слот этого класса. Слот должен быть привязан к самому общему классу, у которого может быть данное свойство.

2.2.8. Определение фацетов слотов

Слоты могут иметь различные фацеты, которые описывают тип значения, разрешенные значения, число значений (мощность) и другие свойства значений, которые может принимать слот.

Мощность слота. Мощность слота определяет, сколько значений может принимать слот. В одних типах систем слоты могут иметь только одно значение (единичная мощность), в других — возможно любое число значений (множественная мощность). Некоторые системы позволяют задать минимальную и максимальную мощность в интересах

более точного описания количества значений слота. Минимальная мощность *N* означает, что слот должен иметь не менее *N* значений. Иногда полезно установить максимальную мощность равной 0. Такая установка будет означать, что для определенного подкласса слот не может иметь значений.

Тип значения слота. Фацет типа значения описывает, какие типы значений можно ввести в слот. Вот список наиболее общих типов значений.

Строка — самый простой тип значения, который используется в таких слотах, как название: значением является простая строка.

Слот типа число содержит числовые значения. Иногда используются более конкретные типы, например в формате с плавающей запятой (float) или в целочисленной форме (integer).

Булевы слоты — это простые флаги «да — нет».

Нумерованные слоты определяют список конкретных разрешенных значений слота. Например, в инструментальной среде Protégé [24] нумерованные слоты имеют тип «символ».

Слоты-экземпляры позволяют определить отношения между индивидными концептами, а также задают список разрешенных классов, т.е. классов, экземпляры которых можно использовать.

2.2.9. Домен слота и диапазон значений слота

Разрешенные классы для слотов-экземпляров часто называют диапазоном значений слота.

Некоторые системы позволяют ограничить диапазон значений слота, если слот привязан к определенному классу.

Классы, к которым слот привязан, или классы, свойство которых слот описывает, называются доменом слота.

Если в системе слоты привязаны к классам, то домен слота обычно составляют классы, к которым привязан слот, т.е. необходимость отдельно определять домен отсутствует.

Основные правила определения домена слота и диапазона значений слота схожи друг с другом:

- при определении домена или диапазона значений слота необходимо найти наиболее общие классы или классы, которые могут быть соответственно доменом или диапазоном значений слотов;
- с другой стороны, нет необходимости определять слишком общий домен и диапазон значений. Все классы в домене слота должны быть описаны слотом, а экземпляры всех классов в диа-

пазоне значений слота должны являться потенциальными заполнителями слота;

- нельзя выбирать слишком общий класс для диапазона значений, т.е. надо выбирать класс, охватывающий все заполнители, а не устанавливать диапазон значений как THING (самый общий класс в онтологии).

Например, вместо того чтобы перечислять все возможные подклассы класса «Вино» для диапазона значений слота, можно просто внести в список класс «Вино». В то же время нет необходимости определять диапазон значений слота как THING.

Уточним:

- если список классов, определяющих диапазон значений слота или домен слота, включает класс и его подкласс, то подкласс надо удалить;
- если список классов, определяющих диапазон значений слота или домен слота, включает все подклассы класса *A*, но не включает сам класс *A*, то в диапазон значений должен входить только класс *A*, а не его подклассы;
- если список классов, определяющих диапазон значений слота или домен слота, включает почти все подклассы класса *A*, то, вероятнее всего, для определения диапазона значений лучше подойдет класс *A*;

В системах, где привязка слота к классу равнозначна добавлению класса к домену слота, к привязке слота применяются аналогичные правила. С одной стороны, слот необходимо сделать как можно более общим, а с другой — надо гарантировать, что каждый класс, к которому привязывается слот, на самом деле имеет свойство, которое представляет слот.

2.2.10. Создание экземпляров

Последний шаг — это создание отдельных экземпляров классов в иерархии. Для определения отдельного экземпляра класса требуется выбрать класс, создать отдельный экземпляр этого класса, а затем ввести значения слотов.

Как уже отмечалось ранее, для любой предметной области не существует единственно правильной иерархии классов. Ее конкретная реализация зависит от возможных способов применения онтологии, уровня детализации, необходимого для приложения, личных предпочтений, а также от требований совместимости с другими моделями.

Тем не менее существует ряд руководящих принципов, которые необходимо учитывать при разработке иерархии классов. После определения значительного количества новых классов полезно остановиться и проверить возникающую иерархию на соответствие этим принципам.

Пример.

«Онтология об онтологии», выполненная в ИС Protégé (рис. 2.5):

Онтологии (Ontology)
- Языки разработки (Languages)
- Типы онтологий (Types)
- Методы и методологии (Methods and methologes)
  - Методы (Methods)
  - Методологии (Methologes)
- Интеграция онтологий (Integration)
- Среды разработки онтологий (Sredi)

Сравнение сред разработки (Comparing)

Сравнительная таблица средств управление онтологиями (Sredstva)

Сравнительные характеристики систем по работе с онтологиями (Characteristic)
- Общие (Obshie)
- Онтология (Ontologyes)
- Совместное использование (Ispolzovanie).

Представленные классы в свою очередь подразделяются на слоты и экземпляры.

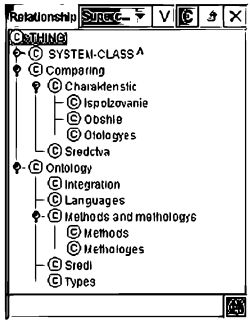


Рис. 2.5. Структура «Онтологии об онтологии» в ИС Protégé

2.3. Модели представления знаний

Способ представления знаний оказывает существенное влияние на характеристики и свойства разрабатываемой системы. Поэтому

представление знаний является одной из наиболее важных проблем, характерных для систем, основанных на знаниях.

Принято выделять следующие формализмы представления знаний: фреймы, семантические сети, исчисление предикатов первого порядка, продукции и комбинированные модели. Все они являются эвристическими вместе с объектно-ориентированными моделями знаний для статических и динамических систем.

2.3.1. Фреймы

Фрейм — это структура данных, предназначенная для представления стереотипных ситуаций [9]. Пример фрейма показан на рис. 2.6.

Cons

Имя	Петр
Фамилия	Иванов
Отчество	Сергеевич
Возраст	18
Работает	МарГТУ
Действие	.*exe

var

Рис. 2.6. Структура фрейма «Человек»

Фрейм определяется именем и совокупностью слотов. Слоты характеризуют объекты, классы и утверждения и могут иметь значение либо оставаться незаполненными. Значением слота может быть имя другого фрейма.

Имена слотов постоянны, а их значения — переменная информация, описывающая текущее состояние БД прикладной системы и изменяющаяся в зависимости от текущей ситуации. Иногда в качестве связи между двумя фреймами используются имена фреймов. Для этого в исходном фрейме в значениях слотов имеется обращение к имени фрейма наследования (рис. 2.7).

Кроме декларативного и процедурного назначения, слоты фрейма могут быть определены принудительно следующим образом:

- слоты, приводящие к выполнению действий в зависимости от полученных фреймом значений при сопоставлении, называются слотами-слугами;
- слоты, которые активизируются вне зависимости от значений других слотов, называются слотами-дьяволами.

Таким образом, фрейм может определять исходные состояния, например состояние A, и переход в новое состояние B на основе исходного описания с помощью фрейма A' в новое состояние, описываемое фреймом B'.

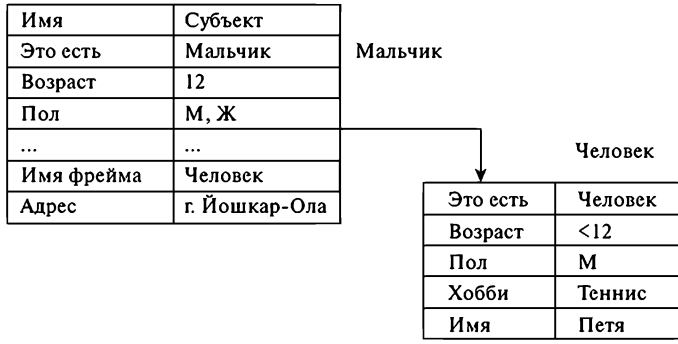


Рис. 2.7. Структура фрейма наследования

При использовании представления в виде фреймов предполагается, что значения верхних слотов фрейма заданы, а значения нижних заполняются в соответствии с ситуацией. Фреймы можно сравнить с процедурами в традиционных языках программирования. Описание процедуры в программе соответствует понятию фрейма-прототипа, а выполняемое ею действие — понятию фрейма-реализации.

Использование фреймов предполагает реализацию информационной системы на основе объектно-ориентированного подхода, подразумевающего объединение классов в совокупности однотипных объектов. Объекты имеют свойства, характерные для данного класса. При построении иерархии классов устанавливается соответствие между классами, фреймами и характеристиками объектов со слотами. Родственные свойства объектов наследуются, т.е. объект, находящийся на нижнем уровне классификации, имеет все свойства, присущие данному классу объектов.

К самым важным свойствам объектно-ориентированного подхода относятся:

- ***абстракция данных***, в соответствии с которой объекты представляют собой упрощенное, идеализированное описание реальных сущностей предметной области. Если соответствующие модели адекватны решаемой задаче, то работать с ними намного удобнее, чем с низкоуровневым описанием всех возможных свойств и реакций объекта;
- ***инкапсуляция,*** в соответствии с которой любой класс рассматривается как черный ящик. Пользователь класса видит и может использовать только интерфейсную часть класса (т.е. список декларируемых свойств и методов класса) и не вникает в его вну-

треннюю реализацию. Данные принято инкапсулировать в классе таким образом, чтобы доступ к ним по чтению или записи осуществлялся не напрямую, а с помощью методов. Принцип инкапсуляции (теоретически) позволяет минимизировать число связей между классами и соответственно упростить независимую реализацию и модификацию классов. Сокрытие данных — неотделимая часть ООП, управляющая областями видимости и являющаяся логическим продолжением инкапсуляции. Целью сокрытия является невозможность для пользователя узнать или испортить внутреннее состояние объекта. Применительно к представлению в виде фреймов доступ к определенным значениям слотов осуществляется при появлении или отсутствии значений данных в слотах-слугах и слотах-дьяволах;

- ***полиморфизм*** — свойство, позволяющее функции (методу) с одним и тем же именем иметь разный программный код (полиморфный код) в зависимости от того, объект какого класса используется при вызове данного метода. Полиморфизм обеспечивается тем, что класс-потомок изменяет реализацию метода класса-предка с обязательным сохранением сигнатуры метода;
- ***наследование*** — свойство, отражающее возможность порождать один класс от другого с сохранением всех свойств и методов класса-предка (прародителя или суперкласса) с добавлением при необходимости новых свойств и методов.

В отличие от описания процедур в традиционных языках программирования использование фреймов осуществляется не только по имени, но и по состоянию фрейма, описывающего ситуацию. В примере, рассмотренном ниже, это дает возможность получить модель состояния B', имея только часть информации о состоянии A'.

Пример.

Слоты фрейма A'	Правило
ОПИСАНИЕ	Нахождение в токсичной зоне. Опасность для жизни.
ЕСЛИ (потенциальная релевантность)	Запах ядовитого вещества?
ЕСЛИ (истинная релевантность)	Нахождение в зоне источника токсичности?
ТО <сообщение пользователю>	<не дышать>/<покинуть зону>
ТО <включить в агенду>	<действия, указанные в экспертной системе, системные указания>

Приоритет	Высокий
Частота использования	Включение в агенду 945 Использование при выводе 4 (т.е. правило рассматривалось 945 раз в списке конфликтующих правил и не исключалось 4 раза)

Представление правила фреймом характеризуется:

- наличием не одного, а нескольких условий и действий для различных уровней анализа правила;
- наличием достаточного количества описательных (невыполняемых) слотов, которые могут быть использованы для реализации управления.

Информация о правиле хранится в нескольких слотах. Это позволяет использовать разную интерпретацию правил в зависимости от целей разработки ЭС.

Все модели представления данных предназначены для воплощения их в правилах логического вывода. Это основное назначение модели. Модели в правилах представляются утверждениями.

Фрейм-прототип при реализации утверждения включает понятия пространства утверждения.

Фрейм-реализация — это внешние ссылки пространства утверждения и настройка на конкретную ситуацию (см. подраздел 1.5.1).

Сценарий (при обработке изображений) — это совокупность взаимосвязанных сцен (для систем обработки изображений) или фреймов.

2.3.2. Семантические сети

Семантическую сеть C в общем случае можно определить как

$$C = (x_1, x_2, \ldots, x_n, R_1, R_2, \ldots, R_m), \qquad (2.1)$$

где $x_1, x_2, \ldots, x_n$ — некоторые фиксированные множества объектов (содержимое понятий); $R_1, R_2, \ldots, R_m$ — система отношений на элементах этих множеств.

Отношения могут быть как бинарными, так и n-арными. Деление на объекты условное, так как сами отношения могут рассматриваться как объекты, связанные своими отношениями. Такое явное выражение связей между сущностями значительно упрощает осуществление процедур доступа к ним и дальнейшее продвижение по сети.

Графическое изображение семантической сети (рис. 2.8) представляет собой орграф. Вершинами орграфа являются предложения или понятия естественного языка, а дуги представляют отношения

между ними и отражают степень влияния одной выделенной сущности на другую. Дуга с инцидентными ей узлами является семантическим представлением правила.

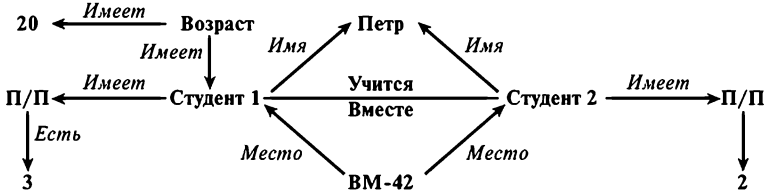


Рис. 2.8. Семантическая сеть

Необходимо отметить недостаточную выразительную мощность семантических сетей. Установлено [10], что дуга моделирует ситуацию не более чем из трех аспектов предметной области. Это обусловливает негибкость в представлении изменяющихся событий и большую сложность по сравнению с фреймовым представлением.

Направленность дуги семантической сети (рис. 2.9) определяет отношения типа «субъект — объект». К особо важным характеристикам дуги семантической сети относится реализация отношений типа «является». Дуга семантической сети может представлять собой бинарный предикат.

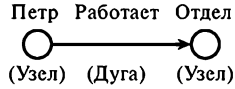


Рис. 2.9. Функциональный элемент семантической сети

Совокупность фреймов может быть связана семантической сетью, любой из узлов которой может иметь связь с любым количеством других узлов данной сети. В традиционном понимании семантическая сеть — это конструкция двух основных компонентов — узлов и дуг.

Узлы моделируют понятия предметной области, дуги моделируют отношения между парой понятий. Каждая из таких пар отношений представляет простой факт. Любой из узлов может быть соединен с любым числом других узлов, в результате чего формируется сеть фактов.

С помощью семантической сети (рис. 2.10) можно моделировать достаточно сложные отношения между различными понятиями, описывающими проблемную область.



Рис. 2.10. Пример семантической сети

Между объектами в семантической сети можно установить отношение типа «все со всеми», при котором каждый узел семантической сети связан дугами со всеми остальными ее узлами.

Семантическую сеть можно представить в виде бинарного предиката, т.е. как отношение с двумя аргументами, например «работает (Петр, отдел)». Такую информацию можно формализовать в виде списковых структур данных.

Кроме того, описание семантической сети можно задать *n*-аргументным отношением между понятиями при условии, что имя отношения будет иметь одно и то же значение.

При реализации семантической сети в базе данных информация описывается в виде троек: объект, атрибут, значение.

К недостаткам использования семантической сети можно отнести:

- трудности реализации свойства наследования между понятиями;
- сложность реализации временных отношений между понятиями.

2.3.3. Исчисление предикатов первого порядка

Широкое распространение получило представление знаний, основанное на исчислениях различного вида. Процедурная семантика такого метода описывает способ достижения цели через удовлетворение некоторого множества условий. Реализация правила осуществляется при выполнении всех необходимых условий, заложенных в него. При таком подходе связи (отношения) между объектами можно представить с помощью исчисления высказываний [7].

Модель реализуется путем доказательства логических высказываний. Логическое высказывание может быть доказано при условии, что могут быть доказаны все составляющие этого высказывания, кроме головного. Логика высказывания строится на описании различного рода предикатов.

Предикат в узком смысле — это свойство объекта. В широком смысле — это описание отношений между свойствами объекта, которые могут быть определены или не определены. Свойства предиката описываются совокупностью его аргументов. Корректность описания отношений между ними основывается не только на истинно-функциональных отношениях между входящими фактами, но и на их внутренней структуре, а также на понимании таких выражений, как «все», «всякий», «достаточно» и т.д., тесно связанных с определением кванторов общности и существования, лежащих в основе исчисления предикатов первого порядка.

Таким образом, отношения на языке логики предикатов можно задать в виде правил, представляющих собой совокупности фактов, связанных между собой логическими операциями конъюнкции («и», &, ∧, and), дизъюнкции («или», ∨, or), инверсии («не», ~, not), импликации («если, то», →) и др., а также кванторами общности (∀) и существования (∃).

Пример.
Мать (Анна, Вера) & Сестра (Анна, Лиза) → Мать (Лиза, Вера).
∀ (*X*) Мать (*X*, Вера) & Сестра (*X*, *Y*) → Мать (*Y*, Вера).

Основные компоненты модели. Основной единицей модели представления знаний является факт.

Факт — это отношение, представляющее собой объект и совокупность свойств объекта или процедур, которые определяются аргументами. Отношение — это предикат. Аргументы в предикатах задаются с помощью констант или с помощью переменных.

Пример.
Является (Петр, *X*);
Живет (*Y*, Воронеж, *T*),

где *X*, *Y*, *T* — переменные, определяющие интерпретацию факта в высказывании.

Высказывание — это элементарное утверждение, использующееся для построения сложных предложений.

Формирование предикатов осуществляется в соответствии со следующими правилами:

- каждый предикат имеет имя;
- каждый предикат обязательно имеет аргументы;
- количество аргументов предиката не ограничено;
- последовательность аргументов в задании предиката определяется последовательностью их интерпретаций в заданной проблем-

ной области. Эта последовательность заранее известна программисту и не меняется на протяжении всей программы.

Простейшим видом высказываний в модели представления знаний являются правила. В общем смысле ***правила*** — это сложные умозаключения или высказывания, которые формируются из простых с помощью логических операций (булевых функций). Особое значение имеет импликация, предназначенная для формирования каузальных (причинно-следственных) отношений.

Пример.
Работает (Петр, ЭВМ) → Оператор (Петр).
Разработал (Петр, программа) & (~ Работает (программа)) →
Исправить (Петр, программа) ∨ Передать (программа, *Y*).

Для утверждения истинности гипотезы на основе фактов, заданных в исходных данных (в базе данных), или на основе интерпретации значений переменных используются сложные правила. Мерой истинности переменных, используемых при доказательстве правила или сложного высказывания, являются кванторы общности и существования.

Квантор общности ∀ (*X*) читается как «для всех *X*» и определяет область «все». Квантор существования ∃ (*X*) читается как «существует значение переменных *X*» и определяется как «некоторые».

Примечание. Кванторы применяются только между высказываниями или отношениями. Использование логических операций между кванторами недопустимо.

Пример.
∀ (*X*) ∃ (*Y*) Руководит *(Y, X)* & отчитывается *(X, Y)*.
∀ (*Y*) ∃ (*X*) Руководит *(Y, X)* & отчитывается *(X, Y)*.

Еще одним компонентом модели являются ***утверждения***. На языке логики предикатов утверждения могут быть представлены:

- правилами, используемыми для доказательства;
- правилами вывода.

Правила вывода (или продукции) — это сложные предложения, определяющие новые правила и факты, т.е. предназначенные для задания новых стратегий с использованием новых фактов.

Различия между правилами доказательств и правилами вывода можно проиллюстрировать следующим образом.

Пример.
Правило:
∀ *(X)* Руководит *(X, Y)* → Отчитывается *(Y, X)*.
Правило вывода:
∀ *(X, Y, Z)* Руководит *(X, Y)* & Руководит *(X, Z)* → Отчитывается *(Z, Y)*, что позволяет вывести новое правило: Руководит *(Y, Z)*.

Основные недостатки логической модели знаний. Доказательство на языке логики предикатов происходит монотонно и аддитивно, т.е. рассуждения с помощью описания их предикатами не соответствуют мыслительной деятельности эксперта.

Аддитивность (от лат. *additivus* — придаточный, добавочный) — свойство величин по отношению к сложению, состоящее в том, что значение величины, соответствующее целому объекту, равно сумме значений величин, соответствующих его частям, в некотором классе возможных разбиений объекта на части.

Монотонность в языке логики предикатов определяет последовательное применение всех правил в аксиомах. В результате этого одно из правил, как ожидается, должно привести к результату. Это неизменно, но не всегда приводит к результату.

Варьирование правилами — это форма, состоящая из основы и нескольких (не менее двух) измененных воспроизведений (вариаций) правил. Вариационная форма взаимодействия правил возникает только при систематическом их применении, что недопустимо в рассматриваемой модели.

2.3.4. Модель представления знаний в виде правил продукции

Все представленные ранее модели (фреймы, семантические сети, исчисление предикатов первого порядка) могут быть реализованы как продукционные.

Системы, основанные на продукционной модели, состоят из трех типов компонентов:

- базы правил;
- базы фактов, содержащей декларативные знания о предметной области, используемые в качестве аргументов в условиях применимости продукций;
- интерпретатора продукций, реализующего функции анализа условий применимости, выполнения и управления выбором продукций.

Продукции наряду с фреймами являются наиболее популярными средствами представления знаний в системах ИИ. Они, с одной стороны, близки к логическим моделям, что позволяет организовывать на них эффективные процедуры вывода, а с другой — более наглядно отражают знания, чем классические логические модели. В них нет жестких ограничений, характерных для логических исчислений, что дает возможность изменять интерпретацию элементов продукции.

В общем виде под продукцией понимается следующее выражение:

$$(i);\ Q;\ P;\ A \Rightarrow B;\ N. \qquad (2.2)$$

Переменная (i) указывает имя продукции, в качестве которого может выступать некоторая лексема, отражающая суть данной продукции (например, «покупка книги»), или порядковый номер продукции в памяти системы.

Элемент Q характеризует сферу применения продукции.

Основным элементом продукции является ее ядро $A \Rightarrow B$.

Интерпретация ядра продукции зависит от того, что стоит слева и справа от знака секвенции $\Rightarrow$. Обычное прочтение ядра продукции выглядит так: ЕСЛИ A, ТО B.

Более сложные конструкции ядра допускают в правой части альтернативный выбор, например: ЕСЛИ A, ТО $B1$, ИНАЧЕ $B2$.

Секвенция может истолковываться в обычном логическом смысле как знак логического следования B из истинного A (если A не является истинным выражением, то о B ничего сказать нельзя). Возможны и другие интерпретации ядра продукции, например, A описывает некоторое условие, необходимое для того, чтобы можно было совершить действие B.

Элемент P отражает условие применимости ядра продукции. Обычно P представляет собой логическое выражение (как правило, предикат). Когда P принимает значение «истина», ядро продукции активизируется. Если P «ложно», то ядро продукции не может быть использовано.

Элемент N описывает постусловия продукции. Они актуализируются только в том случае, если ядро продукции реализовалось. Постусловия продукции описывают действия и процедуры, которые необходимо выполнить после реализации B. Выполнение N может происходить сразу после реализации ядра продукции.

Значения данных объединены для формирования информационного ядра. Элементы информационного ядра являются аргументами выражения, реализуемого продукцией. То выражение, которое соответствует информационному ядру, при сопоставлении загружается. Сеть функций дополняет доказательство для получения решений.

Если в памяти системы хранится некоторый набор продукций, то они образуют систему продукций, в которой должны быть заданы специальные процедуры управления продукциями, актуализации продукций и выбора для выполнения той или иной продукции из числа актуализированных. В ряде систем ИИ используют комбинации сетевых и продукционных моделей представления знаний.

Все представленные ранее модели (фреймы, семантические сети, исчисление предикатов первого порядка) могут быть реализованы как продукционные. Такие модели применяются при разработке систем искусственного интеллекта, связанных с обработкой изображений, лингвистической обработкой текста и распознаванием речи.

Контрольные вопросы

1. Что характеризует понятия «концептуализация» и «формализация» при определении проблемной области задачи?
2. Опишите структуру таксономической классификационной схемы.
3. Охарактеризуйте понятие и определите назначение онтологии предметной области.
4. Какова роль онтологий при поиске в сети Интернет?
5. Для чего предназначены модели знаний?
6. Охарактеризуйте известные модели представления знаний.

3. ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА ЛОГИЧЕСКОГО ПРОГРАММИРОВАНИЯ

3.1. Язык логического программирования Пролог

После почти 20-летнего успешного развития в 1996 г. Пролог стал стандартом логических языков программирования. Главным отличием его от так называемых процедурных языков, таких как Borland C, Delphi, Java и др., является его декларативный характер.

Если в традиционных языках программирования необходимо знать заранее, как решить ту или иную задачу, то в декларативном Прологе это делает «сам» интерпретатор. Эта особенность языка делает его незаменимым в областях, где сложность задач и объем необходимых знаний превосходят возможности человека. Это свойство наиболее подходит для исследований в области ИИ.

Пролог — это декларативный язык, в среде которого необходимо точно и грамотно на логическом языке описать условие задачи, а ее выполнение — это результат некоторого рутинного процесса, выполняемого интерпретатором. Фрагменты Пролог-программы иногда называют базами знаний. Далее рассмотрение языка осуществляется для версии Турбо Пролог.

Пролог реализует управляющую структуру в виде обратной цепочки логического вывода, т.е. доказательство от противного. Этим частично исключается монотонность.

Составляющие Пролог-программы. В самом общем виде Пролог-программа может быть представлена как совокупность двух основных разделов.

Первый раздел содержит постановку задачи, т.е. описание ее условия в виде набора исходных данных, представленных в форме БД.

Второй раздел описывает алгоритм, т.е. порядок выполнения программы, и представляет собой совокупность утверждений (формул), содержащих факты, правила и вопросы, необходимые для достижения поставленной цели.

Таким образом, в структуре программы выделяются следующие виды утверждений: факты, вопросы (запросы) и правила. В бинарной

интерпретации факт включает только «голову» утверждения. В конце факта ставится точка. Вопрос (запрос) представляет собой неосновной терм, всегда содержащий только непустое «тело» утверждения. Правило состоит из подцели и условий ее доказательства, т.е. заполненных «головы» и «тела» утверждения. В конце правила также ставится точка.

Утверждения. Простейший вид утверждений Пролог-программы — это факт. Он констатирует наличие отношений между объектами или переменными. Отношение в программе называется предикатом.

Пример.

```
нравится ("Лена", "теннис").
играет ("Костя", "футбол").
имеет ("Сергей", ___).
отец ("Катя", "Дмитрий").
плюс (3, 2, 5).
```

Составляющими отношений или предикатов являются атомы и переменные. Атомы в Пролог-программе записываются строчными буквами. Если атом необходимо записать с прописной буквы, его значение заключается в апострофы (двойные кавычки). В конце факта в базе данных программы ставится точка.

Вопросы в Пролог-программе имеют тот же формат, что и факты. Их можно различить по контексту, расположению на экране в режиме отладки программы или по наличию знака «?» в начале или в конце утверждения (в зависимости от диалекта интерпретатора). Семантически вопрос в Пролог-программе определяет наличие того или иного отношения между объектами.

Пример.

```
нравится ("Лена", "хоккей").
нравится ("Лена", "теннис").
? нравится ("Лена", X).
X = "теннис".
```

Правила предназначены для вывода логического следствия на основе имеющейся БД. Правила в Пролог-программе записываются в виде конструкции, называемой резольвентой:

$$t_1 :- a_1, a_2, ..., a_N. , \tag{2.3}$$

где t_1 — подцель (теорема или промежуточная цель программы); a_1, a_2, ..., a_N — аксиомы, посредством которых достигается подцель; :- — знак определения функции (читается: «определяется, как...»); , — знак операции конъюнкции (логическое «и»); · — обязательный знак окончания правила (резольвенты).

Пример.

```
Дано: "все люди ошибаются", "Сократ — человек".
Первую фразу можно формализовать следующим образом:
"для всех X, X ошибается, если X — человек".
Запишем исходные данные в терминах Пролог-программы:
failable (X):- person (X).
person (socrates).
Сформулируем запрос:
failable (socrates),
в результате получим ответ: yes.
```

3.2. Основные разделы программы

Описание данных присутствует в двух основных разделах Пролог-программы, один из которых называется predicates (отношения), а другой — clauses (утверждения). В разделе predicates описываются предикаты, используемые в БД, с описанием их типа. Для определения основной цели программы служит ключевое слово goal. В раздел clauses заносятся факты и правила, известные априорно. О содержимом этого раздела можно говорить как о данных, необходимых для работы программы.

Пример.

```
predicates
     cat (integer)
     food (symbol)
goal
     some (goals).
clauses
     cat (10).
     food (whiskas).
```

Заполнение разделов программы. Такие разделы Пролог-программы, как trace, project (название проекта) и include (внешний исходный файл) заполняются по желанию.

Раздел domains, объявляющий новые типы области данных, может быть заполнен по желанию или необходимости.

Пример.

```
domains
     man = symbol
     time = symbol
```

Раздел database объявляет предикаты, которые по желанию можно добавлять или удалять.

Пример.

```
database
     work (man, time)
```

Раздел global predicates или predicates объявляет области определения каждого предиката или глобального предиката в обязательном порядке.

Пример.

```
predicates
     know (man, man)
     work (man, time)
clauses                                  /* текст программы */
     work ("Петр", "днем").
     work ("Иван", "днем").
     know (A, B):- work (X, S), work (Y, S), X < > Y.
goal
     know (A, B)./* цель может быть задана по желанию */
```

Предикаты в Пролог-программе могут быть системными и пользовательскими, т.е. созданными разработчиком программы.

Логические термы. Совокупность синтаксических структур в Пролог-программе называется логическими термами. Термы могут иметь простую или сложную структуры.

Константы в Пролог-программе могут быть числом или атомом. Атом представляет собой последовательность строчных и прописных символов, цифр и специальных знаков, которая обязательно начинается со строчной буквы. Если атом в программе начинается с прописной буквы, то его заключают в апострофы (двойные кавычки).

Пример.

```
name (man, "Петр")
```

Основными называются термы, не содержащие переменных, т.е. факты. Термы, содержащие переменные, называются *неосновными*.

Сложный терм является в Пролог-программе или функтором, или оператором. Определение сложного терма в программе содержит имя, представляющее собой атом, и совокупность аргументов, которые в свою очередь также могут быть простыми и сложными термами.

Переменные в Пролог-программе. Переменные в Пролог-программе являются не местом в памяти, а реальным объектом. Место действия одной переменной — одна резольвента. При осуществлении механизма возврата значения всех переменных в резольвенте, через которые осуществляется возврат, очищаются.

Переменная в Пролог-программе представляет собой последовательность знаков подчеркивания, строчных и прописных букв, которая всегда начинается с прописной буквы или со знака подчеркивания.

Пример.

```
domains                                    /* раздел данных */
     тип фигуры = symbol
     выражение для площади = string
predicates
     площадь (тип фигуры, выражение для площади)
     фигура (тип фигуры, выражение для площади)
goal
     площадь (круг, Z).
clauses
     площадь (X, Y):- фигура (X, Y), write ("X=", X,
     "Y=", Y).
     фигура (квадрат, "сторона * сторона").
     фигура (круг, "pi * радиус * радиус").
     фигура (треугольник, "1/2 * основание * высота").
     фигура ( _, "неизвестно").
```

Наличие символа «_» означает использование фиктивной переменной. В рассмотренном примере с ее помощью реализован квантор общности.

В разделе данных domains объявлены типы данных пользователя («тип фигуры» и «выражение для площади»), а также указано, что «тип фигуры» эквивалентен стандартному символьному типу (symbol), а тип «выражение для площади» — стандартному строковому типу (string).

3.3. Рекурсивные вычисления в Пролог-программе

Рекурсивное описание правила содержит в своем теле ссылку на заголовок этого же правила. Возможны следующие три варианта рекурсивных правил:

- правая рекурсия — *pr*1() :- *pr*11(), *pr*12(), ... , *pr*1N(), *pr*1().
- левая рекурсия — *pr*1() :- *pr*1(1), *pr*21(), *pr*22(), ... , *pr*2M().
- обобщенная рекурсия — *pr*1() :- *pr*11(), *pr*12(), ... , *pr*1N(), *pr*1(), *pr*21(), *pr*22(), ... , *pr*2M().

Для исключения зацикливания во время выполнения рекурсивного правила необходимо предусмотреть условия завершения рекурсии, которое обычно реализуется одним из двух способов:

- заданием в программе альтернативного правила или факта *pr*1(), не содержащего рекурсии (выход происходит при успешном выполнении этого правила);
- формированием условия выхода одним из предикатов *pr*11(), *pr*12() ... , — выход происходит, если в процессе выполнения правил хотя бы один из предикатов завершается неуспехом.

Предикаты *pr*21(), ... , pr^2M() не влияют на выполнение рекурсии, а выполняются только после выхода из нее. Эти предикаты получают значения переменных из стека, в который они помещаются на время выполнения рекурсии. Производимые при этом вычисления называют хвостовыми вычислениями. Идеи реализации рекурсивных вычислений представлены в примере и в табл. 3.1, составленной на основе соответствующей Пролог-программы.

Таблица 3.1

Вызов предиката	Подстановки	Вычисления	Хвостовые вычисления	Результат вычислений
fact (3, Rez)	N=3, Rez=R	Next_N=N-1=2	R=N*P	Rez=R=3*2=6
fact (2, P)	N=2, P=R^1	Next_N^1=N^1-1=1	R^1=N^1*P^1	P^*=R^1=2*1=2
fact (1, P^1)	1=1, P=1	—	—	—

Пример.

```
domains
     number, product = integer
goal
     fact (3, Rez), write ("Факториал 3=", Rez), nl.
clauses
     fact (1, 1):- !.
     fact (N, R):- Next_N = N - 1,
     fact (Next_N, P):- R = N * P.
```

3.4. Процесс реализации вывода

Абстрактный интерпретатор Пролог выполняет вычисления, получая вопрос *P* и последовательность резольвент *Q*. Если *P* выводится из *Q*, то результат «yes», если не выводится, то — «no». Возможен также и третий вариант, когда результат непредсказуем.

В процессе реализации логического вывода в Пролог-программе используются следующие механизмы:

- унификации;
- сопоставления (согласования, совпадения);

- прямой трассировки;
- возврата.

Сопоставление — это процесс проверки двух термов на их соответствие друг другу. Термы сопоставимы в двух случаях: если они идентичны либо если они становятся идентичными при конкретизации своих переменных.

Если термы тождественны (идентичны), то процесс терпит успех, и неуспех — в противном случае.

При попытке согласования целевого утверждения из БД выбирается единичное утверждение, голова которого сопоставима с целевым утверждением. Если такое утверждение найдено, то из БД выбирается следующее утверждение, которое сопоставляется со следующим целевым утверждением. Если такой процесс терпит успех, т.е. все целевые утверждения согласуются путем прямого продвижения по программе, то он называется *прямой трассировкой*.

Примечание. Даже если все утверждения согласованы в поставленной цели один раз путем прямой трассировки, то можно получить другие доказательства той же цели, используя механизм возврата (управление процессом вывода с помощью предикатов fail, «!» и repeat).

Механизм возврата. Процесс согласования конъюнкций подцелевых утверждений в теле резольвенты осуществляется слева направо:

(голова) цель:– (тело) подцель_1, подцель_2, ..., подцель_*N*.

Каждая подцель согласуется с фактом, хранящимся в БД, или уже доказанным утверждением. Однако может возникнуть ситуация, когда одна из подцелей не может быть согласована. При этом осуществляется сдвиг влево на одну предшествующую подцель и выполняется ее сопоставление с новыми значениями из БД, и так до тех пор, пока не будет найдена сопоставимая подцель. Если она найдена, то процесс продолжается дальше слева направо до тех пор, пока не будет доказано целевое утверждение.

Если подцелей больше нет и факты в БД исчерпаны, то доказательства заканчиваются неудачей. При осуществлении механизма возврата значения всех переменных очищаются.

Пример.

```
predicates
     type (symbol, symbol)
     is (symbol, symbol)
     lives (symbol, symbol)
     can_swim (symbol)
goal
     can_swim (what), write (what," плавает"), nl, fail.
```

```
clauses
      type (mammal, animal).
      type (fish, animal).
      is ("зебра", mammal).
      is ("окунь", fish).
      is ("сельдь", fish).
      lives ("зебра", on_ground).
      lives ("сельдь", in_water).
      lives ("окунь", in_water).
      can_swim (Y):- type (X, animal), is (Y, X), lives
      (Y, in_water).
```

Поведение работы Пролог-программы можно сравнить с поиском выхода в лабиринте [7]. Пометим стрелкой направления движения. Предположим, что попали к развилке. Если все пути помечены стрелкой, то возвращаемся по пути, противоположному стрелке. Если таких путей несколько, то возвращаемся по тому, по которому шли последний раз (для этого следует присвоить каждому пути такой номер N, чтобы последний новый путь получал номер старше всех номеров уже пройденных путей). Если на развилке есть непройденный путь, то он помечается стрелкой и осуществляется следование по этому пути.

Если при движении возникает тупик, то возвращаемся по этому пути обратно до предыдущей развилки. Эта стратегия полная, и она приводит к выходу из лабиринта, если даже существует в нем единственный маршрут.

Шаги выполнения программы на языке Пролог. Итак, рассмотрим самый общий алгоритм работы Пролог-программы.

1. Нахождение подходящего правила для текущей цели (или подцели).
2. Передача и согласование параметров цели и правила.
3. Определение в зависимости от исхода шага 2 текущей (новой) цели или возврат к предыдущей цели с отменой значений для переменных, полученных в результате выполнения шага 2 для последней цели.
4. Создание точек ветвления программы с помощью предикатов repeat, fail и «!».
5. Инициализация переменных процедуры и восстановление их значений при возврате.

3.5. Предикаты

Системные предикаты. В программе на языке Пролог используются следующие группы системных предикатов:

- предикаты типа;
- арифметические и логические предикаты;
- предикаты ввода-вывода;
- предикаты, влияющие на ход выполнения программы (отсечения, вызов пользователя);
- металогические предикаты (работа с термами, классификация термов);
- предикаты управления и связи с операционной системой;
- предикаты отладки программы.

Предикаты типа позволяют задать новую переменную определенного типа, а также проверить принадлежность переменной к тому или иному типу, например:

integer (C) — задает переменную C целого типа;

real (C) — задает переменную C вещественного типа;

char (C) — задает переменную C символьного типа;

symbol (C) — задает переменную C символьного типа;

string (S) — задает переменную строкового типа;

atom (A) — цель истинна, если терм A является атомом;

atomic (A) — цель истинна, если терм A является атомом или целым числом;

var (D) — цель истинна, если терм D является переменной;

nonvar (D) — цель истинна, если терм D не является переменной;

Арифметические предикаты. В Пролог-программе справедливы и используются все виды арифметических операций:

+ — сложение;

- — вычитание;

* — умножение;

/ — деление;

exp — возведение в степень.

Кроме этого имеются другие встроенные математические операции:

mod — модуль, остаток от целочисленного деления;

div — частное от деления;

sin — синус;

cos — косинус;

ln — натуральный логарифм.

Предикат унификации. Термин «унификация» (от лат. *unio* — единство и *facere* — делать) означает приведение чего-либо к единой норме, форме, единообразию. Унификация в программе на языке Пролог — это операция приведения к единому значению. Следует особо отме-

тить, что предикат «=» не является оператором присваивания, хотя и может выполнять его функцию в частном случае.

Пример. Унификация как присваивание значения.

Программа вычисления факториала:

```
predicates
     factorial (integer, real)
     factorial (integer, real, integer, real)
goal
     factorial (5, X),
     write ("X = ", X).
clauses
     factorial (Fact, N):- factorial (Fact, N, 1, 1).
     factorial (Fact, N, Fact, N):- !.
     factorial (Fact, N, P, I):-
     NewP = P + 1,                          /* унификация */
     NewI = NewP * I,                       /* унификация */
     factorial (Fact, N, NewP, NewI).
```

Результат выполнения программы: X = 120.

Унификация в общем виде осуществляется следующим образом: сначала вычисляется выражение в правой части, которое затем унифицируется со значением в левой части.

Пример. Унификация выражения с переменной.

Дано выражение: R = exp (X * ln (Y)).

```
predicates
     power (integer, integer)
goal
     readint (X),
     readint (Y),
     power (X, Y).
clauses
     power (X, Y):-
     R = exp (Y * ln (X)),      /* операция унификации */
     write(X, "^", Y, "=", R), nl.
```

Двухместный предикат «=» представлен в программе в инфиксной форме записи. Он проверяет, унифицируются ли друг с другом его аргументы. Если в одном из аргументов содержатся неконкретизированные переменные, то в случае успешной унификации они конкретизируются.

Пример. Унификация структуры.

Дано выражение:

? студент (Петров, 4, 21) = студент (X, Y, Z),

где X= Петров, Y= 4, Z=21.

```
domains
     summa = real
```

```
        proizv = real
        itog = real
    predicates
        summa_chisel (real, real, summa)
        proizv_chisel (real, real, proizv)
        itog_chisel (summa, proizv, itog)
    goal
        makewindow (1, 10, 2, "Сумма и произведение", 0, 0,
        25, 80),
        write ("Введите первое число:\n A = "),
        readreal (A),
        write ("Введите второе число:\n B = "),
        readreal (B),
        write ("Сумма:\n C = A + B = "),
        summa_chisel (A, B, C), write (C),
        write ("\nПроизведение:\n D = A * B= "),
        proizv_chisel (A, B, D), write (D),
        write ("\nИтог:\n E = C + D = "),
        itog_chisel (C, D, E), write (E).
    clauses
        summa_chisel (A, B, C):- C = A + B.
                                  /* операция унификации */
        proizv_chisel (A, B, D):- D = A * B.
                                  /* операция унификации */
        itog_chisel (C, D, E):- E = C + D.
                                  /* операция унификации */
```

Одной из фаз процесса доказательства целевого утверждения является унификация его аргументов с аргументами утверждений базы знаний. Рассмотрим, как работает предикат унификации в том случае, когда аргументами цели являются списки [22].

Введем понятие шаблона (образца) списка как формы описания множества (семейства) списков, обладающих определенными свойствами.

Например:

- шаблон списка [X | Y] описывает любой произвольный список, состоящий не менее чем из одного элемента;
- шаблон [X1, X2 | Y] — список, состоящий не менее чем из двух элементов;
- аналогично [X1, X2, X3 | Y] — список, содержащий не менее трех элементов; а шаблон в виде переменной Z — любой список, в том числе и пустой.

Шаблон может содержать как переменные, так и константы. Например, шаблон [b | Z] задает любой список, первым элементом которого является элемент b.

При унификации происходит сопоставление шаблонов. Если шаблоны целевого утверждения и утверждения базы знаний представляют списки с несовместимыми различными свойствами (разные классы списков), то унификация заканчивается неудачей. Так, например, нельзя сопоставить списки

[X1, X2 | T], [a], [X1, X2, X3 | Z], [1, 2] и т.д.

Если шаблоны не противоречат друг другу, то осуществляется конкретизация отдельных переменных шаблона, т.е. присвоение им значений соответствующих констант или сцепление с соответствующими переменными другого шаблона. В результате оба шаблона должны стать идентичными и породить общее решение — новый шаблон.

Примеры: шаблон [[X, Y] | Z] задает список, начинающийся с подсписка из двух элементов; при сопоставлении его со списком [[a, b], c] унификация проходит успешно и переменные принимают следующие значения: X = a, Y = b, Z = [c].

Шаблоны [X, Y | Z] и [[a, b] | T] при сопоставлении дают общее решение — новый шаблон [[a, b], Y | Z]; образец [[X | Y] | Z] описывает любой список, начинающийся с подсписка, в котором есть, по крайней мере, один элемент — элемент X. Он соответствует, например, таким спискам:

[[a]] при X = a, Y = [], Z = [];
[[a, b], c] при X = a, Y = [b], Z = [c].

Если в процессе сопоставления и присвоения значений шаблоны не могут стать идентичными, то унификация заканчивается неудачей, как в следующем случае:

['бал', Y, Y] [X, X, 'цех'],

где сопоставление показывает, что элементы обоих списков должны быть одинаковы, однако ‘бал’ не равно ‘цех’ [22].

В Пролог-программе (версия Турбо-Пролог 2.0) используются следующие ***предикаты отношений*** [25]:

< — меньше;
> — больше;
<= — меньше или равно;
>= — больше или равно;
= — равно;
>< или <> — не равно.

Запись отношений также имеет инфиксную форму — «выражение (знак отношения) выражение» — и является термом, т.е. синтаксически правильной конструкцией языка Пролог. Терм считается синтаксически неверным, если в одной из частей (правой или левой от знака отношения) не будет стоять выражение либо в одной из частей будет только константа или атом. Интерпретатор вычисляет выражение в правой части, а затем — в левой. После этого выражения сравниваются с учетом знака отношения.

Рассмотрим ***предикаты ввода-вывода.*** *Предикат ввода* read используется в Пролог-программе в следующих нотациях:

readln (var) — ввод строковой переменной;

readint (var) — ввод целой переменной;

readreal (var) — ввод вещественной переменной;

readchar (var) — ввод символьной переменной.

При работе предиката read в Пролог-программе выполняется конкретизация переменной X с термом из текущего входного потока (значением, вводимым пользователем в режиме диалога с клавиатуры или посредством формы, отображаемой на экране монитора, либо из внешнего файла с данными или из БД). Если значение X будет недопустимым, то конкретизация не произойдет, программа терпит неуспех, а результат может быть непредсказуем.

Предикат вывода в Пролог-программе имеет следующую форму:

write (<аргумент_1>, ... , <аргумент_ N>).

Он выводит на текущее устройство (по умолчанию — экран дисплея) значения аргументов <аргумент_1>, ... , <аргумент_ N>, которые могут быть константами или переменными с заранее присвоенными значениями.

В предикате write можно использовать символы, начинающиеся со знака «\» и имеющие специальные значения:

\k — символы, имеющие ASCII код числа k;

\n — возврат каретки и перевод строки;

\t — табуляция.

Предикат форматного вывода

writef (format, arg1, ... , argN)

подобен write, но осуществляет форматированный вывод в соответствии с параметром format, который имеет вид:

%p,

где символ «p» может принимать следующие значения, соответствующие определенному формату выводимого числа: d — нормальное десятичное число (символы и целые числа); u — целое без знака; s — строка (атомы или строки); c — символ (цифры или буквы); g — вещественное число в коротком формате; e — вещественное число в экспоненциальном представлении; f — вещественное число в десятичном представлении (по умолчанию); x — шестнадцатеричное число (символы и целые числа).

К предикатам ввода-вывода также относятся предикаты окна.

Предикат открытия окна имеет следующий формат:

makewindow (A, B, C, D, E, F, G, L),

где A — номер открытого окна; B — число, определяющее цвет фона в окне; C — (число) атрибут рамки окна; D — заголовок открытого окна (строка); E — номер ряда (строки), с которого начинается окно; F — номер столбца начала окна; G — высота окна; L — ширина окна.

Предикат закрытия окна имеет вид:

removewindow (W, Z),

где W — номер окна; Z — число 0 или 1, задающее обновление экрана после закрытия окна (по умолчанию 0).

Пример.

```
predicates
     run (char)
goal
     run (X).
clauses
     run (X):- makewindow (1, 2, 3, "Привет", 0, 0, 25,
     80),
     write ("Здравствуйте!"),
     readchar (X),
     removewindow.
```

Предикаты управления работой программы. В процессе работы Пролог-программы осуществляется:

1) нахождение правила для текущей цели (подцели);
2) конкретизация переменных данного правила по аксиомам;
3) возврат, если сопоставимых аксиом нет;
4) ветвление программы;
5) означивание переменных и дальнейшее сопоставление правил с аксиомами.

Предикат цикла (repeat) предназначен для организации циклических операций, он является недетерминированным и задается в программе следующим образом:

```
predicates
     nondeterm repeat
clauses
     repeat:- repeat.
     repeat.
```

Пример.

Программа переводит введенные символы в верхний регистр, выход из цикла осуществляется по нажатию клавиши Enter:

```
predicates
     nondeterm repeat
     typewrite
goal
     makewindow (1, 2, 3, "Result", 0, 0, 20, 60),
     typewrite,
     removewindow.
clauses
     repeat.
     repeat:- repeat.
     typewrite:-
       repeat,
       readchar (C),
       upper_lower (C1,C),/* перевод в верхний регистр */
       write ("\n", "C = ", C, "C1 = ", C1),
       char_int (C1,13).
                  /* выход из цикла по клавише Enter */
```

Для реализации ветвления программы используются два ***предиката отсечений*** — fail и cut.

Предикат cut, чаще обозначаемый в программе знаком «!», предназначен для исключения возможного перебора всех вариантов решений.

Пример.

Программа прекращает ввод символов с экрана при нажатии клавиши Enter:

```
predicates
     typewrite (char)
goal
     makewindow (1, 2, 3, "Entering", 0, 0, 25, 80),
     readchar (C),
     typewrite (C).
clauses
     typewrite ('\13'):- !.
     typewrite (C):-
     write (C),
     readchar (C1),
     typewrite (C1).
```

Если предикат repeat предназначен для успешного завершения цели, то предикат fail определяет неуспех. Предикат fail предназначен

для вывода всех возможных решений цели, имеющихся в БД, и работает следующим образом. Если он встречается в программе, то текущая цель становится неразрешимой. Очистка всех переменных и доказательства проводится заново исходя из последовательности аксиом, заданных в БД (поэтому важно, в каком порядке заданы аксиомы и правила).

3.6. Списковые структуры

Пролог работает с такими структурами данных, как бинарные деревья и структурные списки, причем списки являются частным случаем бинарных деревьев.

Для выполнения операций, связанных с обработкой текстов, в таких языках программирования, как LISP, необходимо наличие операции следования. В Пролог-программе процедура следования задается в виде списковой структуры данных.

Список представляет собой упорядоченный набор объектов (элементов списка), следующих друг за другом. Элементы списка должны принадлежать к одному и тому же доменному типу. В теле программы список задается с помощью символа «*».

Пример.

```
domains
     list_num = integer *        /* список целых чисел */
predicates
     num (list_num)
```

или

```
domains
     list_num = element *  /* список чисел типа element */
     element = integer
                  /* тип element определен как целое */
```

Список в Пролог-программе — это совокупность термов, разделенных запятыми и заключенных квадратные в скобки.

Пример.

```
["Лена", "Петр", "Олег", "Сергей"];
[1, 2, 3, 6, 9, 3, 4];
[3.2, 4.6, 1.1, 2.64, 100.2];
[yesterday, today, tomorrow];
```

[element01] — список из одного элемента;

[] — пустой список;

G = graf ([a, b, c, d], [r (a, b), r (b, c), r (b, d), r (d, c)]) — описание графа в виде совокупности списков его вершин и дуг (ребер).

Для повышения наглядности программ в Прологе предусматриваются специальные средства списковой нотации, позволяющие представлять списки не только в традиционном виде [элемент_1, элемент_2, ... , элемент_N], но и в бинарной форме: [H | T], где H (Head) — это голова списка, T (Tail) — хвост или окончание списка, а знак «|» служит разделителем. Например, список [a, b, c, d] может быть записан следующими способами:

- [a] | [b, c, d];
- [a, b] | [c, d];
- [a, b, c] | [d];
- [a, b, c, d] | [].

Следует отметить, что аргумент T может быть определен рекурсивно, т.е. может сам являться списком и в свою очередь также состоять из головы и хвоста. Для ограничения рекурсии в любом списке Пролог-программы используют символ «пустой список» [3].

Пример.

Программа вычисления суммы элементов векторов X и Y, представленных в виде списков: X = (x1, x^2, ... , xN) и Y = (y1, y^2, ... , yN).

```
domains
     vector = integer*
predicates
     product (vector, vector, integer)
goal
     product ([1, 2, 3], [7, 8, 9], Rez),
     write ("Rez = ", Rez).
clauses
     product ([], [], 0).
     product ([X | Xs], [Y | Ys], S):- product (Xs, Ys,
     Sp),
     S = X + Y + Sp.
```

Поясним работу программы. Рекурсивное правило описывает отсроченные вычисления (табл. 3.2). После рекурсивного вызова остается невыполненным хвост правила, конец которого помещается в стек до тех пор, пока не произойдет сопоставление с первым вариантом правила. После того как это произойдет, частичная сумма Sp получит значение 0 и все накопленные в стеке хвостовые вычисления будут выполнены в обратном порядке.

Таблица 3.2

Вызов предиката	Подстановки	Хвостовые вычисления	Результат вычислений
product ([1, 2, 3], [7, 8, 9], Rez)	X = 1, Y = 7, Xs = [2, 3], Ys = [8, 9], Rez = S	S = X*Y + Sp	S = 1*7+43=50
product ([2, 3], [8, 9], Sp)	X' = 2, Y' = 8, Xs' = [3], Ys' = [9], Rez = Sp	Sp = X'*Y'+Sp	Sp=2*8+27=43
product ([3], [9], S')	X" = 3, Y" = 9, Xs" = [2, 3], Ys" = [8, 9], Sp=Sp'	Sp'=X'*Y'+Sp	Sp'=3*9+0=27
product ([], [], Sp")	Xs" = [], Ys" = [], Sp" = 0	—	Sp"=0

3.7. Вызов внешних функций из Пролог-программы и интерфейс с программами на других языках программирования

Вызов внешних функций из Пролог-программы продемонстрируем на примере реализации интерфейса с программой, написанной на языке программирования Borland C.

Компиляция программы из среды Turbo-Prolog. Основная программа на языке Турбо-Пролог [3] компилируется из среды в OBJ-файл. Во всех глобальных именах должен стоять предварительный символ подчеркивания. Описание внешней функции, которая будет вызвана из основной программы, имеет вид:

_имя_функции () — language c.

Пример.

```
/* Основная программа вызывает внешнюю функцию _hello_c, */
/* написанную на Borland C */
global predicates
     _message (string) — (i) language c
     _hello_c — language c
goal
     _message ("Hello from Turbo Prolog!"),
     _hello_c.
```

```
clauses
     _message (S):-
     makewindow (13, 7, 7, "", 10, 10, 3, 50),
     write (S),
     readchar (_),
     removewindow.
```

Компиляция программы на языке Borland C. Все функции должны быть скомпилированы в obj-файлы с использованием опции компилятора. Так, для среды BC++ 2.0 эта директива выглядит следующим образом:

bcc. exe -c -ml имя_файла (модель памяти large).

К глобальному имени функции необходимо добавить символы «_0», например: имя_функции_0 ().

Пример.

```
/* Функция _hello_c, написанная на языке Borland C */
/*и вызываемая из Пролог-программы */
void hello_c_0 ()
{
     message_0 ("Hello from Turbo C!");
}
domains
     integerlist = integer*
global predicates
     _message (integer) - (i) language c
     _hello_c (integer) - (i) language c
predicates
     write_list (integerlist)
     int_list (integerlist)
     read_list (integerlist)
     add_list (integerlist, integerlist, integerlist)
     read_list1 (integerlist, integerlist, string)
goal
     makewindow (1, 14, 13, "Привет, это Пролог!", 0, 0,
25, 80),
     clearwindow,
     int_list (List),
     read_list (List),
     removewindow (1, 0).
clauses
     int_list ([]).
     add_list ([], L, L).
     add_list ([N | L1], L2, [N | L3]):-
       add_list (L1, L2, L3).
       read_list1 (X1, X2, S):- str_int (S, Z),
       add_list (X1, [Z], X2),!.
     read_list1 (X1, X2, S):-
```

```
    write_list (X1),
    fail.
read_list (List):-
    write ("Введите элемент списка "),
    readln (S),
    read_list1 (List, L, S),
    read_list (L).
write_list ([]):- nl.
write_list([H|T]):-
    _hello_c (H).
    _message(N):-
    makewindow (13, 30, 30, "Сообщение", 5, 5, 15, 50),
    write(N),
    readchar (_),
    removewindow.
```

Получение исполнимого файла. Первый этап процесса получения исполнимого файла представляет собой компиляцию исходного текста программы, отличающуюся от работы интерпретатора тем, что в ее результате создается объектный файл. Для рассматриваемого примера — это файлы Пролога hello_pr. obj и hello_pr. sym, а также файл на языке Borland C — hello_c. obj. Компилятор выполняет преобразование исходной программы в программу на языке машинных команд.

Следующий этап — это компоновка, т.е. собственно создание исполнимого файла, заключающееся в объединении в общем коде всех необходимых объектных файлов и подстыковке требуемых библиотек. В рассматриваемом примере объектные модули на языках Пролог и Borland C включаются в исполнимый файл outfile. exe.

В дистрибутиве Пролог роль компоновщика выполняет специальная программа связи, или так называемая TLINK-программа.

Вызов TLINK-программы из командной строки ИС Турбо Пролог для рассматриваемого примера имеет следующий вид:

Tlink init. obj hello_pr. obj hello_c. obj hello_pr. sym, outfile,, prolog

где init.obj — стандартный файл Турбо-Пролог; hello_pr.obj, hello_pr.sym — файлы, получаемые после компиляции hello_pr.pro; outfile.exe — результирующий исполнимый файл.

3.8. Пример реализации экспертной системы на языке Пролог

В завершение раздела приведем полный листинг программы на языке Пролог, которая выводит все цепочки взаимозачетов пред-

приятий, имеющихся в базе знаний, и производит зачет по выбранной цепочке.

```
/* Экспертная система "Взаимозачет" */
domains
     list_str = string*
    list_list = list_str*
database – cep
    cep (list_str, integer)
database – db
    db_dan (string, integer, string)
predicates
    dolg (string, integer, list_str)
    prinad (string, list_str)
    min (integer, integer, integer)
    vivod (list_str)
    run
    wr (list_list, integer)
    analis
    el_num_in_list (integer, list_list, list_str)
    zachet (list_str, integer)
goal
    run.
clauses
    run:-
      makewindow (1, 23, 23, „", 0, 0, 25, 80),
      makewindow (2, 33, 44, "Взаимозачеты ", 3, 5, 20,
      70),
      write ("Составить цепочки взаимозачетов"), nl,
      write ("Введите первого должника: "),
/* правило, управляющее работой программы */
      readln (X),
      write ("Введите сумму: "),
      readint (Sum), nl,
      write ("Цепочки должников: "), nl,
      consult ("db. txt", db),
       not (dolg (X, Sum, [X])), nl,
       analis, nl,
       retractall (cep ( _, _)),
       write ("Для завершения работы нажмите любую
       клавишу."),
       readchar ( _),
       removewindow,
       removewindow.
/* Правило осуществляет поиск цепочки должников */
    dolg (X, Sum, Z):-
       db_dan (X, Sum1, Y),
       min (Sum, Sum1, Sum2),
```

```
        not (prinad (Y, Z)),
        dolg (Y, Sum2, [Y | Z]).
      dolg (X, Sum, L):-
        not (db_dan (X, _, _)),
        not (L= [X]), not (cep (L, _)),
        asserta (cep (L, Sum)),
        fail.
/* правило определяет принадлежность элемента списку */
      prinad (X, [X | _]):- !.
      prinad (X, [ _ | Y]):- prinad (X, Y).
/* правило определяет минимальный из двух элементов */
      min (S, S1, S):- S < S1,!.
      min ( _, S1, S1).
/* правило осуществляет вывод элементов */
/* списка (одной цепочки должников) */
      vivod ( []):- !.
      vivod ( [H | List]):-
        write (H), nl,
        vivod (List).
/* правило выводит все цепочки */
      wr ( [], _):- !.
      wr ( [H | L], N):-
        write ("Цепочка номер ", N, ":"), nl,
        vivod (H), cep (H, S), write ("Сумма зачета: ",
        S), nl,
        readchar ( _), nl, N1=N+1,
        wr (L, N1).
/* правило производит зачет по заданному номеру
цепочки */
      analis:-
        findall (L, cep (L, _), List), not (List = []),
        wr (List, 1),
        write ("Введите номер цепочки: "),
        readint (N),
        el_num_in_list (N, List, NL),
        cep (NL, S),
        write ("Сумма зачета: ", S), nl,
        zachet (NL, S),
        write ("Зачет произведен."),
        save ("db. txt", db),!.
      analis :- write ("Должников нет!").
/* правило находит в списке элемент по заданному
номеру */
     el_num_in_list (1, [H | _], H):-!.
     el_num_in_list (N, [ _ | List], NL):-
       N1=N-1,
       el_num_in_list (N1, List, NL).
/* правило производит изменения в базе данных (зачет) */
```

```
      zachet ( [ _], _):- !.
      zachet ( [H | [H1 | NL]], S):-
        zachet ( [H1 | NL], S),
        db_dan (H1, S1, H),
        S2=S1-S,
        retract (db_dan (H1, S1, H)),
        asserta (db_dan (H1, S2, H)).
/* конец листинга программы */
```

3.9. Диалекты и языки, используемые для задач искусственного интеллекта

За более чем полувековую историю проблемы искусственного интеллекта были разработаны [7] и используются в настоящее время следующие диалекты языка Пролог: C-Prolog, Quintus Prolog, SKW-Prolog, Arity Prolog, SWI-Prolog [29] и др.

Для обработки проблемных областей, связанных с математикой и описываемых с помощью формул и логических выражений, предназначен язык FORLOG.

Кроме того, хорошим языком программирования задач ИИ зарекомендовал себя LISP (LISt Processing), разработанный для символьных вычислений и модифицированный применительно к потребностям разработки интеллектуальных систем. LISP — функциональный язык, синтаксис и семантика которого обусловлены теорией рекурсивных вычислений.

В языке LISP осуществляется обработка списков, аргументами которых могут быть объекты, их свойства и функции. В свою очередь каждую функцию можно представить списком, среди аргументов которого имеется другая функция. Работа формализма ориентирована на обработку функций. Управляющая структура языка LISP находит среди аргументов списка описание состояния объекта или ситуации и выполняет процедуру, включенную в этот список. Если процедура содержит список, то он сопоставляется с текущей ситуацией. Управляющая структура языка LISP ориентирована на сопоставление с образцом, а модель — на исчисление предикатов.

Контрольные вопросы

1. Опишите принцип работы инструментального средства, реализующего логическую модель знаний.

2. Каким образом формируется база знаний Пролог-программы?
3. Каким образом определяются переменные в Пролог-программе?
4. Каким образом в Пролог-программе осуществляется выбор направления доказательства поставленной цели?
5. Как работает в Пролог-программе механизм унификации?
6. Опишите одни и те же данные в виде списка и структуры данных, укажите основные формы записи списков в Пролог-программе.
7. Какие совокупности единиц входят в базу знаний программы на языке Пролог?
8. Поясните организацию вызова из Пролог-программы внешних процедур, написанных на других языках программирования.

4. ОРГАНИЗАЦИЯ ПРИНЯТИЯ РЕШЕНИЙ В ЭКСПЕРТНЫХ СИСТЕМАХ

4.1. Организация логического вывода в экспертных системах

Продолжим процесс описания представления знаний в системах ИИ.

Блок логического вывода в архитектуре экспертной системы (см. рис. 1.1) предполагает наличие механизма принятия решения или использования правил для получения решений. Механизм применения правил — это управляющая структура, реализующая стратегии принятия решений. Порядок интерпретации фактов из рабочей области в правилах определяется последовательностью цепочек логического вывода. На практике используются два вида цепочек логических рассуждений или их комбинация:

1) прямое доказательство или прямая цепочка логического вывода, например:

если ∃ (*A*), то ∃ (*B*);

если ∃ (*B*), то ∃ (*C*); ...; и так далее до получения *K*, где *K* — цель (решение);

2) обратная цепочка логического вывода или доказательство от противного, например, требуется подтвердить факт *D*:

если ∃ (*D*), то ∃ (*C*);

если ∃ (*C*), то ∃ (*B*);

если ∃ (*B*), то∃ (*A*).

Принципиальным отличием двух вариантов цепочек логического вывода является форма приложения правил и использования фактов предметной области.

В прямой цепочке логических рассуждений предполагается существование факта *A*, если существует факт *B*; существование *B*, если существует *C*; *C*, если существует *D* и т.д.

В обратной цепочке логических рассуждений процесс доказательства по существующим фактам выполняется в обратном направлении. Факт *C* существует при наличии *D*, *B* — при наличии *C*, *A* — при наличии *B*.

При организации прямой и обратной цепочек логического вывода запись правила выполняется в каузальной форме, т.е. в виде «если — то». Использование правила определяется контекстом или стратегией.

Рассмотрим составляющие логического вывода подробнее.

4.2. Правила

В процессе обработки знаний были выделены такие единицы информации, как факты и отношения, которые фиксируются в моделях представления знаний: фреймах, семантических сетях, продукциях, исчислении предикатов первого порядка, а также комбинациях этих моделей. Следующей составной частью представления знаний в ЭВМ являются правила, лежащие в основе реализации механизма организации логического вывода.

Как было указано ранее, системы представления знаний, основанные на правилах, называются продукционными системами, а используемые в них эвристические правила — правилами продукций. Кроме этого, каждое правило, которое может быть реализовано в экспертной системе на основе всех представленных фактов, имеет свой коэффициент уверенности и характеризуется соответствующей мощностью и приоритетом.

Правила, реализованные в ЭС, могут определять новые типы отношений или новые классы отношений. Такие правила называются правилами-доказательствами. Условие каждого правила определяется шаблоном некоторой ситуации (образцом), при наличии которой правило выполняется (или не выполняется).

Механизм принятия решения на основе логического вывода в экспертных системах называется интерпретатором (рис. 4.1).

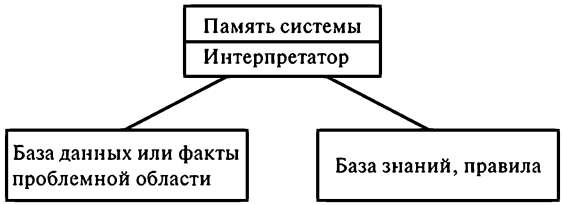


Рис. 4.1. Механизм принятия решения

Данные, описывающие текущую проблемную информацию, и правила, хранящиеся в базе знаний, совмещаются в интерпретаторе. За-

тем на основе логического вывода и с учетом выбранной стратегии рассуждений осуществляется принятие решения. Организация вывода или принятия решения выполняется в четыре этапа: выборка, сопоставление, разрешение конфликтов и выполнение (рис. 4.2).

Выборка представляет собой процедуру формирования списка активных правил, соответствующих текущим проблемным данным, и осуществляется в два шага:

1) синтаксическая выборка, осуществляемая пользователем, выполняет грубое определение правила по состоянию рабочей области памяти и переход к семантической выборке, т.е. к конкретизации правил;

2) семантическая выборка, выполняемая системой.

Дальнейший процесс не зависит от пользователя.

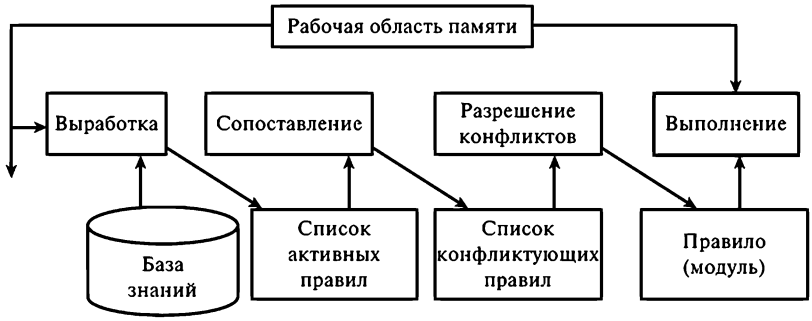


Рис. 4.2. Механизм вывода

Семантическая выборка определяет соответствие выбранных правил, текущей цели или подцели для заданной предметной области. Результатом выборки является совокупность активных правил из базы знаний.

Отобранные правила переходят на стадию сопоставления, заключающуюся в выполнении означивания выбранных правил текущими данными. Результатом сопоставления является список конфликтующих правил, означенных данными и готовых к выполнению (список правил со значениями слотов, наиболее точно характеризующих ситуацию или агенду).

Процедура разрешения конфликтов заключается в означивании правил на основе слотов агенды текущими рабочими данными из проблемной области. Результатом разрешения конфликтов является правило, выполнение которого обеспечивает изменение рабочей области памяти (см. рис. 4.2).

Выполнение — заключительный этап интерпретации, который в зависимости от принятой стратегии, используемого метода или цепочки логического вывода предполагает реализацию правила или модуля (заданной последовательности правил) при сопоставлении с образцом. Результатом этапа является изменение состояния рабочей области памяти или выполнение операций ввода-вывода (см. рис. 4.1).

4.3. Поиск решений

Основными компонентами организации логического вывода в экспертных системах являются цепочки вывода и управляющая структура, реализующая стратегию поиска.

Стратегия поиска решений — это:

- во-первых, задание метаправил по поиску решений;
- во-вторых, использование специфических эвристик;
- в-третьих, механизм, позволяющий усовершенствовать метод поиска решений.

Организация логического вывода в экспертной системе, опираясь на управляющую структуру, использует собственный механизм вывода, называемый интерпретатором. Работа интерпретатора осуществляется на основе данных, описывающих ситуацию и знания в виде правил, объединение которых предполагают получение решения.

Правила используются в экспертных системах, основанных на правилах. В ЭС, основанных на сопоставлении с образцом, их роль играют так называемые модули [12].

Рассмотренный выше механизм вывода (см. рис. 4.2) может быть реализован в двух типах архитектур экспертных систем:

- ЭС первого типа основаны на управлении правилами и содержат:

— рабочую область — текущее состояние памяти;
— источник знаний — данные;
— агенду — совокупность конфликтующих правил;
— правила — все активные правила;

- ЭС второго типа основаны на сопоставлении с образцом и содержат:

— модули, играющие роль правил [12];
— источник знаний, так называемую «классную доску»;

— агенду — список конфликтующих правил;
— политические правила — программы разрешения конфликтов;
— метаправила (общие правила) — правила работы с правилами.

Память системы (см. рис. 4.1) содержит сведения обо всех сеансах интерпретации при поиске решения.

В системах с архитектурой второго типа состояние рабочей области памяти сравнивается с данными и с образцами для сопоставления, представленными на «классной доске» или имеющимися в базе данных.

Политические правила переупорядочивают агенду, результатом чего является изменение состояния «классной доски». Множество политических модулей обеспечивает разнообразие способов подготовки управляющих решений в ЭС, использующих управление, основанное на сопоставлении с образцом.

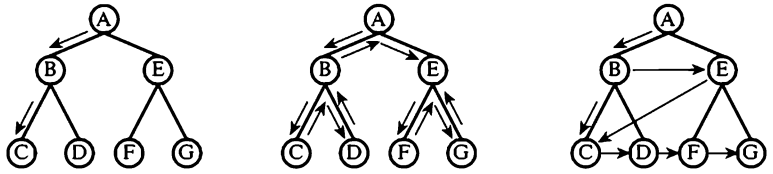
Кроме обычных правил, определяющих принятие решений, в ЭС второго типа используются общие правила (метаправила), позволяющие в десятки раз повысить эффективность ЭС. *Метаправила* — это правила работы с правилами. Выбранный механизм обработки и структура метаправил являются наиболее общими и позволяют охватить при обработке больший объем данных, а последовательное применение частных правил в первой архитектуре может привести к росту базы знаний, и из-за путаницы в частных правилах увеличивается база знаний, что приводит к неадекватным решениям.

Если в архитектуре первого типа сформулированные правила могут быть прочитаны экспертом, даже не являющимся программистом, то во второй архитектуре метаправила могут быть прочитаны и использованы только программистом.

4.4. Управляющая структура

Механизм приложения правил-продукций и правил-доказательств называется управляющей структурой. Эта структура определяет способ поиска фактов предметной области и в зависимости от их наличия или отсутствия применение того или иного правила. Механизм работы с правилами реализуется метауправляющей структурой. Поиск фактов в классической управляющей структуре осуществляется путем реализации одной из стратегий поиска: вглубь или вширь (рис. 4.3, *а, б* и рис. 4.3, *в* соответственно, где *A*, *B*, *C*, ... — факты, участвующие в правилах).

Глубинная стратегия является исчерпывающей, поскольку она раскрывает все возможные значения недавно появившейся в списке слотов пары. Новые правила добавляются к началу списка в соответствии с их приоритетами.



а *б* *в*

Рис. 4.3. Стратегии поиска решений:
а, *б* — вглубь; *в* — вширь

Широтная стратегия, также являющаяся исчерпывающей, раскрывает все возможные значения нового факта. Новые правила добавляются к концу списка в соответствии с их приоритетами. Первым применяется правило, которое может производить новые пункты в списке слотов. Если не пусты оба списка — список правил и список активных слотов (агенда), то приоритет отдается очередному элементу списка правил.

Очевидно, что в процессе рассуждения не исключены конфликтные ситуации. Например, если пункту списка слотов соответствует несколько правил, то какое из них будет выполнено в первую очередь? Для решения подобной проблемы дополнительно к имеющимся могут быть предложены стратегия выборочной оценки и стратегия оценки лучших.

Стратегия выборочной оценки является наиболее эффективной, так как выбирается только одно, самое успешное направление рассуждения. Как только одно правило из списка стало истинным, все остальные правила исключаются из него.

В соответствии со стратегией оценки лучших новые правила перемешиваются с уже имеющимися в списке в соответствии с приоритетами. Первым применяется правило, которое может производить новые пункты в списке слотов. Если не пусты оба списка, то приоритет отдается очередному элементу списка слотов.

Перечень стратегий поиска может быть расширен. Как правило, различные инструментальные средства искусственного интеллекта дополняют его собственными стратегиями.

4.5. Технологии принятия решений в системах с базами знаний

Технологии принятия решений классифицируются в соответствии со следующими подходами:

- поверхностный подход подразумевает использование правил общего характера, релевантных предметной области и полученных только от эксперта в виде эвристик. Поиск решения ведется индуктивным методом;
- при структурном подходе правила общего и частного видов принятия решений используются на основе поиска по дереву решения или с помощью специального механизма (стратегии принятия решений);
- глубинный подход подразумевает процедуру систематизации проблемной области с учетом модели представления знаний. Правила общего и частного видов формируются исходя из используемой модели, состояния предметной области и рекомендации экспертов.
- совмещенный или комбинированный подход сочетает в себе все перечисленные ранее методы.

4.6. Методы поиска, реализованные в экспертных системах

Методы поиска можно классифицировать следующим образом:

- по определению предметной области:
  - — по размерности пространства;
  - — количеству пространств и определению места и времени;
  - — моделям, описывающим предметную область;
  - — совокупности моделей;
  - — определению неопределенности, т.е. точности задания данных, размытости представления информации и т.д.;
- по представлению результатов:
  - — по количеству представленных результатов (один, несколько, все);
  - — полноте представления информации и результата.

Рассмотрим некоторые известные методы.

Слепой поиск. Стратегия слепого списка использует стратегии поиска вглубь и поиска вширь в совокупности с прямой и обратной це-

почкой логического вывода. Пример реализации приведен на рис. 4.4, на котором конечные вершины графа являются результатом поиска.

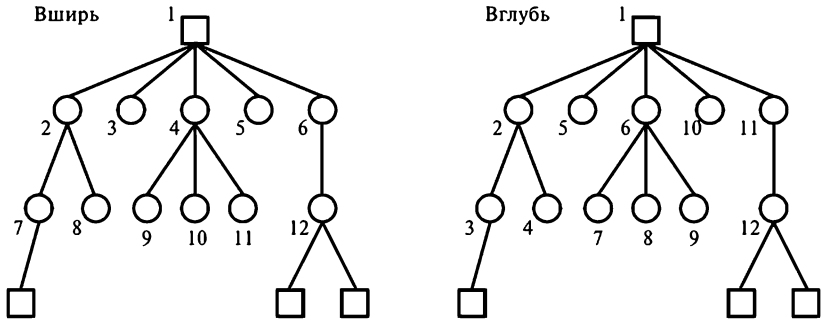


Рис. 4.4. Методы поиска

Метод редукции. Данный метод описывается с помощью и/или-графа. Реализация задачи подразумевает ее разбиение на совокупность подзадач, каждая из которых представляется дугой графа. Каждая дуга графа имеет свое назначение. Различают дизъюнктивные ветви (дуги «или») и конъюнктивные дуги (дуги «и»).

При выполнении подзадач с помощью дуги «или» должна быть выполнена хотя бы одна из подзадач текущей задачи. При реализации дуги «и» должны быть выполнены все подзадачи. Конъюнктивные дуги на графе объединяются специальным значком (рис. 4.5). При поиске результата на и/или-графе обычно применяются стратегии поиска вширь и вглубь.

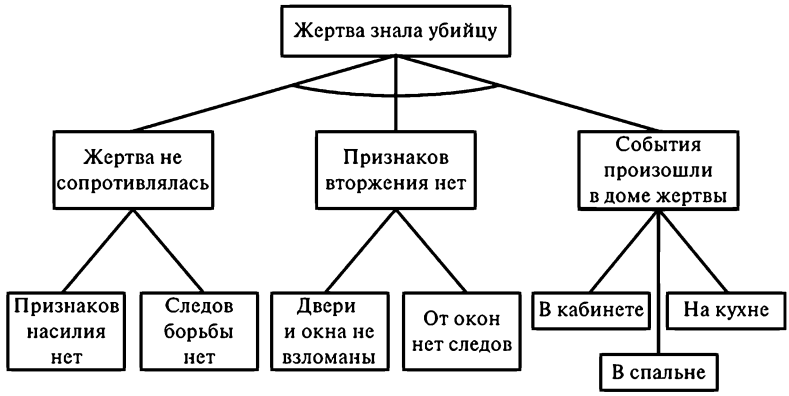


Рис. 4.5. Метод редукции

Эвристический метод поиска. Процедура поиска слепым методом подразумевает, что порядок просмотра вершин определен заранее и не зависит от расположения цели. При увеличении пространства поиска это обстоятельство приводит к существенному росту объема памяти и времени, необходимых для решения задачи. Стремление сократить время поиска привело к созданию эвристических методов поиска, т.е. методов, использующих лишь некоторую информацию о предметной области. При этом методе просматривается не все пространство поиска, а только пути, приводящие к цели с наибольшей вероятностью. Для сокращения пространства поиска используются различные критерии, например такие, как мера «перспективности» вершины, позволяющая сократить объем перебираемых вершин без потери полноты информации.

Метод поиска с помощью генерации и проверки. Данный метод не предусматривает задания пространства поиска в явном виде. В процессе реализации процедуры поиска на каждом очередном шаге генерируется возможное решение и проверяется, ведет ли оно к результату или является результирующим. Генератор считается полным, если он обеспечивает генерацию всех возможных решений, и неизбыточным, если каждое решение генерируется не более одного раза.

При генерации текущего возможного решения (состояния, подзадачи) возникает проблема распределения знаний между генератором и устройством проверки. При реализации в генераторе слепого или эвристического поиска используются минимальные знания об области, достаточные для генерации возможных решений, а устройство проверки определяет, не является ли очередное решение целевым.

Метод поиска в иерархических пространствах. Методы, предназначенные для реализации поиска в одном пространстве, не позволяют решать сложные задачи, так как с увеличением пространства экспоненциально возрастает время поиска. В целях разрешения данной проблемы общее пространство поиска разбивается на ряд иерархически связанных подпространств, поиск в которых осуществляется в первую очередь в соответствии со специальным алгоритмом.

Методы поиска в иерархических пространствах делятся на поиск в факторизованном пространстве, а также поиск в фиксированном и изменяющемся множестве пространств [12]. Модель предметной области в данном случае может представлять комбинацию моделей.

4.7. Использование процедур

В традиционных языках программирования при вызове модуля или процедуры используется имя процедуры. При разработке систем искусственного интеллекта выбор модуля (образца или правила) осуществляется на основе текущего состояния проблемной области.

Совокупность правил, реализованных в управляющей структуре в соответствии с алгоритмом на основе исходной структуры данных, выполняется процедурами различного вида:

- семантического и синтаксического контроля;
- моделирования;
- обработки и изменения структур данных и сред их сопровождения;
- поиска решений.

Экспертная система должна выполнять процедуры, позволяющие реализовать процессы таким же образом, как это делает человек при использовании собственного опыта экспертных знаний и рассуждений. К одной из самых важных особенностей человеческого мышления можно отнести способность вырабатывать достаточно грамотные решения, опираясь на явно неполный или противоречивый набор исходных данных. Таким образом, в полноценной ЭС должны быть реализованы процедуры обработки неопределенностей.

4.8. Представление неопределенности в информационных приложениях с базами знаний

Организация логического вывода в экспертных системах предполагает наличие процедур описания процесса или его моделирования, организацию диалога с пользователем, а также обработку неполной и неточной информации. Это связано с тем, что описание процесса моделирования определяется не только данными, составляющими информационное ядро проекта, но и сведениями из БЗ, а также полученными в результате диалога с пользователем.

Неопределенность (рис. 4.6) можно определить как степень соответствия процесса или состояния характеристикам реального мира. Например, в системах искусственного интеллекта для обработки изображений неопределенностью является само изображение. Для его

определения используются не только детерминированные, но и статистические и эвристические процедуры.

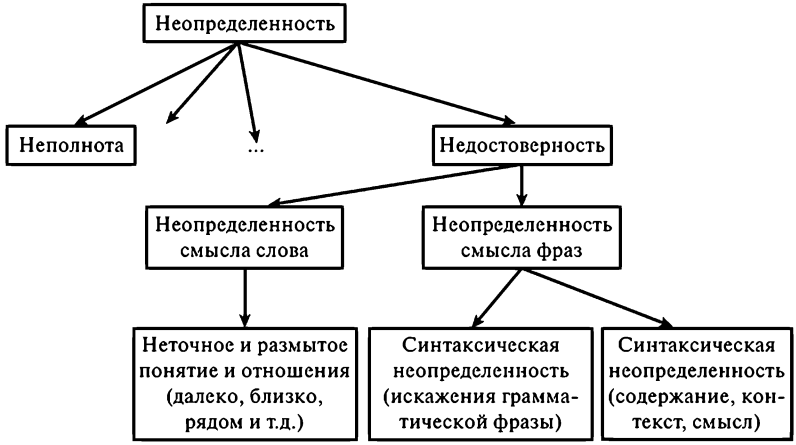


Рис. 4.6. Фрагмент классификации неопределенностей

Для исключения семантической неопределенности слов используются вероятностные характеристики и аппарат нечетких множеств.

Неопределенность смысла фраз идентифицируется с использованием лингвистического анализатора текста, причем отдельно анализируется как синтаксическая неопределенность (за счет грамматических искажений), так и семантическая (смысловые искажения).

Оценка семантической составляющей неопределенности может быть получена с помощью:

- вероятностных показателей различного рода;
- коэффициентов уверенности и мощности правил в методе редукции;
- введения интервалов значений и вероятностей попадания в заданный интервал;
- использования формулы Байеса;
- лингвистических переменных;
- переменных неопределенности и т.д.

Мощность правила — это апостериорная характеристика, определяемая по формуле

$$P_{\text{пр}} = K_{\text{уф}} \cdot K_{\text{упр}}, \qquad (4.1)$$

где $K_{\text{уф}}$ — коэффициент уверенности факта в условии; $K_{\text{упр}}$ — коэффициент уверенности правила.

В зависимости от назначения системы коэффициенты уверенности либо определяются разработчиком системы, либо вносятся в систему пользователем в процессе ее эксплуатации.

Интервалы в определении неопределенности задаются как [−1, +1]. Конкретные значения показателей являются вероятностными характеристиками, однако «−1» — всегда «ложь», а «+1» — всегда «истина».

Формула Байеса [18] имеет следующий вид:

$$P(H_i|E) = \frac{P(E|H_i) \cdot P(H_i)}{P(E)}, \tag{4.2}$$

где $P(H_i)$ — априорная вероятность гипотезы H_i; $P(H_i \mid E)$ — вероятность гипотезы H_i при наступлении события E (апостериорная вероятность); $P(E \mid H_i)$ — вероятность наступления события E при истинности гипотезы H_i; $P(E)$ — вероятность наступления события E.

Апостериорная вероятность некоторого события E определяется через совокупность попарно несовместных гипотез H_i, образующих полную группу событий. Событие E существует, если существует гипотеза H_i. Вероятность вычисляется для каждой пары «гипотеза H_i — событие E».

Использование лингвистических переменных и переменных неопределенности характерно для систем, связанных с обработкой естественного языка. Переменная неопределенности описывается следующим кортежем:

$$X_{\text{нсопр}} := < a, X, m >, \tag{4.3}$$

где a — наименование переменной; X — область определения значений переменных; m — нечеткое множество, в которое попадает X значение нечеткой переменной или переменной неопределенности.

Лингвистическая переменная является следствием использования переменной неопределенности или используется вместо нее в системах обработки текстов. Лингвистическая переменная определяется кортежем:

$$X_{\text{л}} := < \beta, \alpha, X, S, T >, \tag{4.4}$$

где β — наименование лингвистической переменной; α — интервал значений, в который попадает лингвистическая переменная, областью для которой является X; S — синтаксическая процедура использования лингвистической переменной; T — семантическая процедура использования лингвистической переменной.

Контрольные вопросы

1. Опишите последовательность обработки фактов в прямой и обратной цепочках логического вывода.
2. Назовите основные составляющие для организации логического вывода в системах ИИ.
3. Чем различаются стратегии поиска решений в системах с базами знаний?
4. Опишите механизм работы интерпретатора в системах с базами знаний.
5. Определите назначение управляющей структуры в системах ИИ.
6. Какие методы определения неопределенности используются в системах ИИ и в экспертных системах в частности?

5. ИНТЕЛЛЕКТУАЛЬНЫЕ ТЕХНОЛОГИИ ОБРАБОТКИ ИНФОРМАЦИИ

5.1. Интеллектуальные системы, основанные на нечеткой логике

Описанные ранее положения можно применять для логического вывода утверждений.

Пример.
Известно, что $x \in A$, $A \subset B$, тогда в соответствии с аксиомой вывода: $x \in B$.
Пусть A — множество народных депутатов, а B — множество пользующихся правом бесплатного проезда в общественном транспорте.
Тогда утверждение $A \subset B$ трансформируется в правило вывода: «Если лицо является народным депутатом, то оно пользуется правом бесплатного проезда в общественном транспорте».

Если множества не сравнимы непосредственно, может потребоваться дополнительное функциональное преобразование, которое позволит рассматривать одно множество как подмножество другого [2].

Нечеткое правило можно сформулировать как условное высказывание: «Если X есть A, то Y есть B»,
где A и B — нечеткие множества.

На языке математики это записывается в виде упорядоченной пары

$$(A, B), \tag{5.1}$$

где A — нечеткое подмножество пространства входных значений X; B — нечеткое подмножество пространства выходных значений Y,

либо как отношение (оператор):

$$R : R = A \to B. \tag{5.2}$$

Отношение R можно рассматривать как нечеткое подмножество прямого (декартова) произведения $X \times Y$ множества предпосылок X и множества следствий Y.

Примечание. Каждое правило определяет «нечеткое пятно» (декартово произведение $A \times B$) в пространстве состояний системы. Чем обширнее нечеткие множества A и B, тем больше и неопределеннее «нечеткое пятно». Нечеткие правила являются блоками для построе-

ния знаний. Можно сказать, что каждое нечеткое правило действует как ассоциативная память, связывающая нечеткий отклик B с нечетким стимулом A [26].

Система нечеткого логического вывода представляет собой композицию нечетких правил [27]:

$$\bigcup_{p=1}^{k_j}\left[w_{jp}\cdot\left(\bigcap_{i=1}^{n}x_i=a_{i,jp}\right)\right]\rightarrow y=b_j, j=\overline{1,m}, \qquad (5.3)$$

где m — количество нечетких термов, степень принадлежности к которым требуется определить; k_j — количество правил вывода, необходимых для определения степени принадлежности к нечеткому терму b_j; n — количество условий, реализующих правило вывода; w_{jp} — вес правила; x_i — входное значение, принадлежащее нечеткому терму $a_{i,jp}$; y — выходное значение.

Правила, входящие в (5.3), обычно имеют вид:

«Если цена велика и спрос низкий, то оборот мал», (5.4)

где «цена» и «спрос» — входные переменные; «оборот» — выходное значение; «велика», «низкий» и «мал» — функции принадлежности (нечеткие множества), определенные на множествах значений «цены», «спроса» и «оборота» соответственно.

Пример.

Система кондиционирования может быть описана правилами: «Если температура в комнате высокая, то скорость вращения вентилятора высокая» и «Если температура в комнате низкая, то скорость вращения вентилятора низкая». Результатом преобразования посылки «Температура в комнате 30 °С» для кондиционера может служить указание «Включить вентилятор». Все значения температур, при которых необходимо его включение, образуют подмножество во множестве условий, приводящих к включению вентилятора. Преобразование производится функцией управления, роль которой в данном случае может выполнять термостат.

Нечеткие правила вывода образуют базу правил. Следует особо отметить, что в нечеткой экспертной системе, в отличие от традиционной, работают все правила одновременно, однако степень их влияния на выход может быть различной. Таким образом, в основе нечетких экспертных систем лежит принцип суперпозиции множества правил при оценке их влияния на конечный результат.

Процесс обработки нечетких правил вывода в экспертной системе состоит из четырех этапов:

1) вычисление степени истинности левых частей правил (между «если» и «то») — определение степени принадлежности входных значений нечетким подмножествам, указанным в левой части правил вывода;

2) модификация нечетких подмножеств в правой части правил вывода (после «то») в соответствии со значениями истинности, полученными на первом этапе;

3) объединение (суперпозиция) модифицированных подмножеств;

4) скаляризация результата суперпозиции, т.е. переход от нечетких подмножеств к скалярным значениям.

Для определения степени истинности левой части каждого правила нечеткая экспертная система вычисляет значения функций принадлежности нечетких подмножеств от соответствующих значений входных переменных. Например, для правила (5.4) определяется степень вхождения конкретного значения переменной «цена» в нечеткое подмножество «велика», т.е. истинность предиката «цена велика». К вычисленным значениям истинности могут применяться логические операции. Наиболее часто используются следующие определения операций нечеткой логики:

$$\begin{aligned} \text{truth}\ (\sim X) &= 1 - \text{truth}\ (X), \\ \text{truth}\ (X\ \&\ Y) &= \min\{\text{truth}\ (X), \text{truth}\ (Y)\}, \\ \text{truth}\ (X \vee Y) &= \max\{\text{truth}\ (X), \text{truth}\ (Y)\}, \end{aligned} \qquad (5.5)$$

где X и Y — высказывания; truth (Z) — степень истинности высказывания Z.

Полученное значение истинности предназначено для модификации нечеткого множества, указанного в правой части правила. Для выполнения такой модификации применяют метод «минимума» (Correlation-min Encoding) либо метод «произведения» (Correlation-product Encoding).

Первый метод ограничивает функцию принадлежности множества, указанного в правой части правила, значением истинности левой части (рис. 5.1).

Во втором методе значение истинности левой части используется как коэффициент, на который умножаются значения функции принадлежности (рис. 5.2).

Результатом выполнения правила является нечеткое множество. Говоря более строгим языком, происходит ассоциирование переменной и функции принадлежности, указанной в правой части.

Выходы всех правил вычисляются нечеткой экспертной системой отдельно, однако в правой части нескольких из них может быть указана одна и та же нечеткая переменная. Как было сказано ранее, при определении обобщенного результата необходимо учитывать все правила. Для этого система осуществляет суперпозицию нечетких множеств,

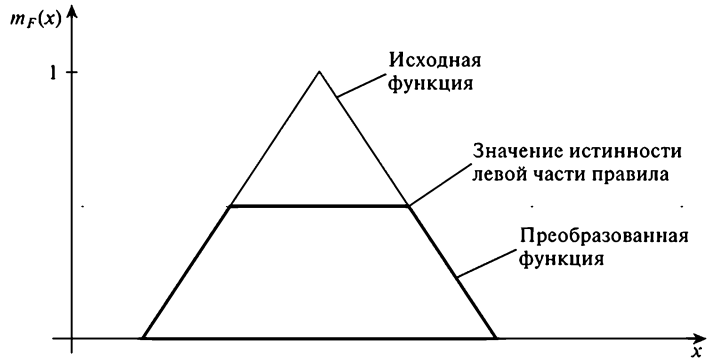


Рис. 5.1. Метод «минимума»

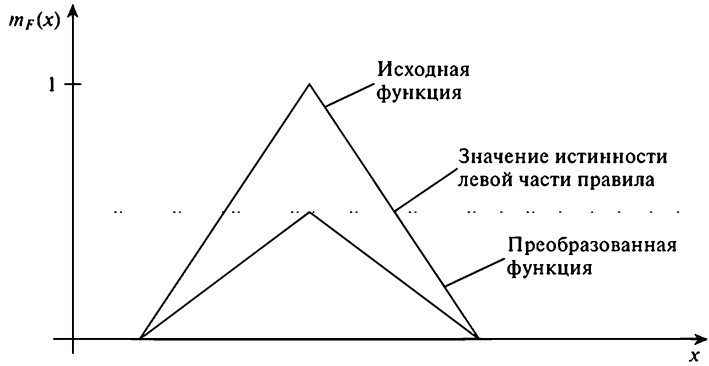


Рис. 5.2. Метод «произведения»

связанных с каждой из таких переменных. Эта операция называется нечетким объединением правил вывода.

Например, правая часть правил «Если цена мала, то спрос велик» и «Если цена велика, то спрос мал» содержит одну и ту же переменную — «спрос». Два нечетких подмножества, получаемые при выполнении этих правил, должны быть объединены экспертной системой.

Суперпозиция функций принадлежности нечетких множеств обычно определяется методом максимума комбинации (Max Combination):

$$\forall x, m_{\text{sum}\,F}(x) = \max\{m_{iF}(x)\}, i = \overline{1,n}, \tag{5.6}$$

где $m_{sum\,F}(x)$ — результирующая функция принадлежности; $m_{i\,F}(x)$ — нечеткие множества.

Реализация этого метода для двух функций представлена на рис. 5.3.

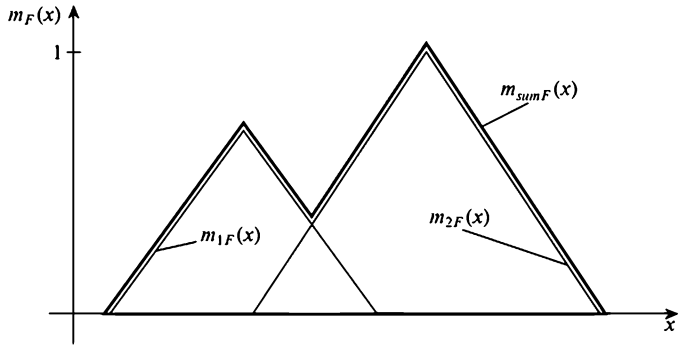


Рис. 5.3. Метод Max Combination

Другой метод суперпозиции, называемый Sum Combination, заключается в суммировании значений всех функций принадлежности (рис. 5.4):

$$\forall x, m_{\text{sum}\,F}(x) = \sum_{i=1}^{n} m_{iF}(x). \tag{5.7}$$

Самым простым, но и наименее используемым, является подход, при котором суперпозиция не производится, а выбирается одно из правил вывода, результат которого считается интегральным. Такой прямой метод используется для задания функции принадлежности четко измеримых понятий, например таких, как скорость, время, давление и т.д.

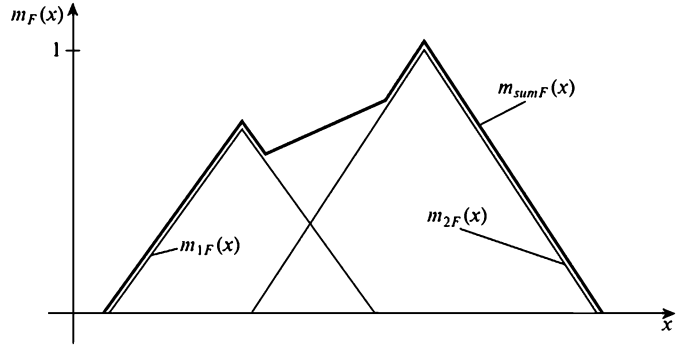


Рис. 5.4. Метод Sum Combination

Конечным этапом обработки базы правил вывода является переход от нечетких значений к скалярным. Процесс приведения нечеткого множества к некоторому единственному значению называется скаляризацией или дефаззификацией (от англ. *Defuzzification*).

Обычно это значение определяется методом нахождения центра тяжести функции принадлежности (Centroid Defuzzification Method) или методом максимального значения функции принадлежности (Modal Defuzzification Method), проиллюстрированными соответственно на рис. 5.5 и 5.6.



Рис. 5.5. Скаляризация методом нахождения центра тяжести



Рис. 5.6. Скаляризация методом нахождения максимума

Конкретный выбор методов суперпозиции и скаляризации осуществляется в зависимости от желаемого поведения нечеткой экспертной системы. В качестве инструментального средства для разработки нечеткой экспертной системы можно воспользоваться пакетом CubiCalc 2.0 компании Hyper Logic Corporation [2].

5.2. Нейронные сети

Еще в середине 1980-х гг. многие исследователи обратили внимание, что системы искусственного интеллекта ввиду их слабой способности к самообучению, встретившись с ситуацией, не предусмотренной разработчиком, либо формируют сообщение об ошибке, либо дают совершенно неправильные результаты. Для преодоления подобных проблем было предложено использовать искусственные нейронные сети.

Под искусственными НС подразумеваются вычислительные структуры, моделирующие биологические процессы, обычно ассоциируемые с процессами, происходящими в человеческом мозге. НС представляют собой распределенные параллельные системы, способные к адаптивному обучению путем анализа положительных и отрицательных воздействий. Элементарным преобразователем в данных сетях является искусственный нейрон, названный так по аналогии с биологическим прототипом [6].

5.2.1. Биологический и искусственный нейроны

Нервная система и мозг человека состоят из нейронов, соединенных между собой нервными волокнами, способными передавать электрические импульсы [14]. Процессы восприятия и передачи сигналов от органов чувств (кожи, ушей, глаз) к мозгу, мышление и управление действиями — все это реализовано в живом организме в виде обмена электрическими импульсами между нейронами. Нервная клетка, или ***нейрон***, является особой биологической клеткой (рис. 5.7).

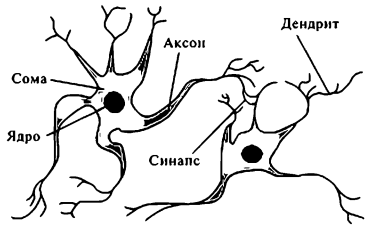


Рис. 5.7. Упрощенная структура биологического нейрона

Нейрон состоит из тела, или сомы, а также отростков нервных волокон двух типов: дендритов, принимающих импульсы, и единственного аксона, по которому нейрон может передавать импульс. Тело нейрона включает ядро и плазму. Нейрон получает сигналы (импульсы) от аксонов других нейронов через дендриты (приемники) и передает

сигналы, сгенерированные телом клетки, вдоль своего аксона (передатчика), который в конце разветвляется на волокна. На окончаниях этих волокон находятся специальные образования — синапсы, влияющие на амплитуду импульсов.

Синапс является элементарной структурой и функциональным узлом между двумя нейронами (волокно аксона одного нейрона и дендрит другого). Под воздействием приходящего импульса в синапсе высвобождаются химические вещества, называемые нейротрансмиттерами, которые диффундируют через синаптическую щель, возбуждая или затормаживая, в зависимости от типа синапса, способность нейрона-приемника генерировать электрические импульсы. Результативность передачи импульса синапсом может настраиваться проходящими через него сигналами так, что синапсы могут обучаться в зависимости от активности процессов, в которых они участвуют. Эта зависимость от предыстории действует как память. Важно отметить, что веса синапсов могут изменяться со временем, а значит меняется и поведение соответствующих нейронов.

Другими словами, каждый нейрон характеризуется внутренним состоянием и порогом возбудимости, а его входы делятся на возбуждающие и тормозящие. Поступивший на возбуждающий вход сигнал повышает степень активности нейрона, а на тормозящий — наоборот, снижает ее. Если сумма сигналов на возбуждающих и тормозящих входах превышает порог возбудимости, то нейрон формирует выходной сигнал, поступающий на входы связанных с ним других нейронов, т.е. происходит распространение возбуждения (сигнала) по нейронной сети.

Кора головного мозга человека содержит около 10^{11} нейронов и представляет собой протяженную поверхность толщиной 2—3 мм и площадью около 2200 см^2. Каждый нейрон связан с 10^3—10^4 другими нейронами. Таким образом, мозг человека в целом содержит приблизительно 10^{14}—10^{15} взаимосвязей.

Нейроны взаимодействуют короткими сериями импульсов продолжительностью, как правило, несколько миллисекунд. Сообщение передается посредством частотно-импульсной модуляции. Частота может изменяться от нескольких единиц до сотен герц, что в миллион раз медленнее, чем в быстродействующих электронных схемах. Тем не менее сложные задачи распознавания человек решает всего за несколько сотен миллисекунд, что не доступно большинству современных ЭВМ. Процесс принятия решения контролируется сетью нейронов, затрачи-

вающих на выполнение одной операции всего несколько миллисекунд. Единственным объяснением такого феномена стало предположение, что для решения таких сложных задач мозг «запускает» параллельные программы, каждая из которых содержит около 100 шагов. Явление получило название «массовый параллелизм». Основываясь на таком подходе, можно обнаружить, что количество информации, посылаемое от одного нейрона другому, должно быть очень малым (несколько бит). Из этого следует, что основная часть информации не передается непосредственно, а захватывается и распределяется в связях между нейронами [6].

Искусственный нейрон (далее — нейрон) представляет собой искусственную структуру, моделирующую свойства биологического нейрона. Одной из наиболее простых и общих моделей нервной клетки является так называемая модель МакКаллока — Питса [31], представленная на рис. 5.8.

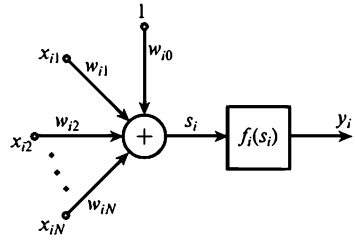


Рис. 5.8. Модель нейрона по МакКаллоку — Питсу

Математически модель нейрона можно записать следующим образом:

$$\left.\begin{aligned} s_i &= \sum_{j=1}^{N} w_{ij} \cdot x_{ij}(t) + w_{i0} \\ y_i &= f_i(s_i) \end{aligned}\right\}, \tag{5.8}$$

где s_i — суммарный сигнал или функция состояния нейрона; N — количество входов нейрона; w_{ij} — совокупность весов входных сигналов; x_{ij} — совокупность сигналов на входе нейрона; y_i — выходной сигнал нейрона; f_i — функция активации нейрона.

Одной из первых искусственных нейронных сетей является так называемый персептрон Розенблатта (от лат. *perceptio* — получение, собирание). ***Персептроном*** называют однослойную нейронную сеть, состоящую из нейронов с пороговой функцией активации [13].

Понятие функции активации является фундаментальным в теории нейронных сетей. Функция активизации f_j определяет реакцию нейрона на совокупность внешних воздействий, выраженную величиной выходного сигнала, как функции от его текущего состояния.

В настоящее время при моделировании в нейросетевом базисе используется большое разнообразие функций активации, различающихся главным образом видом переходной характеристики [2, 6, 11]. Наиболее часто встречающиеся функции активации приведены в табл. 5.1.

Таблица 5.1

Функция активации	Формула	Примечание
Ступенчатая пороговая (персептрон)	$f_j(s_i)=\begin{cases}0 \text{ при } s_i < a\\ 1 \text{ при } s_i \ge a\end{cases}$	a — величина порога
Линейная пороговая	$f_j=\begin{cases}0 \text{ при } s_i < a_1,\\ k\cdot s_i + b \text{ при } a_1 < s_i < a_2,\\ 1 \text{ при } s_i \ge a_2\end{cases}$	a_1, a_2 — величины порогов; k, b — параметры функции
Линейная	$f_j = k\cdot s_i + b$	k, b — параметры функции
Сигмоидальная униполярная (логистическая)	$f_j=\frac{1}{1+e^{-\beta\cdot s_i}}$	β — параметр, влияющий на форму функции
Сигмоидальная биполярная (гиперболический тангенс)	$f_j=\frac{e^{\beta\cdot s_i}-e^{-\beta s_i}}{e^{\beta s_i}+e^{-\beta\cdot s_i}}$	β — параметр, влияющий на форму функции
Гауссиана	$f_j = e^{-k\cdot(s_i-a)^2}$	k, a — параметры функции
Экспоненциальная	$f_j = e^{-\beta\cdot s_i}$	β — параметр, влияющий на форму функции

5.2.2. Классификация нейронных сетей

Нейронная сеть представляет собой совокупность нейронов, определенным образом соединенных друг с другом и с внешней средой посредством связей, характеризуемых определенными весовыми коэффициентами [6]. В зависимости от функций, выполняемых нейронами в сети, можно выделить три их типа:

- входные нейроны, на которые подается вектор входного воздействия или образ внешней среды. Обычно эти нейроны не осуществляют никаких вычислений, а лишь передают информацию со своего входа на выход путем собственной активации;
- выходные нейроны, выходные значения которых представляют выходы нейронной сети. Преобразования в нейронах этого типа происходят в соответствии с выражениями (5.8);
- промежуточные нейроны, которые составляют основу нейронных сетей. Преобразования в них также выполняются выражениями (5.8).

В процессе функционирования сети осуществляется преобразование входного вектора в выходной, т.е. переработка информации. Конкретный вид преобразования, выполняемого сетью, обусловливается не только характеристиками составляющих ее нейронов, но и особенностями архитектуры сети, а именно:

- топологией межнейронных связей;
- выбором определенных подмножеств нейронов для ввода и вывода информации;
- способами обучения сети;
- наличием или отсутствием конкуренции между нейронами;
- способами управления и синхронизации процессов информационного обмена между нейронами.

С точки зрения топологии, можно выделить три основных типа нейронных сетей (рис. 5.9): полносвязные, многослойные (слоистые) и слабосвязные (с локальными связями).

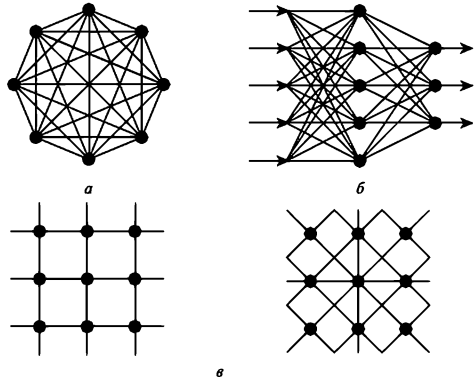


Рис. 5.9. Варианты архитектуры нейронных сетей:
а — полносвязная сеть; *б* — многослойная сеть
с последовательными связями; *в* — слабосвязные сети

В *полносвязных нейронных сетях* (рис. 5.9, *а*) каждый нейрон передает свой выходной сигнал остальным нейронам, в том числе и самому себе. Все входные сигналы подаются на входы всех нейронов. Выходными сигналами сети могут быть все или некоторые выходные сигналы нейронов после нескольких тактов функционирования сети.

В *многослойных нейронных сетях* (рис. 5.9, *б*) нейроны объединяются в слои. Каждый слой содержит совокупность нейронов с едиными входными сигналами. Число нейронов в слое может быть любым и не зависит от количества нейронов в других слоях. В общем случае сеть состоит из Q слоев, пронумерованных слева направо. Внешние входные сигналы подаются на входы нейронов входного (нулевого) слоя, а выходами сети являются выходные сигналы последнего слоя. Кроме входного и выходного слоев, в многослойной нейронной сети есть один или несколько скрытых слоев. Связи от выходов нейронов некоторого слоя q к входам нейронов следующего слоя $(q+1)$ называются последовательными.

В свою очередь в многослойных нейронных сетях выделяют следующие типы НС:

1) монотонные являются частным случаем слоистых сетей, отличаются от последних рядом дополнительных условий, накладываемых на связи и нейроны. Каждый слой, кроме последнего (выходного), разбит на два блока: возбуждающий и тормозящий. Связи между блоками также разделены на тормозящие и возбуждающие. Если от нейронов блока A к нейронам блока B ведут только возбуждающие связи, то это означает, что любой выходной сигнал блока является монотонной неубывающей функцией любого выходного сигнала блока A. Если же эти связи только тормозящие, то любой выходной сигнал блока B является невозрастающей функцией любого выходного сигнала блока A. Для нейронов монотонных сетей необходима монотонная зависимость выходного сигнала нейрона от параметров входных сигналов;

2) без обратных связей, в которых нейроны входного слоя получают входные сигналы, преобразуют их и передают нейронам первого скрытого слоя, и так далее вплоть до выходного, который выдает сигналы для интерпретатора и пользователя. Если не оговорено противное, то каждый выходной сигнал q-го слоя подастся на вход всех нейронов $(q+1)$-го слоя; однако возможен вариант соединения q-го слоя с произвольным $(q+p)$-м слоем.

Среди многослойных сетей без обратных связей различают полносвязные (выход каждого нейрона q-го слоя связан с входом каждого нейрона $(q+1)$-го слоя) и частично полносвязные. Классическим ва-

риантом слоистых сетей являются полносвязные сети прямого распространения (рис. 5.10);

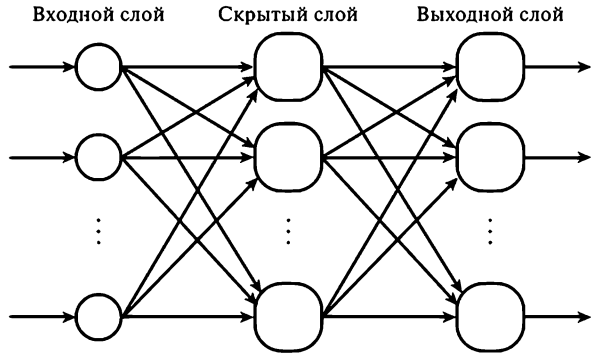


Рис. 5.10. Многослойная сеть прямого распространения

3) с обратными связями, в которых информация с последующих слоев может передаваться на предыдущие слои нейронов. Эти сети также могут быть подразделены на следующие классы:

- *слоисто-циклические*, отличающиеся тем, что слои замкнуты в кольцо. Последний слой передает свои выходные сигналы первому, при этом все слои равноправны и могут как получать входные сигналы, так и выдавать выходные;
- *слоисто-полносвязные*, состоящие из слоев, каждый из которых представляет собой полносвязную сеть. Сигналы передаются как от слоя к слою, так и внутри слоя. В каждом слое цикл работы распадается на три этапа: прием сигналов с предыдущего слоя, обмен сигналами внутри слоя, выработка выходного сигнала и передача его последующему слою;
- *полносвязно-слоистые*, аналогичные по своей структуре слоисто-полносвязным, но функционирующие другим образом. В таких НС фазы обмена информацией внутри слоя и передачи ее следующему не разделяются. На каждом такте нейроны всех слоев принимают сигналы от нейронов как своего слоя, так и последующих.

Примером сетей с обратными связями могут послужить частично-рекуррентные сети Элмана и Жордана, представленные на рис. 5.11.

Наконец, в *слабосвязных нейронных сетях* (см. рис. 5.9, *в*) нейроны располагаются в узлах прямоугольной, гексагональной или октогональной решетки. В таких НС каждый нейрон связан с четырьмя,

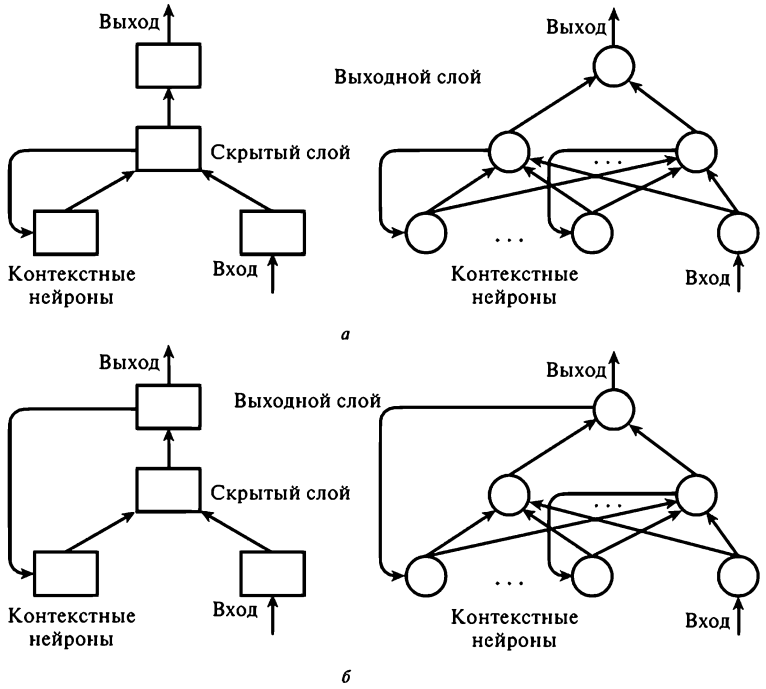


Рис. 5.11. Сети с обратными связями:
а — Элмана; *б* — Жордана

шестью или восемью своими ближайшими соседями, называемыми окрестностями фон Неймана, Голея и Мура соответственно.

По типам структур нейронов известные нейронные сети можно разделить на гомогенные (однородные), состоящие из нейронов одного типа с единой функцией активации, и гетерогенные, типы и функции активации нейронов которых различны.

Существуют бинарные и аналоговые сети. Первые из них оперируют только двоичными сигналами, и выход каждого нейрона может принимать значение либо логического ноля (заторможенное состояние), либо логической единицы (возбужденное состояние).

Еще одна классификация делит нейронные сети на синхронные и асинхронные. В первом случае в каждый момент времени лишь один нейрон меняет свое состояние, во втором — состояние изменяется одновременно у целой группы нейронов, как правило у всего слоя. Алгоритмически ход времени в нейронных сетях задается итерационным выполнением однотипных действий над нейронами.

Сети можно классифицировать также по числу слоев. Теоретически число слоев и число нейронов в каждом слое может быть произвольным, однако фактически оно ограничено ресурсами компьютера, на котором реализована нейронная сеть. Чем сложнее сеть, тем более сложные задачи она может решать.

Независимо от способа реализации нейронную сеть можно рассматривать как взвешенный ориентированный граф. Узлы этого графа соответствуют нейронам, а ребра — связям между ними. С каждой связью ассоциирован вес — рациональное число, отображающее оценку возбуждающего или тормозящего сигнала, передаваемого по этой связи на вход нейрона-приемника при возбуждении нейрона-передатчика. Нейронная сеть образуется путем объединения ориентированными взвешенными ребрами выходов нейронов с входами. Граф межнейронных соединений может быть ациклическим либо произвольным циклическим, что также может служить одним из классификационных признаков нейронной сети.

Принятие некоторого соглашения о тактировании сети (времени срабатывания нейронов) дает аппарат задания алгоритмов посредством нейронных сетей. Разнообразие этих алгоритмов ничем не ограничено, так как можно использовать нейроны с различными функциями активации и состояния, а также двоичными, целочисленными, вещественными и другими форматами представления значений весов и входов. Это дает возможность описывать в терминах нейронных сетей решение как хорошо формализованных задач из области математики и физики, так и плохо формализуемых задач распознавания, классификации, обобщения и ассоциативного запоминания [2].

Сети могут быть конструируемыми или обучаемыми. В конструируемой сети число и тип нейронов, граф межнейронных связей, веса входов нейронов определяются при создании сети исходя из решаемой задачи. Например, при конструировании НС, функционирующей как ассоциативная память, каждая входная последовательность из заранее определенного набора, предназначенного для запоминания сетью, участвует в определении весов входов нейронов сети. После конструирования функционирование сети заключается в следующем. При подаче на входы частичной или ошибочной входной последовательности сеть через какое-то время переходит в одно из устойчивых состояний, предусмотренных при ее конструировании. При этом на входах сети появляется одна из запомненных последовательностей, признаваемая сетью как наиболее близкая к изначально поданной.

В обучаемых сетях графы межнейронных связей и веса входов изменяются при выполнении алгоритма обучения. Алгоритмы обучения сети делятся на наблюдаемые, ненаблюдаемые и смешанные (гибридные). Первые при обучении сравнивают заранее известный выход с получившимся значением. Вторые обучаются, не зная заранее правильных выходных значений, но группируя «близкие» входные векторы так, чтобы они формировали один и тот же выход сети. Ненаблюдаемое обучение используется, в частности, при решении задачи кластеризации. При смешанном алгоритме обучения часть весов определяется при наблюдаемом, а часть — при ненаблюдаемом обучении. Обучение осуществляется путем предъявления примеров, состоящих из наборов входных данных в совокупности с соответствующими результатами при наблюдаемом обучении и без последних — при ненаблюдаемом.

Как правило, выбор структуры нейронной сети осуществляется в соответствии с особенностями и сложностью задачи. Для решения отдельных типов задач уже существуют оптимальные конфигурации. Если задача не может быть сведена ни к одному из известных типов, то приходится решать сложную проблему синтеза новой конфигурации. При этом необходимо помнить, что возможности сети возрастают с увеличением числа нейронов сети, плотности связей между ними и числом слоев. Однако введение обратных связей наряду с увеличением возможностей сети поднимает вопрос о динамической устойчивости сети.

5.2.3. Задачи, решаемые с помощью нейронных сетей

Знания в сети представлены в неявном виде. Нельзя выделить конкретный структурный элемент сети, который представлял бы отдельное правило или сущность предметной области. Знание отражено именно во взвешенных связях между множествами отдельных элементов сети. В данном случае имеют место распределенные знания, которые нельзя представить в виде простого перечисления числовых или символических элементов. По этой причине часто можно встретить утверждение, что НС выполняет несимволическую или субсимволическую (от англ. *sub symbolic*) обработку информации.

В сетях связности знания сохраняются не в декларативном виде, поэтому они не могут быть доступны для интерпретации со стороны какого-либо внешнего процесса. Доступ к знаниям и процесс логического вывода могут быть описаны только в терминах активности сети.

Однако даже в случае, если узлы представляют сущности предметной области, общая картина активности множеств узлов сети может скрывать понятия достаточно высокого уровня, объединяющие определенные аспекты сущностей, представленных узлами. Например, пусть возбуждены узлы, представляющие слова «гонки», «машина» и «водитель». Таким способом может быть представлено понятие «водитель гоночной машины», а может — сам факт вождения гоночной машины. В любом случае такое представление может расцениваться как субсимволическое, поскольку составляющие его узлы не могут быть сведены к какой-нибудь синтаксической структуре, имеющей однозначный, явно выраженный смысл. Точно так же нельзя выполнить и семантический анализ состояния множеств узлов с помощью какого-либо внешнего набора правил.

Любая нейронная сеть используется в качестве самостоятельной системы представления знаний, которая в практических приложениях выступает, как правило, в качестве одного из компонентов системы управления либо модуля принятия решений, передающих результирующий сигнал на другие элементы, не связанные непосредственно с искусственной нейронной сетью [11].

Задачи, решаемые нейронными сетями, можно условно подразделить на несколько основных групп:

- аппроксимации и интерполяции;
- распознавания и классификации образов;
- сжатия данных;
- прогнозирования;
- идентификации;
- управления;
- ассоциации.

В каждом из указанных приложений нейронная сеть играет роль универсального аппроксиматора функции от нескольких переменных, реализуя нелинейную функцию

$$Y = f(x), \tag{5.9}$$

где Y — реализация векторной функции нескольких переменных; x — входной вектор.

Постановки значительного количества задач моделирования, идентификации и обработки сигналов могут быть сведены именно к аппроксимационному представлению.

Для классификации и распознавания образов сеть необходимо обучить важнейшим их признакам, таким как геометрическое отображе-

ние точечной структуры изображения, относительное расположение важнейших элементов образа, компоненты преобразования Фурье и др. В процессе обучения выделяются признаки, отличающие образы друг от друга, создаются классы и эталоны, которые и составляют базу для принятия решений об отнесении образов к соответствующим классам.

При решении задач прогнозирования роль нейронной сети состоит в предсказании будущей реакции системы в зависимости от располагаемой информации о ее предшествующем поведении. Обладая информацией о значениях вектора x в моменты, предшествующие прогнозированию

$$x(k-1), x(k-2), \ldots, x(k-N),$$

сеть вырабатывает решение, каким будет наиболее вероятное значение последовательности $\hat{x}(k)$ в текущий момент k. Для адаптации весовых коэффициентов сети используются фактическая погрешность прогнозирования

$$\varepsilon = x(k) - \hat{x}(k) \tag{5.10}$$

и значения этой погрешности в предшествующие моменты времени.

При решении задач идентификации и управления динамическими процессами нейронная сеть, как правило, выполняет несколько функций. Она представляет собой нелинейную модель этого процесса, обеспечивающую выработку соответствующего управляющего воздействия. Сеть также выступает в роли следящей системы, адаптирующейся к изменяющимся условиям окружающей среды. Очень большое значение, особенно при управлении роботами, имеет функция классификации, реализуемая при выработке решения о дальнейшем развитии процесса.

В задачах ассоциации нейронная сеть играет роль ассоциативного запоминающего устройства. Можно выделить ЗУ автоассоциативного типа, с помощью которых определяется корреляция между отдельными компонентами одного и того же входного вектора, и ЗУ гетероассоциативного типа, средствами которых устанавливается корреляция между двумя различными векторами. Если на вход сети подается неструктурированный вектор (например, содержащий искаженные шумом компоненты или вообще не содержащий отдельные компоненты), нейронная сеть сможет восстановить оригинальный и очищенный от шумов вектор и сгенерировать при этом полную версию ассоциированного с ним вектора.

Важнейшее свойство нейронных сетей, свидетельствующее об их огромном потенциале и широких прикладных возможностях, состоит в параллельной обработке информации одновременно всеми нейронами. Благодаря этой способности при большом количестве межнейронных связей достигается значительное ускорение процесса обработки информации. Во многих ситуациях становится возможной обработка сигналов в реальном масштабе времени.

Очень большое количество межнейронных соединений приводит к тому, что сеть становится нечувствительной к ошибкам, возникающим в отдельных контактах. Функции поврежденных соединений принимают на себя другие элементы, в результате в деятельности сети не наблюдаются заметные нарушения. Это свойство используется, в частности, при поиске оптимальной архитектуры нейронной сети путем разрыва отдельных связей.

Другое не менее важное свойство нейронной сети состоит в способности к обучению и к обобщению полученных знаний. Сеть обладает чертами так называемого искусственного интеллекта. Натренированная на ограниченном множестве обучающих выборок, она обобщает накопленную информацию и вырабатывает ожидаемую реакцию применительно к данным, не обрабатывавшимся в процессе обучения. Несмотря на значительное количество уже известных практических приложений искусственных нейронных сетей, возможности их дальнейшего использования для обработки сигналов не изучены окончательно, и можно высказать предположение, что нейронные сети еще в течение многих лет будут средством развития информационной техники [11].

В настоящее время известно большое количество программных продуктов, выпускаемых рядом фирм и отдельными исследователями и позволяющих конструировать, обучать и использовать нейронные сети для решения практических задач.

Приведем названия некоторых из них, предназначенных для использования на персональных компьютерах в различных операционных средах:

- NeuroSolutions фирмы NeuroDimension Inc.;
- NeuralWorks Professional II Plus с модулем UDND фирмы NeuralWare Inc.;
- Process Advisor фирмы AIWare Inc.;
- NeuroShell 2 фирмы Ward Systems Group;
- BrainMaker Pro фирмы California Scientific Software.

Тестирование показывает, что все перечисленные пакеты имеют практически одинаковые результаты, а их отличия связаны в основ-

ном с возможностями использования различных нейронных структур, критериев оптимизации и алгоритмов обучения сетей, а также с простотой использования и наглядностью представляемой информации [6].

5.3. Эволюционные вычисления

Ведущую роль в области ИИ играют алгоритмы поиска решений. Многие задачи диагностики, прогнозирования, классификации и распознавания образов могут быть интерпретированы как реализация процедуры поиска в абстрактном пространстве решений. Если целью поиска является оптимизация, то возникает вопрос о выборе оптимальных решений.

Процедуру принятия правильного решения в большинстве случаев можно свести к генерации всех возможных вариантов и выбору из числа возможных такого, который с учетом всех разнообразных факторов и противоречивых требований будет в максимальной степени способствовать достижению поставленной цели [2].

Формализация задачи, как правило, предполагает описание всех важных факторов, в наибольшей степени влияющих на достижение цели, порядка их взаимодействия, ограничительных условий и критерия качества принимаемого решения, на основе которого можно осуществлять выбор варианта решения.

Обычно в качестве критерия выступает некая целевая функция, аргументами которой являются количественные характеристики, описывающие факторы, влияющие на достижение цели в решаемой задаче. При этом решению, приводящему к наилучшему результату, как правило, соответствует экстремальное значение целевой функции, т.е. точка ее максимума или минимума. Все возможные комбинации аргументов при этом образуют пространство поиска задачи, размерность которого определяется числом аргументов целевой функции, а каждая из указанных комбинаций образует точку в данном пространстве.

Эволюционные вычисления (от англ. *evolutionary computation*) — термин, используемый для общего описания алгоритмов поиска, оптимизации или обучения, основанных на некоторых формализованных принципах естественного эволюционного процесса [20]. Основное преимущество эволюционных вычислений в этой области заключается в возможности решения многомодальных (имеющих несколько локальных экстремумов) задач с большой размерностью за счет соче-

тания элементов случайности и детерминированности, подобно тому, как это происходит в природной среде.

Детерминированность этих методов заключается в моделировании природных процессов отбора, размножения и наследования, происходящих по строго определенным правилам. Основным правилом при этом является закон эволюции — «выживает сильнейший», обеспечивающий улучшение находимого решения. Другим важным фактором эффективности эволюционных вычислений является моделирование размножения и наследования. Рассматриваемые варианты решений могут по определенному правилу порождать новые решения, которые будут наследовать лучшие черты своих «предков».

В качестве случайного элемента в методах эволюционных вычислений используется моделирование процесса мутации, при котором характеристики того или иного решения могут быть случайно изменены, что приводит к новому направлению в процессе эволюции решений и может ускорить процесс выработки лучшего решения.

История эволюционных вычислений началась с разработки ряда различных независимых моделей эволюционного процесса, среди которых можно выделить три основных направления исследований [20]:

- генетические алгоритмы;
- эволюционные стратегии;
- эволюционное программирование.

Парадигма генетических алгоритмов была предложена Джоном Холландом (J.H. Holland) в начале 1960-х гг. [21]. Основное отличие генетических алгоритмов заключается в представлении любого варианта решения в виде битовой строки фиксированной длины, манипуляции с которой производятся в отсутствие всякой связи с ее смысловой интерпретацией, т.е. в данном случае применяется единое универсальное представление любой задачи.

Эволюционные стратегии, напротив, оперируют объектами, тесно связанными с решаемой задачей. Каждая из альтернатив решения представляется единым массивом численных параметров, за каждым из которых скрывается, по сути, аргумент целевой функции. Воздействие на данные массивы осуществляется, в отличие от генетических алгоритмов, с учетом их смыслового содержания и направлено на улучшение значений входящих в них параметров. Парадигму эволюционных стратегий предложили Реченберг (I. Rechenberg) и Шефель (H.-P. Schwefel) соответственно в 1973 и 1977 гг.

В основе направления эволюционного программирования лежит идея представления альтернатив в виде универсальных конечных авто-

матов, способных реагировать на стимулы, поступающие из окружающей среды. Соответствующим образом разрабатывались и операторы воздействия на них. Идеи эволюционного программирования были предложены в 1966 г. Фогелем (L.J. Fogel), Оуэнсом (A.J. Owens) и Уолшем (M.J. Walsh).

5.3.1. Основные определения

Рассмотрим ряд терминов, которые широко используются в области эволюционных вычислений [2].

Генетические алгоритмы работают с кодами без их смысловой интерпретации, в связи с чем код и его структуру можно описать понятием ***«генотип»***, а его интерпретацию с точки зрения решаемой задачи — понятием ***«фенотип»***. Каждый код представляет собой одну из точек в пространстве поиска. По аналогии с биологическими терминами экземпляр кода называют хромосомой, индивидуумом или особью.

Генетический алгоритм на каждом этапе работает с несколькими точками поиска одновременно. Набор особей, являющийся выражением совокупности этих точек, называется ***популяцией***. Количество особей в популяции называют ее размером, который в классических генетических алгоритмах является фиксированным.

На очередном шаге работы генетический алгоритм обновляет популяцию путем создания новых особей и уничтожения ненужных. Эти шаги, а также соответствующие им популяции, называют ***поколениями*** и обозначают соответствующим номером. Популяция, полученная из исходной популяции после первого шага работы алгоритма, будет первым поколением, после следующего шага — вторым и т.д.

Генерация новых особей в процессе работы алгоритма происходит на основе моделирования процесса размножения. При этом порождающие особи называются родителями, а порожденные — потомками. Родительская пара, как правило, порождает пару потомков. Непосредственная генерация новых кодовых строк из двух выбранных происходит за счет работы оператора скрещивания, называемого также ***кроссинговером*** (от англ. *crossing over* — скрещивание хромосом). Особь, возникшую в результате кроссинговера, иногда называют *кроссоверо*м (от англ. *crossover*).

При формировании новой популяции оператор скрещивания может применяться не ко всем парам родителей. Наилучшая часть этих пар (элита) может переходить в популяцию следующего поколения непосредственно и без изменений. Такая особая стратегия называет-

ся ***элитизмом***. Ее применение способствует сохранению на высоком уровне общего качества популяции, причем элитные особи участвуют еще и в процессе отбора родителей для последующего скрещивания. Частота возникновения элитизма зависит от вероятности применения оператора скрещивания, являющейся одним из параметров генетического алгоритма.

Моделирование процесса мутации новых особей осуществляется оператором мутаций, основным параметром которого также является вероятность мутации.

Поскольку размер популяции фиксирован, то порождение потомков должно сопровождаться уничтожением других особей. Выбор пар родителей из популяции для порождения потомков производится оператором отбора (селекции), а для уничтожения — оператором редукции. Основным параметром их работы обычно является качество особи, определяемое значением целевой функции в точке пространства поиска, соответствующей данной особи.

Операторы отбора, скрещивания, мутации и редукции также называют ***генетическими операторами***.

Критерием остановки работы генетического алгоритма может быть одно из трех событий:

- сформировано заданное пользователем число поколений;
- популяция достигла заданного пользователем качества;
- достигнут некоторый уровень сходимости, при котором дальнейшее улучшение особей происходит чрезвычайно медленно.

Характеристики генетического алгоритма выбираются таким образом, чтобы, с одной стороны, обеспечить наилучшее решение, а с другой — минимизировать затраты времени на его поиск.

5.3.2. Процесс работы генетического алгоритма

Общая схема работы генетического алгоритма представлена на рис. 5.12.

Формирование исходной популяции происходит, как правило, с использованием какого-либо случайного закона, на основе которого выбирается нужное количество точек поискового пространства. Исходная популяция может также быть результатом работы другого алгоритма оптимизации.

Основой порядка применения операторов отбора родительских пар и уничтожения особей является принцип «выживания сильнейшего». При случайном выборе особи для размножения вероятность ее участия в этом процессе вычисляется по формуле

$$P_i = \frac{f_i}{\sum_{j=1}^{N} f_j}, \tag{5.11}$$

где i — номер особи; f_i — значение целевой функции для i-й особи; N — размер популяции.



Рис. 5.12. Процесс работы генетического алгоритма

Целью поиска является нахождение особи с максимумом целевой функции. Очевидно, что одна особь может быть задействована в нескольких родительских парах.

Аналогично может быть решен вопрос уничтожения особей. Вероятность уничтожения должна быть обратно пропорциональна качеству особей, однако обычно просто уничтожаются особи с наихудшим качеством.

Таким образом, выбирая для размножения наилучшие (самые сильные), с точки зрения целевой функции, особи и уничтожая наихудшие (самые слабые), генетический алгоритм постоянно улучшает популяцию, приводя к формированию лучших решений.

Природный процесс наследования, т.е. передача свойств родителей потомкам, моделируется оператором скрещивания.

Простейший оператор скрещивания выполняется в два этапа. Пусть особь представляет собой строку из m элементов. На первом этапе равновероятно выбирается натуральное число k в диапазоне $[1, m-1]$. В соответствии с этим числом, называемым также точкой разбиения, обе исходные строки делятся на две подстроки. На втором этапе строки обмениваются своими подстроками, лежащими после точки разбиения, т.е. элементами с $(k+1)$ по m. В результате получаются две новые строки, унаследовавшие частично свойства обоих родителей (рис. 5.13).

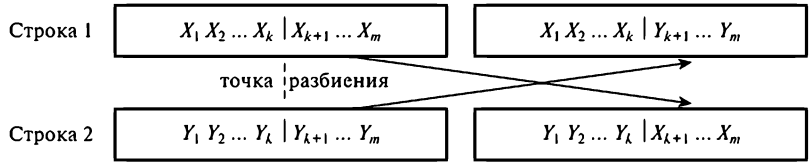


Рис. 5.13. Иллюстрация процесса скрещивания

Для того чтобы обеспечить постоянное появление новых особей в интересах расширения пространства поиска, вероятность применения оператора скрещивания обычно выбирается в пределах от 0,9 до 1.

Количество элитных особей определяется обычно по формуле

$$K = N(1 - P), \tag{5.12}$$

где K — количество элитных особей; N — размер популяции; P — вероятность применения оператора скрещивания.

В случае использования стратегии элитизма скрещиванию подвергаются все выбранные родительские пары, несмотря на то, что вероятность применения оператора скрещивания меньше 1.

Оператор мутации моделирует аналогичный природный процесс. Его применение в генетических алгоритмах обусловлено тем, что исходная популяция, какой бы большой она ни была, охватывает лишь часть пространства поиска. Оператор скрещивания расширяет эту область только до определенной степени, так как использует ограни-

ченный набор значений, заданный исходной популяцией. Внесение случайных изменений в особи позволяет преодолеть это ограничение, значительно сократить время поиска и улучшить качество результата.

Вероятность мутации, в отличие от вероятности скрещивания, выбирается, как правило, достаточно малой. Сам процесс мутации заключается в замене одного из элементов строки на другое значение. Это может быть перестановка двух элементов в строке, замена элемента строки значением элемента из другой строки, в случае битовой строки — инверсия одного из битов и т.п.

В процессе работы алгоритма все указанные ранее операторы применяются многократно и ведут к постепенному изменению исходной популяции. Поскольку операторы отбора, скрещивания, мутации и редукции по своей сути направлены на улучшение каждой отдельной особи, то результатом их работы является постепенное улучшение популяции в целом. В этом и заключается основной смысл работы генетического алгоритма — улучшить популяцию решений по сравнению с исходной.

После завершения работы генетического алгоритма из конечной популяции выбирается особь, которая дает экстремальное (максимальное или минимальное) значение целевой функции и является, таким образом, результатом работы генетического алгоритма. За счет того что конечная популяция лучше исходной, полученный результат представляет собой улучшенное решение.

5.3.3. Пример решения задачи с использованием генетического алгоритма

Рассмотрим применение генетического алгоритма (ГА) для решения задачи поиска максимума одномерной функции [2].

Пусть имеется набор натуральных чисел в интервале [0, 31] и определенная на этом наборе чисел функция $f(x) = x$. Требуется найти максимальное значение функции.

В качестве кода используется двоичный формат аргументов функции (фенотип). Сам код представляет собой двоичную строку из 5 бит (генотип). Целевой функцией непосредственно является рассматриваемая функция $f(x)$, аргументом которой является число, двоичное представление которого в качестве генотипа использует алгоритм.

Установим базовые характеристики ГА: размер популяции — 4, вероятность мутации — 0,01. Процедуру мутации определим как инверсию одного из битов строки, случайно выбранного по равномерному

закону. Оператор скрещивания и отбора аналогичны описанным ранее, стратегия элитизма не используется.

На первом этапе на основе равномерного распределения создается исходная популяция из четырех особей, представленная в табл. 5.2.

Таблица 5.2

№ строки	Код	Значение целевой функции	Вероятность участия в процессе размножения
1	01011	11	11/43
2	10010	18	18/43
3	00010	2	2/43
4	01100	12	12/43

Предположим, что оператор отбора выбрал для производства потомков две пары строк (1, 2) и (2, 4). Работа оператора скрещивания проиллюстрирована в табл. 5.3, причем разбиение каждой пары на подстроки происходит независимо.

Таблица 5.3

№ строки	Родители	Потомки	Значение целевой функции для потомков
1	0 | 1011	00010	2
2	1 | 0010	11011	27
2	100 | 10	10000	16
4	011 | 00	01110	14

Кроме этого, примем, что для младшего бита потомка в строке 3 сработал оператор мутации, вследствие чего данный код изменил свое значение с 10000 на 10001.

Таким образом, популяция за счет порожденных потомков расширилась до восьми особей, представленных в табл. 5.4.

Таблица 5.4

№ строки	Код	Значение целевой функции
Исходная популяция		
1	01011	11
2	10010	18
3	00010	2
4	01100	12

Окончание

№ строки	Код	Значение целевой функции
Порожденные потомки		
5	00010	2
6	11011	27
7	10001	17
8	01110	14

Оператор редукции сокращает популяцию до исходного числа особей, исключив из нее строки с наименьшими значениями целевой функции (1, 3, 4 и 5), после чего популяция первого поколения примет вид, представленный в табл. 5.5.

Таблица 5.5

№ строки	Код	Значение целевой функции	Вероятность участия в процессе размножения
1	10010	18	18/76
2	11011	27	27/76
3	10001	17	17/76
4	01110	14	14/76

На этом шаг работы генетического алгоритма заканчивается. Очевидно, что даже за одну итерацию качество популяции значительно возросло. Если в исходной популяции среднее значение целевой функции было 10,75, а ее минимальное значение составляло 2, то в популяции первого поколения среднее значение возросло до 19, а минимальное значение составило 14. Лучшее же решение увеличилось с 18 до 27 при оптимальном решении 31.

Таким образом, данный пример наглядно иллюстрирует процесс улучшения как популяции в целом, так и наилучшего решения, в частности, в результате работы генетического алгоритма.

5.3.4. Достоинства и недостатки генетических алгоритмов

Цель работы генетического алгоритма заключается в нахождении лучшего по сравнению с имеющимся, а не оптимального решения задачи. Проблема достижения оптимума при этом является вторичной. Другие методы, ориентированные на поиск именно оптимального решения, неприемлемы вследствие чрезвычайной сложности. Именно

в этом кроется причина популярности генетических алгоритмов, хотя, как и всякий другой метод поиска, этот подход не является наилучшим для решения любых задач. Еще одним свойством ГА является невмешательство человека в процесс поиска. Человек может влиять на него лишь опосредованно, задавая параметры ГА.

Эффективность генетического алгоритма для решения каждой конкретной задачи определяется двумя основными факторами: скоростью и устойчивостью работы. Скорость генетического алгоритма оценивается временем, необходимым для выполнения заданного пользователем числа итераций. Если критерием остановки является качество популяции или ее сходимость, то скорость оценивается временем достижения генетическим алгоритмом одного из этих событий. Устойчивость поиска оценивается степенью устойчивости алгоритма к попаданию в точки локальных экстремумов и способностью постоянно увеличивать качество популяции от поколения к поколению.

Достоинства ГА особенно хорошо видны при рассмотрении их в сравнении с традиционными методами:

- генетические алгоритмы работают с кодами, представляющими собой формализованный вид набора параметров, являющихся аргументами целевой функции. Интерпретация этих кодов происходит только перед началом работы алгоритма и после его завершения. В процессе работы ГА манипуляции с кодами происходят независимо от их смыслового содержания, т.е. код рассматривается просто как битовая строка;
- при реализации процедуры поиска ГА обрабатывает одновременно несколько точек поискового пространства, а не переходит последовательно от точки к точке, как в традиционных методах. Это позволяет преодолеть опасность попадания в локальный экстремум полимодальной целевой функции. Использование нескольких точек одновременно значительно снижает вероятность такого события;
- в процессе работы ГА не используют никакой дополнительной информации, кроме данных об области допустимых значений параметров и целевой функции, в произвольной точке, что повышает скорость их работы;
- для порождения новых точек поискового пространства одновременно ГА использует как вероятностные, так и детерминированные правила, что дает значительно больший эффект, чем каждый из этих методов в отдельности.

Недостатки ГА следующие:

- не гарантируется получение оптимального решения;
- эффективно сформулировать задачу, определить критерий отбора хромосом (задать код) и другие параметры ГА может только специалист;
- постановка задачи в терминах ГА не дает возможности проанализировать статистическую значимость получаемого с их помощью решения;
- достаточно высокая вычислительная ресурсоемкость ГА приводит к тому, что в ходе моделирования эволюции многие решения отбрасываются как неперспективные;
- средняя длительность реализации ГА на порядок больше, чем при классических методах решения;
- невысокая эффективность на заключительных фазах моделирования эволюции, объясняемая тем, что механизмы поиска ГА не являются жестко ориентированными на скорейшее попадание в локальный оптимум;
- нерешенными остаются и некоторые другие вопросы, например проблема самоадаптация ГА.

Говоря об эволюционных вычислениях в целом, следует отметить, что они, как и всякий метод, использующий элемент случайности, не гарантируют обнаружения глобального экстремума целевой функции (или оптимального решения) за определенное время. Основное их преимущество состоит в том, что они позволяют найти более «хорошее» решение трудной задачи за меньшее время, чем другие методы. Эволюционные вычисления не являются оптимальным средством для решения любых задач, тем не менее они достаточно эффективны в области инженерного проектирования, планирования, маршрутизации, прогнозирования и т.п.

Следует отметить, что эволюционные вычисления представляют собой скорее подход к решению задач оптимизации, чем алгоритм. Вследствие этого они требуют адаптации к каждому конкретному классу задач путем выбора определенных характеристик и параметров. В настоящее время наблюдается взаимное проникновение различных парадигм эволюционных вычислений и их сращивание в единую концепцию.

К инструментальным средствам, обеспечивающим решения оптимизационных задач с использованием генетических алгоритмов можно отнести следующие программные продукты:

- пакет Evolver 4.0 разработки компании Palisade Corp., представляющий собой дополнение к табличному процессору MS Excel;

- пакет Gene Hunter 1.0 компании Ward System Group;
- пакет Genetic Training Option (GTO) компании California Scientific Software, созданный специально для обучения нейронных сетей и являющийся приложением пакета Brain Maker.

5.4. Комплексный подход к проектированию систем искусственного интеллекта

Комплексное применение рассмотренных интеллектуальных методов обработки информации позволяет существенно повысить эффективность разрабатываемых ИнС.

Возможность использования в рамках одной системы как символического, так и субсимволического подхода (обычно считающихся взаимно исключающими), привела к появлению так называемых гибридных систем. Такие системы потенциально являются мощным инструментом решения сложных проблем, которые не под силу отдельным «чистым» подходам.

Например, генетические алгоритмы могут быть использованы для обучения НС [2], а нечеткая система реализована в виде нечеткой НС [11].

Предстоит еще очень много сделать в теории систем ИИ, прежде чем такие системы смогут в достаточно полной мере эмулировать способность к постоянному совершенствованию, которой обладает человек-эксперт. В этих целях на сегодняшний день исследователям и разработчикам следует разрешить еще целый ряд проблем.

Например, на VIII Международной научно-технической конференции «Интеллектуальные системы» [17] в разработке систем искусственного интеллекта определены следующие основные направления дальнейшего развития в области искусственного интеллекта:

- параллелизм в логическом выводе;
- экспертные системы и вывод в условиях неопределенности;
- аргументация и абдуктивный выход;
- квазиаксиоматические системы;
- машинное обучение и индуктивный вывод;
- мягкие вычисления: нечеткая логика и приближенные вычисления;
- нейронные сети;
- генетические алгоритмы;
- системы когнитивной графики;

- системы семантического Web и онтологии;
- агентно-ориентированное и распределенное решение проблем;
- понимание естественного языка.

5.5. Инструментальные средства представления знаний

Инструментальные средства разработки экспертных систем в зависимости от уровня сложности решаемых задач можно условно подразделить на следующие группы:

- традиционные языки программирования, в том числе языки высокого уровня, объектно-ориентированные и функциональные;
- пустые оболочки или среды разработки экспертных систем. Такая ИС включает собственный встроенный язык организации и манипулирования знаний (их может быть несколько) и язык для взаимодействия с другими языками программирования;
- специализированные системы искусственного интеллекта, содержащие программное ядро, позволяющее реконфигурировать (перенастраивать) систему в зависимости от области знаний.

В зависимости от режима работы экспертные системы и оболочки делятся:

- на динамические, предназначенные для решения задач анализа и синтеза в реальном масштабе времени (к экспертным системам данного вида относится система принятия решений);
- статические, предназначенные для решения задач анализа в реальном масштабе времени и решения задач синтеза в режиме с разделением времени.

5.5.1. Классификация оболочек ЭС

Рассмотрим основные характеристики нескольких типов наиболее популярных экспертных систем:

- статические оболочки для решения задач с данными, неизменными в процессе решения:
  - используемая технология — поверхностная;
  - типы использования правил — только общие;
  - поиск решения — от цели к данным;
  - метод принятия решений — индуктивный подход на основе текущих данных;

  - решаемые задачи — только задачи анализа;
  - получение решений на основе правил, заданных по имеющимся в системе шаблонам;
  - примеры систем: 1-st Class, «Илис», «Интерэксперт»;
- статические оболочки для решения задач анализа и синтеза в режиме с разделением времени:
  - используемые технологии — поверхностный, глубинный, структурный подходы;
  - поиск решений реализован на основе правил, представленных в среде оболочки;
  - использование функций для работы с правилами;
  - поиск решений — от цели к данным, а также от данных к цели; поиск решений вглубь и вширь;
  - примеры систем: KAPPA, NexpertObject, ART, ADC, Clips;
- оболочки для проектирования динамических систем:
  - используемая технология — поверхностный подход;
  - отсутствие системы моделирования;
  - принятие решения — на основе правил общего вида;
  - возможно использование для решения статических задач;
  - примеры систем: Frame work, CubiCalc V2.0;
- оболочки для разработки динамических систем (решения задач анализа и синтеза в реальном масштабе времени):
  - тип технологии — смешанный;
  - использование правил общего и частного вида;
  - наличие системы моделирования;
  - наличие планировщика решений, повышающего эффективность работы системы за счет совокупности имеющихся на текущий момент известных решений;
  - широкий выбор встроенных инструментальных средств;
  - примеры систем: G2, Rethink, RkWorks.

5.5.2. Уровни реализации экспертных систем

Экспериментальный прототип. Система ориентирована на правила общего вида, релевантные решаемой задаче. Количество правил не превышает 50 (обычно от 5 до 15). Система работает неустойчиво. Время, затраченное на разработку, составляет менее 2 мес.

Промышленный образец. Среднее количество используемых правил в таких системах также не более 50, но находится в диапазоне от 20 до 50. В системе имеют место как правила общего вида, так и частные

правила. На частных правилах система работает стабильно. Для доводки системы требуется 3—5 мес.

Промышленная система. Стабильно работающая система, содержащая не менее 100 правил общего и частного вида. Для разработки системы такого уровня требуется 10—12 мес.

Коммерческая система. Стабильно работающая система, используемая в конкретной области человеческой деятельности и предназначенная для продажи. Содержит свыше 100 правил общего характера. На разработку, отладку и тестирование системы требуется не менее 1,5—2 лет.

Контрольные вопросы

1. Назовите методы решения плохо формализованных задач.
2. Какая технология позволяет работать с параметрами, имеющими скорее качественную природу, а не точное значение этих параметров, выражаемое определенной цифрой?
3. Какая технология использует обучение методом «проб и ошибок»?
4. Назовите отличия генетических и эволюционных алгоритмов.
5. Перечислите известные вам оболочки (среды) для создания ЭС.

6. СИСТЕМА МОДЕЛИРОВАНИЯ ЭО КАРРА

Экспертная оболочка КАРРА ориентирована на работу с пользователем, не являющимся профессионалом в области программирования. Необходимость предоставления разработчику ЭС разнообразных инструментов реализации и интерпретации знаний обусловило включение в состав ЭО КАРРА языка высокого уровня, являющегося средством комплексирования компонентов. Выбор средств реализации, анализ их возможностей и возможность модификации алгоритма их функционирования требует от пользователя специальных знаний в области ИИ.

6.1. Представление знаний в ЭО КАРРА

Большинство областей современного знания могут быть в той или иной степени формализованы в соответствии с характером решаемых проблем. В ЭО КАРРА используется универсальная модель — таксономическая схема, в которой связи между объектами представлены в виде отношений типа «часть — целое», «род — вид», «иметь свойство» или «иметь значение». С их помощью происходит формализация структуры области знаний, однако для полного ее определения необходимо также формализовать и процессы, происходящие в ней.

В системе ЭО КАРРА компоненты предметной области представляются такими структурами, как «класс» и «объект» («экземпляр») класса. Отношения между этими структурами представляются «связями», в результате чего получается «иерархия» объектов. Для описания свойств объекта используются «слоты» — своего рода переменные, которые могут принимать определенные значения.

Одним из важнейших свойств ЭО КАРРА является возможность реализации механизма наследования. Общие для классов (объектов) свойства могут быть определены один раз в родительском классе, все потомки которого будут их наследовать. Наследуемые от предков слоты ЭО КАРРА помечает символом «*».

Глобальный для конкретной части иерархии объектов слот передает по наследству и свое значение, что зачастую оказывается лишним. Существует два способа отмены такого наследования:

- сделать слот локальным, для чего в редакторе класса (объекта) выделить требуемый слот, а затем выбрать пункт меню Slots/ Make local. Глобальность слота можно восстановить путем удаления соответствующего локального слота;
- редактировать значение слота на определенном уровне.

В обоих случаях измененное значение наследуется слотами, которые находятся ниже по иерархии.

Процессы, происходящие в модели знаний, реализуются в ЭО KAPPA тремя способами.

Первый способ связан с методами, определяющими поведение слотов в различных ситуациях. Такой способ хранения шаблонов поведения объекта в качестве его атрибутов является примером объектно-ориентированного программирования. Вместо хранения информации непосредственно в слоте система использует метод для вычисления требуемого значения, при этом запрос данных и вычисление происходит лишь в случае необходимости. Методы создаются на специальном языке разработки KAPPA-приложений — KAL (KAPPA Application Language).

Второй способ использует функции. В ЭО KAPPA содержится более 240 стандартных функций для манипулирования БЗ. Кроме этого, пользователь может создавать свои собственные функции с помощью языков KAL и Borland C.

Третий способ реализуется с помощью правил, отражающих шаги процесса и позволяющих делать вывод в соответствии с определенным состоянием объектов. Правила могут определять комплексные взаимодействия между компонентами системы.

Синтаксическая основа правил, методов и функций одинакова. Более подробно синтаксис языка KAPPA-приложений будет рассмотрен ниже.

6.2. Начало работы с ЭО KAPPA

После запуска файла kappa. exe на экране появится окно приветствия, а затем три окна: главное (KAPPA-(untitled)), окно просмотра иерархии объектов (Object Browser) и окно инструментов знаний (Knowledge Tools или KTools) (рис. 6.1).

Главное окно содержит основное меню ЭО KAPPA (File, Edit, Windows и Options) и панель иконок для быстрого доступа к другим окнам оболочки. Кроме того, в этом окне скрыта еще одна часть — комментарии к приложению.

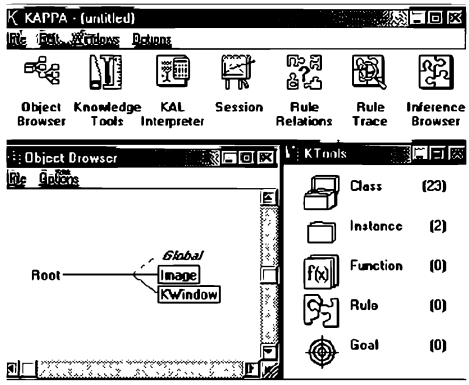


Рис. 6.1. Основные окна ЭО КАРРА

Основное меню содержит четыре пункта:

- работа с файлами;
- редактирование комментария к разрабатываемому приложению;
- манипулирование окнами ЭО КАРРА;
- включение/отключение видимости комментария к приложению.

При открытии или закрытии файла необходимо выбрать его формат: bin (двоичный) или kal (ASCII-файл, подобный txt-файлам). Одновременно может быть открыт только один bin-файл, тогда как kal-файлы можно встраивать друг в друга. При окончании работы в ЭО КАРРА (пункт меню Exit) система потребует подтверждения сохранения рабочего файла, если он был изменен.

Меню Окна (Windows) содержит семь постоянных пунктов, которые позволяют скрыть (Hide) или показать (Show) соответствующие окна ЭО КАРРА:

- просмотр иерархии объектов (Object Browser);
- инструменты знаний (Knowledge Tools);
- интерпретатор языка КАРРА (KAL Interpreter);
- сеанс (Session);
- связи правил (Rule Relations);
- трассировка правил (Rule Tracing);
- просмотр иерархии выводов (Inference Browser).

Открыть эти окна можно также выбором соответствующей иконки в главном окне ЭО КАРРА. Пункт Show Editors в меню окна не доступен до тех пор, пока не открыт ни один из КАРРА-редакторов.

6.3. Окно иерархии объектов (Object Browser)

Это окно предназначено для просмотра и модификации классов и объектов модели, а также связей между ними, наглядно представленных в виде иерархического дерева (рис. 6.2). При запуске ЭО KAPPA даже без открытия конкретного файла в окне Object Browser уже присутствуют следующие обязательные структуры:

- класс Root — фундамент или базовый класс, являющийся родительским по отношению ко всем остальным классам. Этот класс нельзя ни удалить, ни переименовать;
- подкласс Image, предназначенный для хранения объектов, используемых для создания пользовательского интерфейса;
- объект Global — предопределенный объект, созданный для возможности использования сети объектов;
- подкласс KWindow, предназначенный для описания окон сеанса. При запуске ЭО KAPPA в нем уже хранится описание главного окна сеанса — Session.

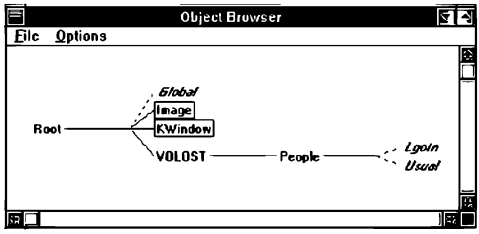


Рис. 6.2. Окно иерархии объектов

Для того чтобы создать в классе подкласс или объект, необходимо выполнить следующие действия:

- выделить класс и щелкнуть по нему правой кнопкой мыши;
- в выпадающем меню выбрать пункт «Добавление Подкласса/Объекта» (Add Subclass/Instance);
- в появившемся окне ввести название подкласса или объекта;
- нажать кнопку OK.

Удаление и редактирование классов и объектов осуществляются аналогичным образом с помощью соответствующих пунктов «Редактировать», «Удалить» (Edit, Delete) выпадающего меню. Однако класс, у которого есть потомки, удалить невозможно!

На рисунке 6.2 показан класс Root, на основании которого создан класс VOLOST, его подкласс People и объекты этого подкласса: Lgotn

и Usual. Объекты отображены курсивом, связи типа «класс — класс» — непрерывной чертой, а связи типа «класс — объект» — штриховой линией. Иерархия объектов направлена слева направо. В ЭО КАРРА каждый слот имеет ряд свойств (опций), для редактирования которых необходимо войти в меню Slots/Edit/Options, предварительно выбрав в редакторе конкретный слот класса (объекта).

В окне Иерархии объектов можно просматривать только часть всей иерархии. Для этого необходимо использовать команду Focus, в результате выполнения которой будет виден только текущий класс и его потомки. Кроме того, окно просмотра можно масштабировать или развернуть на весь экран, используя пункты меню Scale и Options/Fit to Screen соответственно.

6.4. Окно инструментов (Knowledge Tools) и редакторы знаний

Это окно обеспечивает доступ ко всем элементам знаний (классам, объектам, функциям, правилам и целям), а также позволяет создавать, редактировать и удалять любые из них (рис. 6.3).

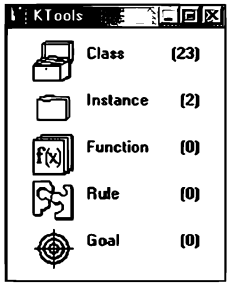


Рис. 6.3. Окно инструментов знаний

Справа от элементов в скобках указано их общее количество в текущей базе знаний. Все типы элементов представлены иконками, при выборе каждой из которых появляются пункты меню: New, Edit, Delete и Rename.

При создании нового элемента система запрашивает его имя, которое должно быть уникальным. Если создается класс или объект, то, кроме его имени, вводится имя родительского класса, который должен существовать (по умолчанию — Root). После этого на экран выводится

окно соответствующего редактора, в котором можно уточнить детали нового элемента. Для редактирования уже существующего элемента необходимо выбрать его имя из списка, после чего активируется окно соответствующего редактора.

В ЭО КАРРА имеются редакторы объектов, классов, слотов, методов, правил, целей и функций. Каждый из них содержит необходимые инструменты (меню, кнопки и т.д.), а также пункт меню Edit со стандартными возможностями — Cut, Paste и Copy.

Редакторы объектов и классов можно также открыть из окна иерархии объектов (Object Browser). Редакторы слотов и методов доступны только из редактора соответствующего объекта или класса, так как отражают их свойства и поэтому жестко к ним привязаны.

Одновременно на экране может быть открыто несколько окон редакторов, однако нельзя открыть одновременно несколько редакторов для одного и того же элемента знаний.

6.4.1. Редактор классов (Class Editor)

Позволяет создавать, удалять и редактировать классы (рис. 6.4). Окно редактора содержит панель меню, окна для просмотра слотов (Slots) и методов (Methods), связанных с классом, а также окно редактирования комментария (Comment). В верхней части окна редактора указан родительский класс текущего класса.

Меню редактора содержит четыре пункта: Update, Edit, Slots и Methods.

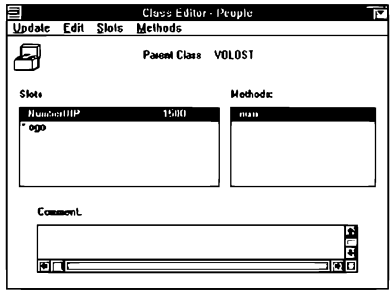


Рис. 6.4. Окно редактора классов

Пункт Update содержит шесть подпунктов: New — создать новый класс; Open — открыть определение класса из списка существующих;

Save — сохранить изменения в отредактированном классе; Rename — переименовать класс; Close — закрыть редактор; Help — справка или помощь.

Для добавления, удаления и редактирования слотов и методов используются пункты меню Slots и Methods.

Пункт Slots имеет пять подпунктов: New — создать новый слот и открыть редактор слота; Edit — открыть редактор выбранного слота; Delete — удалить слот; Rename — переименовать слот; Make Local — запрещает наследование слота (см. подраздел 6.1). Если выбранный слот является наследуемым, то пункты Delete и Rename недоступны.

Подменю Methods содержит пункты, аналогичные одноименным в подменю Slots.

6.4.2. Редактор объектов (Instance Editor)

Редактор объектов по своему внешнему виду, пунктам меню и функционированию полностью аналогичен редактору классов (см. подраздел 6.4.1).

6.4.3. Редактор слотов (Slot Editor)

Редактор слотов (рис. 6.5) обеспечивает создание, просмотр значений, удаление и редактирование слотов. Окно редактора содержит панель меню и два редактируемых поля — Value и Comment.

Поле Value содержит значение или список значений соответствующего слота. Если слот многозначный (список значений), то значения вводятся каждое в отдельной строке либо в одной строке, но взятые в двойные кавычки и разделенные пробелами.

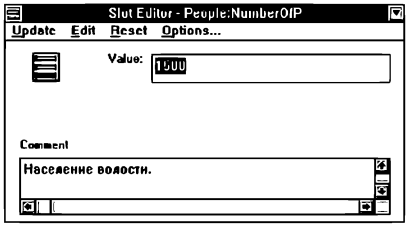


Рис. 6.5. Окно редактора слотов

Пункты Update и Edit аналогичны рассмотренным ранее (см. подраздел 6.4.1). Пункт Reset (Сброс) позволяет сбросить значение слота в наследуемое значение (если это возможно) или в нулевое (началь-

ное) значение. Если слот не имеет локального значения, то этот пункт меню не доступен.

Выбор пункта меню Options (Свойства) вызывает окно редактирования свойств слота (рис. 6.6).

В окне находятся следующие поля редактирования:

- If Needed Method — метод, выполняемый при запросе значения слота, в случае если это значение не определено (не задано);
- Before Change Method — метод, выполняемый непосредственно перед изменением значения слота;

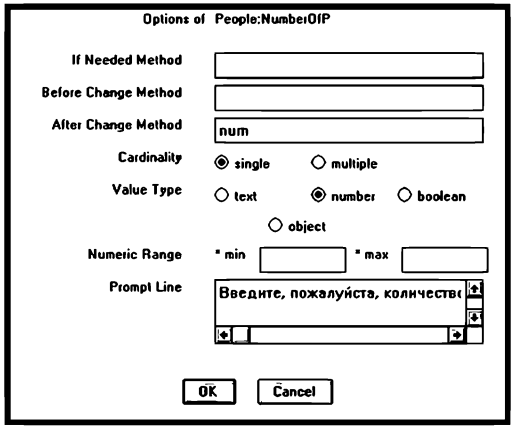


Рис. 6.6. Окно редактирования свойств слота

- After Change Method — метод, выполняемый сразу после изменения значения слота;
- Cardinality — количество значений слота: single (однозначный слот, по умолчанию), multiple (многозначный слот, значения задаются в виде списка);
- Value Type — тип значения: text (текст, строка), number (целое или вещественное число), boolean (логическое — true или false), object (объект — имя класса или экземпляра);
- Numeric Range — ограничения и допустимые значения (если задается в виде списка значений, то эти значения перечисляются через пробел);
- Prompt Line — текст подсказки пользователю.

Примечание. Пока не будет сохранен весь модифицируемый класс, ни одно из изменений, произведенных в редакторе слотов, не повлечет за собой эффекта.

6.4.4. Редактор методов (Method Editor)

Используется для создания, просмотра, удаления и редактирования методов. В окне редактора находятся меню и три поля редактирования (Arguments, Body и Comment), имеющих текстовый формат (рис. 6.7).

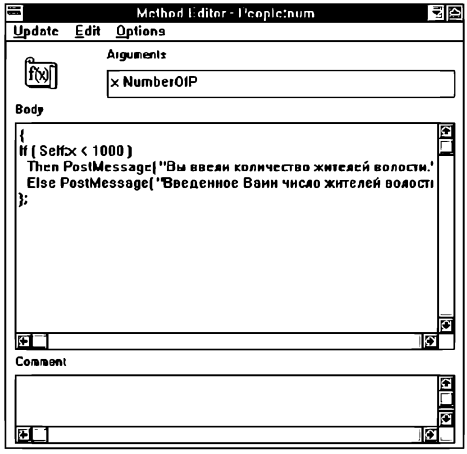


Рис. 6.7. Окно редактора методов

В поле Arguments необходимо указать внутреннюю переменную метода и через пробел — ее конкретное значение для данного слота. Благодаря такой форме записи метод, описанный один раз, можно использовать в любом потомке класса, которому непосредственно принадлежит этот метод.

Поле Body описывает действие, производимое методом.

Меню редактора содержит три пункта: Update, Edit и Options. Пункт меню Update/Check Syntax позволяет проверить метод на наличие синтаксических ошибок.

Примечание. Ни одно из выполненных в редакторе метода изменений не будет иметь эффекта, пока не будет сохранен весь модифицируемый класс.

6.4.5. Редактор функций (Function Editor)

Этот редактор предназначен для создания, просмотра и редактирования функций. Окно редактора содержит три редактируемых поля (Arguments, Body, Comment) и меню (Update, Edit, Options), назначение которых аналогично полям окна редактора методов (см. подраздел 6.4.4).

6.4.6. Редактор правил (Rule Editor)

Этот редактор предназначен для определения (создания), просмотра и изменения правил (рис. 6.8).

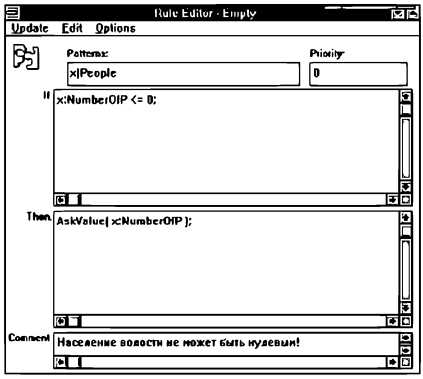


Рис. 6.8. Окно редактора правил

Окно редактора содержит стандартное меню и пять редактируемых полей: Patterns, Priority, If, Then и Comment.

Напомним, что в общем виде правила имеют следующую конструкцию:

If <предпосылка> Then <заключение>.

Как предпосылки, так и заключения представляют собой выражения, состоящие из объектов, их слотов и значений слотов. Эти выражения заносятся в соответствующие поля редактора правил.

В поле Patterns записываются переменные, которые будут использоваться в правилах, причем слева от вертикальной черты вводится имя переменной, фигурирующей в правиле, а справа — класс («старейший» предок), которому принадлежит эта переменная. Правило будет работать только в том случае, когда таким образом будут описаны все входящие в него переменные.

Поле Priority позволяет установить приоритет правила в порядке рассуждений. Прежде чем выполнять правила в соответствии со списком рассуждения (Rule List), каждая стратегия оценки упорядочивает правила из данного списка по их приоритету. По умолчанию приоритет равен нулю и правила в списке записаны в случайном порядке. Диапазон значений приоритета лежит в пределах [–32 000; 32 000]. Чем больше число, тем выше приоритет правила.

6.4.7. Редактор цели (Goal Editor)

Позволяет создавать и редактировать цели (рис. 6.9).

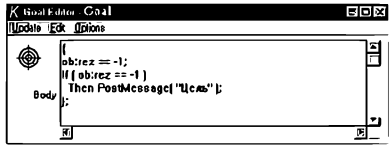


Рис. 6.9. Окно редактора цели

Окно редактора содержит стандартное меню (Update, Edit и Options) и два редактируемых поля (Body и Comment).

Поле Body (тело цели) должно содержать KAL-выражение, определяющее заключение, которое использует механизм рассуждений при попытке проверить, достигнута ли эта цель.

6.5. Окно интерпретатора (KAL Interpreter)

В окне интерпретатора можно выполнить любую корректную функцию, что позволяет быстро проверить правильность ее написания и работы. Находясь в любом окне ЭО КАРРА, где доступно использование выражений в KAL-коде, нажатием правой кнопки мыши активизируем всплывающее меню, с помощью которого можно оперативно выбрать необходимую из имеющихся функций, а также нужный элемент знаний: класс, объект, правило или цель.

Функции разделены на следующие группы:

- манипулирования элементами знаний — Knowledge;
- математические — Math;
- работы со строками — String;
- работы с групповыми объектами и сетями объектов — Lists;
- логические — Logical;
- работы с файлами — File;
- управления рассуждением и взаимодействием с ОС — Control;
- работы с окнами КАРРА и объектами интерфейса — Windows;
- функции, определенные пользователем, — User Functions.

При выборе той или иной группы функций на экране появляется окно Select Command, содержащее список функций указанной группы, а при активации пункта меню All Functions — полный список всех

функций, имеющихся в ЭО. При выделении щелчком мыши одной из функций внизу окна появляется формат ее написания и список аргументов.

KAL-интерпретатор имеет свое меню, содержащее пункты File и Edit. Пункт File содержит только один пункт — Save, позволяющий сохранить содержимое окна в файле. Кроме стандартных пунктов, в подменю Edit имеются еще два пункта: очистки окна — Clear Window и проверки синтаксиса — Check Syntax. Следует помнить, что интерпретатор ограничивает содержимое окна десятью страницами текста и в случае необходимости автоматически удаляет лишнее.

Пример.

В классе People определим слот NumberOfP — количество человек. Тогда и объекты Lgotn, и Usual получат свойство NumberOfP, перешедшее к ним по наследству от родительского класса People (см. рис. 6.2).

Создадим класс Ground, подобный рассмотренному ранее классу People:

```
=>MakeClass (Ground, VOLOST);
Ground
```

Оценим содержимое элемента «People:NumberOfP»:

```
=> People:NumberOfP;
150
```

6.6. Окно сеанса (Session)

Для конечного пользователя KAPPA-приложений это окно является основным элементом интерфейса. Разработчик приложения может настроить окно Session в соответствии с решаемой задачей, что облегчит взаимодействие пользователя с системой. ЭО KAPPA предоставляет пакет стандартных компонентов, с помощью которых можно организовать необходимый интерфейс программы и пользователя. Кроме имеющихся компонентов, можно создавать новые, используя язык программирования Borland C, и связывать их с ЭО KAPPA.

Окно Session (рис. 6.10) состоит из области отображения и меню.

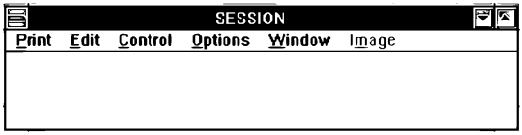


Рис. 6.10. Окно сеанса

В процессе программирования пользователь размещает в области отображения объекты, необходимые ему для создания требуемого ин-

терфейса. В этих целях могут быть использованы функция MakeImage (см. раздел 10), набор графического инструментария — Tool Box или подменю Image.

Окно сеанса имеет два режима работы:

- Layout — режим размещения компонентов, чаще называемый режимом конструктора. Пользователь имеет возможность разместить в окне Session экземпляры объектов, которые понадобятся ему для решения задачи, а также настроить их свойства и методы;
- Runtime — режим выполнения, в котором разработанный интерфейс предоставляется конечному пользователю.

Меню окна сеанса включает следующие пункты:

- пункт Print — для распечатки отдельного объекта или всего экрана;
- пункт Edit, включающий пять подпунктов, доступных только в режиме конструктора:
  - Edit Image — редактировать объект;
  - Rename Image — переименовать объект;
  - Delete Image — удалить объект;
  - Hide Image — скрыть объект (объект сохраняется в БЗ, но не отображается в окне сеанса);
  - Show Hidden Images — восстановление видимости ранее скрытого объекта путем его выбора из списка скрытых объектов;
- пункт Control, позволяющий активизировать решаемые во время сеанса проблемы и получить объяснения рассуждений. Этот пункт содержит три подпункта: Start, Call Function и Explain, аналогичные рассмотренным в подразделе 7.6 (окно Rule Trace);
- пункт Options, позволяющий выбирать режим работы окна сеанса и создавать в нем новые объекты и состоящий из четырех подпунктов:
  - Tool Box — отображает панель инструментов (доступен в режиме конструктора);
  - Image Adjuster — отображает панель регулировки размеров и расположения объектов;
  - Set Tab Stops — устанавливает порядок перехода фокуса управления между объектами при нажатии клавиши табуляции;
  - Layout Mode — осуществляет выбор режима работы окна сеанса: режим выполнения (Runtime) и режим конструктора (Layout);
- пункт Window содержит такие подпункты, как New, Delete, Rename Object, Select, Hide, Unhide и Attributes.

Рассматриваемая версия 1.2 ЭО KAPPA позволяет создавать несколько окон сеанса (с разными именами), но главным всегда остается окно Session.

В режиме Runtime доступны лишь подпункты Hide и Unhide (скрыть и показать окно соответственно). В зависимости от ситуации может быть предложен список окон сеанса. В режиме конструктора доступными становятся все пункты меню Window, включая Delete и Rename Object (удаление и переименование текущего объекта соответственно), а также подпункты New (открыть новое окно) и Attributes (изменение основных свойств текущего окна). Если открыто несколько окон сеанса, то активизируется подпункт Select, обеспечивающий выбор другого окна в качестве текущего.

Подменю Image содержит список предопределенных компонентов — классов активных образов (Active Images Classes), на основании которых могут быть созданы новые объекты — экземпляры. В подменю Image приведены следующие стандартные компоненты: Button, Text, Edit, Transcript, Bitmap, Drawing, State Box, Meter, Line Plot, Slider, Single List Box, Multiple List Box, Check Box Group, Radio Button Group, Check Box и Select Image. Все они более подробно будут рассмотрены в разделе 9.

6.7. Окно связи правил (Rule Relations)

Это окно предназначено для просмотра связи правил, т.е. составных частей формируемой цепи рассуждений (рис. 6.11). При выборе интересующего правила в левой группе (IF Dependencies) отображаются правила, заключения которых являются для него предпосылками, а в правой группе (THEN Dependencies) — правила, для которых заключение рассматриваемого правила является предпосылкой.

Слоты, значения которых анализируются в этих правилах, можно просмотреть посредством выбора пункта List Slots из всплывающего меню при выборе конкретного правила.

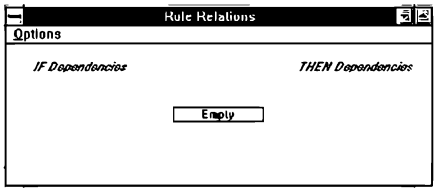


Рис. 6.11. Окно связи правил

Просмотреть связи любого правила, которое отображается в этом окне, можно путем его выбора мышью или с помощью пункта меню Options/Show Relations. Кроме того, из этого окна можно загрузить редактор правил.

6.8. Окно трассировки правил (Rule Trace)

Это окно позволяет просматривать процесс рассуждения в пошаговом режиме и является инструментом отладки (трассировки) системы правил (рис. 6.12).

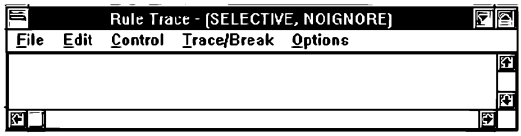


Рис. 6.12. Окно трассировки правил

Главное меню окна содержит следующие пункты (рис. 6.13):

- File — работа с файлами:
  - Save Window Content — сохранить содержимое окна в файле;
- Edit — редактирование содержимого окна:
  - Clear Window — очистить окно (информация в окне безвозвратно удаляется из памяти);
- Options — свойства, содержит подпункты Resize Window Contents, Active Trace и Step Mode, которые будут подробно рассмотрены в подразделе 8.4.3;
- Trace/Break — управление режимами трассировки:
  - Rule Trace и Slot Trace — выбор правил и слотов для трассировки;
  - Rule Break и Slot Break — установка прерываний трассировки;
- Control — управление процессом трассировки:
  - Start — начинает процесс трассировки;
  - Assert — помещает выбранный слот в список подлежащих рассмотрению слотов;
  - SetForwardChainMode — устанавливает режим работы списков подлежащих рассмотрению правил и слотов (см. подраздел 8.4.1);
  - ForwardChain — формирует прямую цепь рассуждений;

— BackwardChain — формирует обратную цепь рассуждений;
— ResetForwardChain — очищает списки подлежащих рассмотрению правил и слотов;
— Call Function — выполняет пользовательскую функцию, предварительно предложив список всех имеющихся;
— Explain — активирует средство объяснений (см. подраздел 6.10).

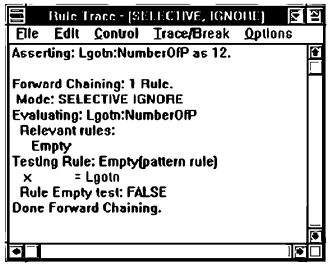


Рис. 6.13. Листинг результатов работы прямой цепи рассуждений в окне трассировки правил

Для запуска процесса трассировки правил необходимо:
1) выбрать в главном окне ЭО КАРРА иконку Rule Trace;
2) в меню Trace/Break выбрать режим Rule Trace;
3) выбрать необходимые правила из списка Available Rules;
4) поместить необходимые правила в список Selected Rules;
5) выбрать требуемый режим трассировки в подменю Control;
6) запустить процесс трассировки путем выбора подпункта Start.

6.9. Окно просмотра иерархии выводов (Inference Browser)

Позволяет просматривать в графической форме правила, активируемые в процессе рассуждения, т.е. направление вывода решения (сразу после окончания процесса рассуждения). Это окно можно использовать для обнаружения ошибок в базе знаний (рис. 6.14).

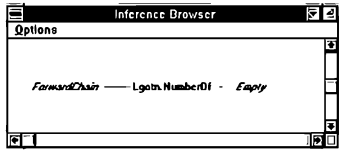


Рис. 6.14. Окно просмотра иерархии выводов

Меню Options выполняет те же функции, что и одноименный пункт в окне Object Browser (см. подраздел 6.3) за исключением пункта меню Step Mode, позволяющего запустить режим формирования обратной цепи рассуждений (см. подраздел 8.5).

Кроме того, при выборе левой кнопкой мыши правила в окне выводится меню Edit/Show Relations, в котором Edit вызывает редактор правил с текстом текущего правила, а Show Relations активирует окно Rule Relations для выбранного правила.

6.10. Средство объяснений ЭО КАРРА

Использование данного средства позволяет поставить перед ЭО вопрос: «Почему слот имеет такое значение?» Такая возможность существует для любого однозначного слота. Этим средством можно воспользоваться одним из следующих способов:

- в окне Session выбрать пункты Control, Explain и выделить интересующий слот;
- проделать те же операции в окне Rule Trace;
- воспользоваться функцией Explain интерпретатора языка KAL.

Окно объяснений (рис. 6.15) содержит текст объяснений и две кнопки: Quit (выход) и Explain (объяснить). Последняя позволяет запрашивать объяснение значения других слотов, а информация помещается в то же окно. Если значение слота изменено в результате работы правила, то указывается это правило и относящийся к нему комментарий.

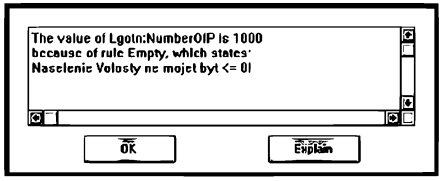


Рис. 6.15. Средство объяснений

Контрольные вопросы

1. Перечислите известные вам оболочки (среды) для создания ЭС.
2. Какими структурами представлены компоненты предметной области в ЭО КАРРА? Назовите их характеристики.

3. Определите способы активизации процессов модели знаний в ЭО КАРРА.
4. Определите назначение инструментов ЭО КАРРА.
5. Каким образом в окне иерархии вывода иллюстрируется механизм логического вывода?
6. Как используется окно сеанса при реализации интерфейса пользователя в ЭО КАРРА?
7. Как функционируют редакторы объектов и классов?
8. Перечислите элементы окна свойств слота.
9. Перечислите функции редакторов в инструментарии ЭО КАРРА.
10. Определите использование понятия «метод» в инструментарии ЭО КАРРА.

7. СТАНДАРТНЫЕ ФУНКЦИИ ЭО КАРРА

В зависимости от назначения основные функции ЭО КАРРА можно условно разделить на ряд групп, в соответствии с которыми они и будут рассмотрены. Специфика языка KAL и ЭО КАРРА не позволяет представить синтаксис функций в классической нормальной Бэкусовской форме. Чтобы не усложнять задачу, просто оговорим ряд правил их написания и вызова.

Список объектов представляет собой последовательность пар вида

<имя объекта> | <имя класса>,

разделенных пробелами и заключенных в квадратные скобки. В случае если список пуст, квадратные скобки все равно указываются.

Синтаксис списка аргументов и списка формальных параметров аналогичен синтаксису списка объектов. Список заключен в квадратные скобки, аргументы разделяются пробелами и не являются обязательными.

Пример.

[People | Lgotn People | Usual] — список объектов из двух элементов.

[1 2 78 56] — список аргументов.

[] — пустой список (объектов, аргументов и пр.).

При обычной записи аргумента на его месте необходимо указать требуемое значение параметра либо значение NULL.

Если аргумент заключен в квадратные скобки — [<имя класса>], то он не является обязательным и его можно опустить.

Запись вида {<вариант_1> ¦ <вариант_2>} означает возможность равнозначного выбора одного из указанных вариантов.

7.1. Функции манипулирования знаниями

7.1.1. Функции работы с классами

Class? (<имя класса>); — определяет, существует ли класс с указанным именем.

DeleteClass (<имя класса>); — удаляет класс из текущей базы знаний.

GetClassComment (<имя класса>); — возвращает строку — комментарий к указанному классу.

MakeClass (<имя класса> [<имя родительского класса>]); — создает новый класс в текущей базе знаний.

RenameClass (<имя класса>, <новое имя класса>); — переименовывает существующий класс.

SetClassComment (<имя класса>, <строка>); — добавляет строку — комментарий к указанному классу.

ToInstance (<имя класса>); — преобразует класс в объект с тем же именем.

WriteAllClasses (); — сохраняет содержание всех классов в предварительно открытом ASCII-файле.

WriteClass (<имя класса>); — сохраняет содержание класса в предварительно открытом ASCII-файле.

7.1.2. Функции работы с объектами

По своему написанию, порядку вызова и спискам аргументов функции работы с объектами практически полностью аналогичны функциям работы с классами. Для вызова функций достаточно в их названиях слово Class (класс) заменить словом Instance (объект), а в аргументах — <имя класса> на <имя объекта>.

7.1.3. Функции работы с иерархией объектов

CountAllInstances (<имя класса>); — возвращает количество всех объектов заданного класса и всех его подклассов.

CountAllSubClasses (<имя класса>); — возвращает число всех подклассов указанного класса и их подклассов.

CountInstances (<имя класса>); — возвращает число объектов указанного класса.

CountSubClasses (<имя класса>); — возвращает количество подклассов указанного класса.

ForAll (<список объектов>, <выражение>) — выполняет действие, определяемое выражением, для всех объектов, указанных в списке.

GetDepth (<имя класса>); — возвращает глубину иерархии ниже заданного класса, т.е. число уровней дочерних объектных структур (подклассов и объектов).

GetInstanceList (<имя класса>, <имя объекта>: <имя многозначного слота>); — сохраняет список имен непосредственных объектов заданного класса в заданной переменной (многозначный слот).

GetParent ({<имя объекта> ¦ <имя класса>}); — возвращает имя родительского класса указанного объекта или класса.

GetSubClassList (<имя класса>, <имя объекта>: <имя многозначного слота>); — сохраняет список имен непосредственных подклассов заданного класса в заданной переменной (многозначный слот).

HideBrowserInstances (<имя класса>); — скрывает все объекты указанного класса в окне иерархии объектов.

HideBrowserSubClasses (<имя класса>); — скрывает все подклассы указанного класса в окне иерархии объектов.

IsAKindOf? (<имя объекта>, <имя класса>); — определяет, является ли заданный объект потомком указанного класса. Возвращает true или false.

IsThereAny? (<список объектов>, <шаблон>) — просматривает все объекты данного класса и возвращает первый объект, соответствующий заданному образцу.

MoveHierarchy (<имя класса>, <имя нового родительского класса>); — переподчиняет указанный класс со всеми его подклассами и объектами другому родительскому классу.

MoveInstance (<имя объекта>, <имя нового родительского класса>); — переподчиняет указанный объект другому родительскому классу.

PrintHierarchy (<имя класса>); — распечатывает иерархию классов и объектов, начиная с указанного класса.

ShowBrowserInstances (<имя класса>); — выводит все объекты указанного класса в окно иерархии объектов.

ShowBrowserSubClasses (<имя класса>); — выводит все подклассы указанного класса в окно иерархии объектов.

WriteHierarchy (<имя класса>); — записывает иерархию классов и объектов, начиная с указанного класса, в заранее открытый ASCII-файл.

7.1.4. Функции работы со слотами

AskValue (<имя объекта>, <имя слота>); — выводит на экран окно диалога с запросом у пользователя значения однозначного слота.

Assert (<имя объекта>, <имя слота>); — добавляет пару «объект : слот» в список слотов для формирования прямой цепи рассуждений.

DeleteSlot (<имя объекта>, <имя слота>); — удаляет слот из объекта.

Explane (<имя объекта>, <имя слота>); — показывает, что побудило слот иметь текущее значение.

GetOwner (<имя объекта>, <имя слота>); — возвращает имя первого предка указанного объекта по иерархии, в котором был определен указанный слот.

GetSlotComment (<имя объекта>, <имя слота>); — возвращает строку (комментарий к указанному слоту).

GetSlotList (<имя объекта>, <имя объекта>: <имя многозначного слота>); — сохраняет список имен локальных слотов или слотов, имеющих локальные значения или опции заданного объекта, в переменной с заданным именем (многозначный слот).

GetSlotOption (<имя объекта>, <имя слота>, <имя свойства>); — возвращает значение указанного свойства слота.

GetValue (<имя объекта>, <имя слота>); — возвращает значение слота.

KnownValue? (<имя объекта>, <имя слота>); — определяет, имеет ли значение указанный слот.

LocalValue? (<имя объекта>, <имя слота>); — определяет, является ли значение указанного слота локальным.

MakeSlot (<имя объекта>, <имя слота>, <выражение>); — создает новый слот указанного объекта и присваивает ему значение, определяемое выражением.

RenameSlot (<имя объекта>, <имя слота>, <новое имя слота>); — переименовывает существующий слот.

ResetSlotOption (<имя объекта>, <имя слота>, <имя свойства>) — присваивает значение по умолчанию выбранному свойству слота.

ResetValue (<имя объекта>, <имя слота>); — присваивает указанному слоту значение, передаваемое по наследству, а при его отсутствии — NULL.

SetSlotComment (<имя объекта>, <имя слота>, <строка>); — добавляет строку — комментарий к указанному слоту.

SetSlotOption (<имя объекта>, <имя слота>, <имя свойства>, <выражение>); — присваивает указанному свойству слота значение, определяемое выражением.

SetValue (<имя объекта>, <имя слота>, <выражение>); — присваивает указанному слоту значение, определяемое выражением.

Slot? (<имя объекта>, <имя слота>); — определяет существование указанного слота в данном объекте.

<имя объекта>: <имя слота> += <выражение>; — увеличивает значение слота на величину, определяемую выражением.

<имя объекта>: <имя слота> = <выражение>; — присваивает однозначному слоту значение, определяемое выражением.

7.1.5. Функции работы с методами

DeleteMethod (<имя объекта>, <имя метода>); — удаляет ранее созданный метод, принадлежащий указанному объекту.

GetMethodComment (<имя объекта>, <имя метода>); — возвращает строку — комментарий к указанному методу.

MakeMethod (<имя объекта>, <имя метода>, <список аргументов>, <тело метода>); — создает в заданном объекте новый метод, определяемый телом метода и списком аргументов. Аргументы в списке разделяются пробелами и не являются обязательными.

Method? (<имя объекта>, <имя метода>); — определяет наличие указанного метода у объекта.

RenameMethod (<имя объекта>, <имя метода>, <новое имя метода>); — переименовывает существующий метод.

SendMessage (<имя объекта>, <имя метода> [<аргументы метода>]); — выполняет указанный метод объекта. При наличии у вызываемого метода аргументов они перечисляются через запятую.

SetMethodComment (<имя объекта>, <имя метода>, <строка>); — добавляет строку — комментарий к указанному методу;

ShowMethod (<имя объекта>, <имя метода>); — выводит текст тела указанного метода в окно интерпретатора.

7.1.6. Функции работы с правилами

ActivateRule (<имя правила>); — добавляет правило в список активных правил.

BackwardChain ([NOASK,] <имя цели>, <список активных правил>); — формирует обратную цепь рассуждений.

DeactivateRule (<имя правила>); — удаляет правило из списка активных правил.

DeleteRule (<имя правила>); — удаляет правило из базы знаний.

ForceCompileRuleNet (<список правил>); — указывает список правил, которые будут использоваться при формировании цепочки вывода. Список правил представляет собой имена правил, перечисленные через запятую.

ForwardChain ([NOASSERT,] <имя цели >, <список активных правил>); — формирует прямую цепь рассуждений.

GetRuleComment (<имя правила>); — возвращает строку — комментарий к указанному правилу.

MakeRule (<имя правила>, [<имя объекта> | <имя класса>,] If <выражение>, Then <выражение>); — создает в текущей базе знаний новое правило. Аргумент вида <имя объекта> | <имя класса> является шаблоном переменной правила (patterns, см. подраздел 6.4.6).

RenameRule (<имя правила>, <новое имя правила>); — переименовывает существующее правило;

ResetForwardChain (); — очищает список формирования прямой цепи рассуждений.

Rule? (<имя правила>); — определяет существование правила с заданным именем.

SetActiveRules (<список правил>); — помещает правила из указанного списка в список активных правил.

SetForwardChainMode (<стратегия>, <режим>); — устанавливает стратегию и режим формирования прямой цепи рассуждений. Более подробно эта функция и задаваемые ей стратегии и режимы будут рассмотрены в подразделе 8.4.1.

SetRuleComment (<имя правила>, <строка>); — добавляет строку — комментарий к указанному правилу.

SetRulePriority (<имя правила>, <приоритет>); — устанавливает приоритет правила.

WriteAllRules () — сохраняет все правила в предварительно открытом для записи ASCII-файле.

WriteRule (<имя правила >); — сохраняет правило в предварительно открытом ASCII-файле.

7.1.7. Функции работы с целями

DeleteGoal (<имя цели>); — удаляет цель из базы знаний.

GetGoalComment (<имя цели>); — возвращает строку — комментарий к указанной цели.

Goal? (<имя цели>); — проверяет существование цели.

MakeGoal (<имя цели>, <выражение>); — создает в текущей базе знаний новую цель, определяемую указанным выражением.

RenameGoal (<имя цели>, <новое имя цели>); — задает новое имя цели;

SetGoalComment (<имя цели>, <строка>); — добавляет строку — комментарий к указанной цели.

TestGoal (<имя цели>); — проверяет достижение заданной цели.

7.2. Математические функции

Помимо стандартных арифметических операций, в ЭО КАРРА имеется набор специальных функций для элементарных математических вычислений.

Abs (<выражение>); — абсолютное значение аргумента.

Acos (<выражение>); — вычисляет арккосинус аргумента.

Asin (<выражение>); — вычисляет арксинус аргумента.

Atan (<выражение>); — вычисляет арктангенс аргумента.

Ceil (<выражение>); — возвращает ближайшее целое число, большее, чем значение аргумента.

Cos (<выражение>); — вычисляет косинус аргумента.

Floor (<выражение>); — возвращает ближайшее целое число, меньшее, чем значение аргумента.

Integer? (<выражение>); — определяет, является ли значение аргумента целым числом.

Log (<выражение>); — вычисляет натуральный логарифм аргумента.

Negative (<выражение>); — возвращает отрицательное значение аргумента, если оно является числом. Аналогично знаку « — ».

Number? (<выражение>); — определяет, является ли значение аргумента числом.

Sin (<выражение>); — вычисляет синус аргумента.

Sqrt (<выражение>); — вычисляет корень квадратный аргумента.

Tan (<выражение>); — вычисляет тангенс аргумента.

<выражение>^<выражение> — возводит число в заданную степень.

7.3. Функции работы со строками

FindSubString (<строка>, <подстрока>, <размер подстроки>); — возвращает позицию первого вхождения подстроки заданного размера в строку. Если подстрока не найдена, то возвращается false. Если размер подстроки задан отрицательным числом, то строка анализируется с конца.

FormatValue (<шаблон>, <список значений>); — возвращает строку, содержащую список значений, отформатированных в соответствии с заданным шаблоном. Значения в списке перечисляются через запятую. Шаблонная строка содержит специальные кодовые символы, определяющие формат представления значений при выводе в соответствии с ANSI C стандартом.

Пример.

FormatValue ("Uch%d", i) при i=1 возвратит строку «Uch1».
FormatValue ("Uch%d ", i, j) при i=1, j=5 возвратит «Uch1 Uch5».

StringLength (<строка>); — возвращает длину строки в символах.

SubString (<строка>, <начальная позиция>, <конечная позиция>); — возвращает подстроку, начиная с начальной позиции и заканчивая конечной, в случае неудачи — ERROR.

TextCase (<регистр>, <строка>); — конвертирует все символы строки в указанный регистр (нижний — Lower или верхний — Upper).

<строка> # <строка>; — формирует новую текстовую строку, являющуюся результатом сложения (конкатенации) двух строк-аргументов.

<строка> #= <строка>; — осуществляет сравнение двух строк и возвращает true при их равенстве и false — в противном случае.

7.4. Функции работы со списками

По названиям функций для работы со списками легко судить о выполняемых ими операциях. В случае необходимости любая функция работы со списками вместе со всеми аргументами может быть взята из всплывающего меню KAL-интерпретатора или любого другого редактора, в котором возможно использование KAL-выражений.

7.5. Логические функции

ЭО КАРРА обладает стандартным набором логических функций.

<выражение> And <выражение>; — конъюнкция, логическое «и».

<выражение> Or <выражение>; — дизъюнкция, логическое «или».

<выражение> Xor <выражение>; — исключающее «или».

Not (<выражение >); — инверсия, отрицание, логическое «не».

Помимо этого достаточно широко распространена еще одна функция.

Null? (<выражение>); — возвращает true, если значение указанного выражения равно NULL, и false — в противном случае.

7.6. Функции работы с файлами

CloseReadFile (<имя файла>) и CloseWriteFile (<имя файла>) — закрывают файл, открытый для чтения или для записи соответственно.

InterpretFile (<имя файла>); — читает и оценивает KAPPA-выражения из существующего текстового ASCII-файла.

OpenReadFile (<имя файла>) или OpenWriteFile (<имя файла>) — открывают файл для чтения или для записи соответственно.

ReadWord ([<количество символов>]); — читает указанное количество символов из открытого ASCII-файла как одно слово, по умолчанию читается слово стандартной длины.

WriteAll (<имя файла>); — сохраняет все функции, классы, экземпляры классов, правила и цели в предварительно открытом ASCII-файле, который затем (после сохранения) можно прочитать с помощью функции InterpretFile.

7.7. Функции управления

For <переменная цикла> [<начальное значение> <конечное значение> <шаг>] <тело цикла>; — оператор цикла со счетчиком.

If (<условие>) Then (<выражение>) [Else <выражение>]; — условный оператор.

PostError (<строка>); — при возникновении ситуации, несовместимой с дальнейшей нормальной работой, приостанавливает выполнение программы и возвращает строку — сообщение об ошибке.

While (<условие>) <тело цикла>; — оператор цикла с предусловием.

Пример.

```
For i [1 (CountAllInstances (ZemlyOfVOLOST)) 1]
DeleteInstance (FormatValue ("Uch%d", i));
```

7.8. Функции работы с окнами

FreezeWindow (<имя окна>); — делает недоступными все элементы управления окном.

GetBrowserFocus (); — возвращает имя класса, находящегося в фокусе управления окна просмотра объектов.

HideBrowserInstances (<имя класса>); — скрывает экземпляры указанного класса.

HideBrowserSubClasses (<имя класса>); — скрывает подклассы указанного класса.

HideWindow (<имя окна>); — закрывает окно KAPPA-приложения.

IconifyWindow (<имя окна>); — сворачивает окно в иконку в трее.

MaximizeWindow (<имя окна>); — разворачивает окно на весь экран.

PositionWindow (<имя окна>, <X>, <Y>, <ширина>, <высота>); — устанавливает левый верхний угол указанного окна в позицию, определяемую координатами (X, Y), а его размеры — в соответствии с заданными шириной и высотой.

RemoveWindowMenu (<имя окна>); ResetWindowMenu (<имя окна>); — удаляет и восстанавливает строку меню указанного окна.

SetBrowserFocus (<имя класса>); — передает фокус управления классу с именем, указанным в качестве аргумента.

7.9. Функции работы с компонентами

Как уже упоминалось ранее (см. подраздел 6.6), интерфейс пользователя ЭО KAPPA строится из элементов, являющихся экземплярами стандартных объектов, называемых компонентами. В соответствии с принципами ООП каждый их компонентов обладает своими свойствами и методами. Кроме этого, практически для каждого из них существует набор функций, позволяющий динамически в процессе выполнения программы изменять их свойства.

Эти реализующие работу с компонентами функции будут более подробно рассмотрены в разделе 10 применительно к тем компонентам, для которых они предназначены.

7.10. Функции, определенные пользователем

Как и в большинстве других интегрированных сред разработки, в ЭО KAPPA существует возможность использования дополнительных функций, определяемых пользователем. Для работы с такими функциями применяется специальный инструментарий («функции работы с функциями»), рассмотренный ниже.

DeleteFunction (<имя функции>); — удаляет функцию, ранее определенную пользователем, с указанным именем из текущей базы знаний.

Function? (<имя функции>); — определяет существование функции с заданным именем.

GetFunctionComment (<имя функции>); — возвращает строку — комментарий к указанной функции;

MakeFunction (<имя функции>, <список формальных параметров>, <тело функции>); — создает новую функцию, определяемую телом функции и списком формальных параметров. Параметры в списке разделяются пробелами и не являются обязательными. В случае их отсутствия вызов функции будет выглядеть следующим образом:

MakeFunction (<имя функции>, [], <тело функции>);

RenameFunction (<имя функции>, <новое имя функции>); — переименовывает существующую функцию.

SetFunctionComment (<имя функции>, <строка>); — добавляет строку (комментарий к указанной функции).

ShowFunction (<имя функции>); — выводит текст тела функции в окно интерпретатора.

Контрольные вопросы

1. Какой инструмент ЭО КАРРА предназначен для описания всех используемых функций?
2. Какие функции ЭО КАРРА предназначены для манипулирования знаниями?

8. РАБОТА С ПРАВИЛАМИ В ЭО КАРРА

Для работы с правилами в ЭО КАРРА можно использовать обычные программы, обращаясь конкретно к каждому необходимому правилу, или применить встроенные средства:

- механизм формирования прямой цепи событийно-управляемых рассуждений (ForwardChain), в процессе работы которого необходимо по известным предпосылкам достичь требуемой цели;
- механизм формирования обратной цепи рассуждений, управляемых целями (BackwardChain), при котором необходимо восстановить причину (предпосылку) по известному результату (цели).

Рассмотрим более подробно то, что необходимо знать в этой области разработчику баз знаний в ЭО КАРРА.

8.1. Создание и редактирование правил

Напомним, что для начала работы с правилами необходимо выбрать пункт меню Rule или соответствующую иконку в окне инструментов (KTools). На экране появится окно (рис. 8.1) с полем ввода имени правила, которое необходимо создать либо отредактировать.

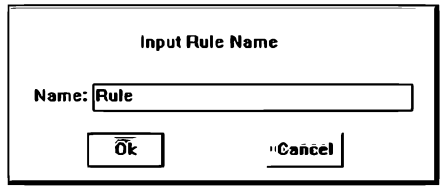


Рис. 8.1. Окно ввода имени правила

После ввода имени правила на экране отображается окно редактора правил (см. рис. 6.8), в котором необходимо ввести заново или отредактировать условие (If), при истинности которого выполняются требуемые действия (Then), приоритет правила (Priority) и определить переменные (Patterns).

8.2. Формирование списка правил

Как уже отмечалось, связь между правилами при формировании цепочки рассуждений осуществляется посредством слотов. Для этого прежде всего необходимо объявить все используемые в правилах слоты с помощью функции Assert (см. подраздел 7.1.4).

Список правил, которые будут использоваться при формировании цепи рассуждений, можно задать следующими тремя способами:

1) указать явно, перечислив имена правил при вызове функций ForwardChain или BackwardChain;

2) занести имена необходимых правил в многозначный слот, имя которого указать в качестве аргумента при вызове функций ForwardChain или BackwardChain;

3) сформировать список правил с помощью функции ForceCompileRuleNet, а затем вызвать функции ForwardChain или BackwardChain без указания аргумента.

Пример.

Способ 1.

```
ForwardChain (NULL, Rule1, Rule2, Rule3);
```

Способ 2.

```
MakeSlot (Global, Rules1_3, (Rule1, Rule2, Rule3));
ForwardChain (NULL, Global: Rules1_3);
```

Способ 3.

```
ForceCompileRuleNet (Rule1, Rule2, Rule3);
ForwardChain (NULL);
```

В зависимости от получаемых результатов состав рассматриваемых правил в процессе рассуждения можно изменить с помощью функций ActivateRule, DeactivateRule и SetActiveRules.

8.3. Создание и редактирование цели

Для работы с целью необходимо выбрать пункт меню Goal или соответствующую иконку в окне инструментов (KTools). На экране появится окно (рис. 8.2) с полем ввода имени цели, которую необходимо создать либо отредактировать.

После этого в окне редактора цели (см. рис. 6.9) необходимо ввести новое или отредактировать существующее тело цели, т.е. условие, при котором прекращается выполнение цепочки рассуждений, а цель считается достигнутой.

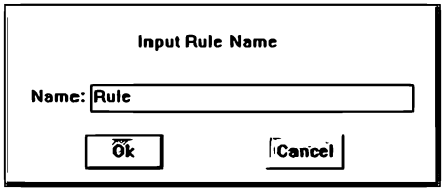


Рис. 8.2. Окно задания цели для обратной цепочки вывода

8.4. Рассуждения в прямом направлении

В процессе формирования прямой цепи рассуждений используется список правил (Rule List) и список слотов (Agenda), представляющий собой список пар вида «объект : слот», назначенных к рассмотрению при рассуждении в прямом направлении. Во время рассуждения состав этих списков взаимосвязано меняется. Слоты, значения которых изменяются в результате выполняемых правил, автоматически добавляются в список Agenda. По умолчанию процесс формирования прямой цепи рассуждений продолжается до тех пор, пока не опустеет один из списков — Agenda или Rule List.

8.4.1. Стратегии принятия решения

Формирование прямой цепи рассуждений может быть осуществлено различными способами, зависящими от режима обработки списка слотов и стратегии разрешения конфликтных ситуаций.

В ЭО KAPPA предусмотрены следующие режимы обработки списка слотов:

- IGNORE — при использовании этого режима в первую очередь будет обрабатываться элемент списка слотов, появившийся в этом перечне самым последним (т.е. очередь будет перестраиваться);
- NOIGNORE — в этом режиме элементы списка слотов обрабатываются в порядке очереди независимо от момента его занесения в список.

В процессе рассуждения не исключены конфликтные ситуации. Например, если одному элементу списка слотов соответствует несколько правил, то какое из них будет необходимо выполнить в первую очередь? Для разрешения подобных проблем предусмотрены четыре различные стратегии:

- SELECTIVE Evaluation — выборочная оценка (по умолчанию);
- DEPTHFIRST — оценка по глубине;
- BREADTHFIRST — широтная оценка;
- BESTFIRST Evaluation — оценка лучших.

Стратегия выборочной оценки считается наиболее эффективной, поскольку подразумевает выбор только одного, наиболее успешного направления рассуждения. Если одно правило из списка стало истинным, то все остальные правила удаляются из Rule List. Список слотов должен быть освобожден до того, как заработает следующее правило из списка.

Глубинная стратегия является исчерпывающей, так как раскрывает все возможные значения вновь появившегося элемента в списке слотов. Отличие глубинной стратегии от выборочной состоит лишь в том, что при ее использовании список правил после каждого успешного выполнения одного правила не очищается. Новые правила добавляются к началу списка в соответствии с их приоритетом. Если не пусты оба списка (Rule List и Agenda), то приоритет отдается очередному элементу списка слотов. Новые слоты добавляются в начало списка слотов.

Широтная стратегия также является исчерпывающей и раскрывает все возможные значения нового факта. Новые правила добавляются к концу списка в соответствии с их приоритетом. Первым применяется правило, которое может производить новые пункты в списке слотов. Если не пусты оба списка (Rule List и Agenda), то приоритет отдается очередному элементу списка правил.

При стратегии оценки лучших новые правила добавляются к правилам, уже находящимся в списке, в соответствии с их приоритетом. Первым применяется правило, которое может производить новые пункты в списке слотов. Если не пусты оба списка (Rule List и Agenda), то приоритет отдается очередному элементу списка слотов.

Выбор стратегии и режима обработки слотов осуществляется в окне Set Forward Chain Modes (рис. 8.3).

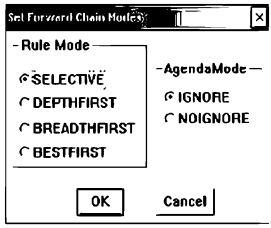


Рис. 8.3. Установка стратегии и режима слотов

Кроме этого, их можно задать путем вызова функции

SetForwardChainMode (<режим>, <стратегия>).

На рисунках 8.4—8.7 представлен пример прямой цепи вывода для трех правил, сформированных с использованием разных стратегий.

Пример.

Rule1 (priority=0): If ob: s1 And ob: s3 == 4 Then ob: rez = 1
Rule2 (priority=1): If Not (ob: s2) And ob: s3 == 4 Then ob: rez = −1
Rule3 (priority=0): If ob: rez == −1 Then ob: v = 5

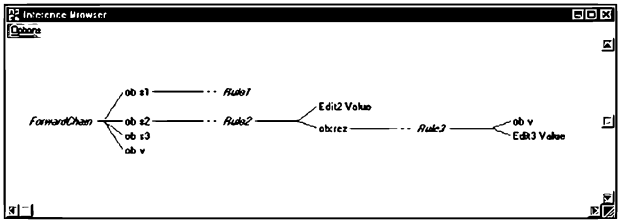


Рис. 8.4. Выборочная стратегия (selective)

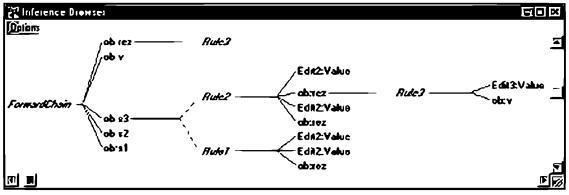


Рис. 8.5. Глубинная стратегия (depthfirst)

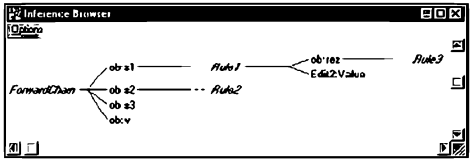


Рис. 8.6. Широтная стратегия (breadthfirst)

В процессе формирования прямой цепи рассуждений отслеживаются все отношения между правилами, участвующими в выводе (рис. 8.8). Из рисунка следует, что правило Rule3 будет проверено только после проверки правил Rule2 и Rule1, и в зависимости от того, какое из этих правил истинно либо имеет больший приоритет (если оба истинны), будет соответственно выполняться правило Rule3.

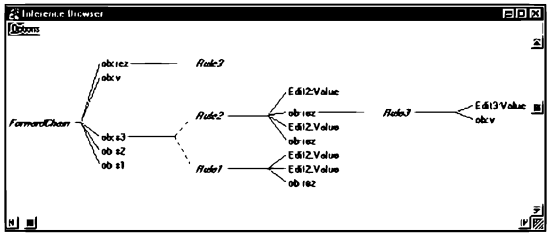


Рис. 8.7. Стратегия оценки лучшего (bestfirst)

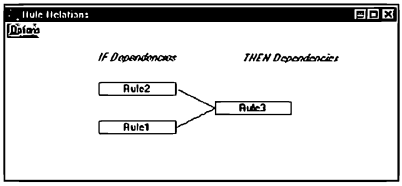


Рис. 8.8. Пример отношения правил

8.4.2. Формирование прямой цепи рассуждений

Рассмотрим более подробно процесс рассуждений, реализованный в ЭО КАРРА, в прямом направлении.

Цикл событийно-управляемых рассуждений начинается, когда элемент вида «объект : слот» переносится из списка слотов, подлежащих рассмотрению (Agenda). Этот элемент представляет собой уже сделанное изменение, последствия которого будут рассмотрены. Кроме того, пара «объект : слот» может быть помещена в список слотов и самим разработчиком.

Текущий элемент проверяется каждым правилом из списка активных правил, сформированного функцией ForwardChain.

Правило рассматривается, если хотя бы одна из предпосылок упоминает текущую пару «объект : слот». Для каждого рассмотренного правила проверяются все сохраняемые предпосылки. Правило, в котором все предпосылки поддаются проверке, является применимым, т.е. готовым к выполнению. Такое правило перемещается из списка и выходит из игры — его операторы-заключения оценены. Если новые данные отвечают первоначальному вопросу (цели), то цепь рассуждений прекращается. В противном случае вход данных побуждает вставку недавно оцененных пар «объект : слот» в список слотов с соответствующими изменениями, последствия которых будут исследоваться в том же порядке.

Если цель не достигнута, то процесс формирования цепи рассуждений продолжается до тех пор, пока список слотов не опустеет.

8.4.3. Активная трассировка при формировании прямой цепи рассуждений

Процесс формирования прямой цепи рассуждений происходит в режиме активной трассировки (Active Trace Mode), который можно наблюдать по состоянию списка слотов (Agenda) и списка правил (Rule List) или же пройти в пошаговом режиме.

Окно трассировки правил (Rule Trace) содержит следующие объекты:

- окно текста трассировки (Trace Text);
- список слотов (Agenda List Box), в котором перечислены все необходимые в процессе рассуждения пары «объект : слот»);
- список трассируемых правил (Rule List Box);
- кнопку выполнения очередного шага (Step Button).

По умолчанию режим активной трассировки выключен, и пользователю доступно только окно Trace Text. В режиме активной трассировки (пункт меню Options окна Rule Trace) становятся видимыми остальные дочерние окна. Кнопка Step Button появляется в режиме пошаговой активной трассировки. Пошаговый режим автоматически вызывает функцию ForwardChain, диалоговая панель запроса аргументов которой показана на рис. 8.9.

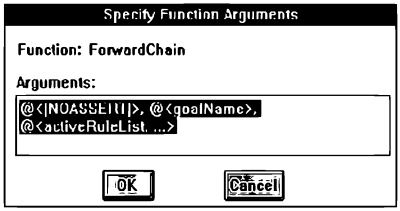


Рис. 8.9. Установка аргументов функции ForwardChain

Размеры всех диалоговых панелей регулируются с помощью окна Resize Trace Window Contents, также открываемого из меню Options (рис. 8.10).

Для дальнейшей работы необходимо в окне настройки трассировки (рис. 8.11) выбрать из списка доступных правил в левой части окна (Available Rules) те, трассировку которых требуется проследить, и поместить их в список в правой части окна (Selected Rules).

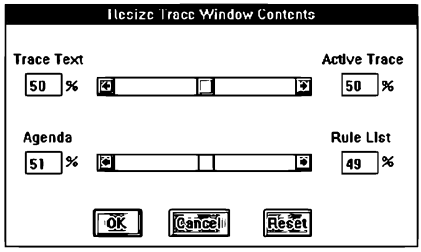


Рис. 8.10. Регулировка размера диалоговых панелей

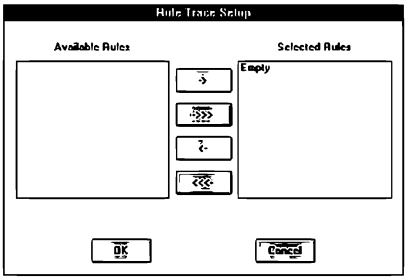


Рис. 8.11. Выбор правил в окне трассировки

Далее аналогичным образом необходимо выбрать требуемые слоты в меню Trace/Break, Slot Trace и поместить их список слотов (Agenda) либо выполнить это с помощью функции Assert. Затем, выбрав пункт меню Options/Step Mode, необходимо указать аргументы функции ForwardChain (рис. 8.12).

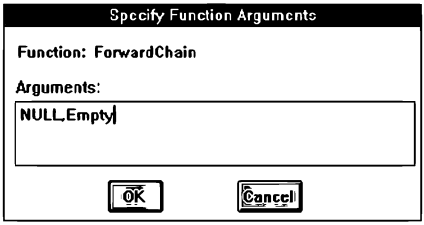


Рис. 8.12. Ввод аргументов функции ForwardChain

После ввода аргументов окно Rule Trace примет вид, показанный на рис. 8.13.

При нажатии кнопки STEP (рис. 8.14) программа переходит к очередному элементу списка слотов и добавляет в список правила, имеющие отношение к этому элементу («объект : слот»).

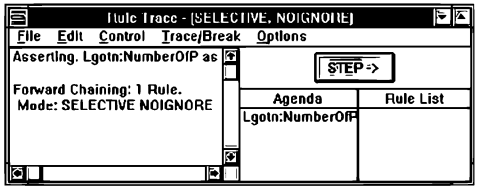


Рис. 8.13. Вид окна трассировки правил в начале работы

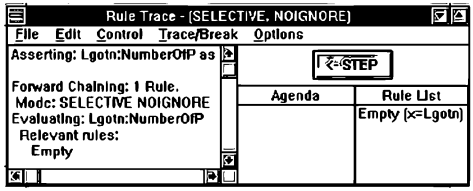


Рис. 8.14. Список активных правил

Стрелка на кнопке STEP показывает направление следующего шага. При ее нажатии программа переходит к очередному правилу в списке и добавляет в список слотов элементы, связанные с этим правилом. При завершении формирования цепочки на экран выдается соответствующее сообщение с результатом формирования цепи рассуждений (рис. 8.15).



Рис. 8.15. Завершение процесса

8.5. Рассуждения в обратном направлении

В рассуждении, управляемом целями, механизм вывода пытается проверить факт достижения цели путем нахождения правил, которые могут его доказать, и дальнейшей попыткой проверки предпосылок этих правил. Предпосылки в свою очередь становятся новыми фактами, которые будут проверяться другими правилами, и т.д.

Для такого формирования цепочки рассуждений требуется предопределенная цель (написанная в Goal-редакторе).

Различают три возможные фазы в обратном рассуждении:

1) расширение (Expanding), при котором происходит попытка оценки условных частей правил и значений пар «объект : слот» в направлении достижения цели;

2) разрушение (Collapsing) — определение достижения цели. Цель может быть достигнута благодаря недавно добавленным с помощью процесса расширения новым фактам. Пары «объек т: слот» и правила могут быть «разрушены» ради достижения цели (становятся лишними);

3) выяснение (Asking), при котором происходит запрос у пользователя значения слота при обнаружении отсутствия последнего. Эта фаза является не обязательной. Для отказа от нее необходимо использовать опцию NOASK при вызове функции.

Вызвать процесс формирования цепочки в обратном направлении можно четырьмя способами:

1) с помощью функции BackwardChain (см. подраздел 7.1.6);

2) из окна трассировки правил путем выбора пункта меню Options/Backward Chain;

3) из окна Inference Browser путем выбора пункта меню Options/Step Mode);

4) программно, с помощью интерпретатора KAL-выражений.

Для более легкого определения состояния текущего процесса при формировании обратного рассуждения в ЭО КАРРА используется система раскраски, объясняемая в табл. 8.1.

Таблица 8.1

Схема раскраски при обратном формировании цепочки

Цвета	Правила	Слоты
Если слоты и правила имеют отношение		
Черный	Имеют отношение и должны быть расширены	
Синий	Расширены, но все еще есть недостаток информации	Были расширены, но пока не проведено выяснение
Как только прошел процесс (выяснение или расширение), сразу правила и слоты становятся красного, серого или зеленого цвета		
Красный	Ложно (FALSE)	Не определен
Серый/оранжевый	—	Определен и не участвовал в работе
Зеленый	Истинно (TRUE)	Определен и участвовал в работе

Окончание

Цвета	Правила	Слоты
Только слоты помещаются в рамку		
Помещаются в рамку	—	Слоты собираются расширяться, разрушаться или «выясняться»

Контрольные вопросы

1. Какой инструментарий ЭО KAPPA предназначен для демонстрации стратегий принятия решений?
2. Как осуществляется взаимодействие правил в цепочках логических рассуждений и стратегиях принятия решений при организации логического вывода в ЭО KAPPA?
3. Назовите отличия прямой и обратной цепочек логических рассуждений.
4. Какой инструментарий ЭО KAPPA предназначен для демонстрации логических рассуждений?
5. Назовите используемые в ЭО KAPPA стратегии принятия решений.

9. СОЗДАНИЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ В ЭО КАРРА

В разделе 6 уже были подробно рассмотрены окно сеанса, его составляющие, а также формы, реализующие взаимодействие пользователя с базой знаний. Напомним лишь, что подменю Image содержит список стандартных компонентов: Button, Text, Edit, Transcript, Bitmap, Drawing, State Box, Meter, Line Plot, Slider, Single List Box, Multiple List Box, Check Box Group, Radio Button Group, Check Box и Select Image. Тому, как эти компоненты используются в процессе разработки интерфейса пользователя KAPPA-программы, и посвящен этот раздел пособия.

Все объекты можно подразделить на три категории:

- отображающие статическую информацию;
- отображающие и позволяющие модифицировать информацию из слотов, принимающих единственное значение;
- отображающие и позволяющие модифицировать информацию из многозначных слотов.

Причем все из них отвечают следующим основным принципам:

- каждый создаваемый объект является экземпляром компонента соответствующего класса;
- каждый объект имеет несколько стандартных, свойственных только ему слотов и методов, определяемых классом компонента, экземпляром которого он является;
- любому объекту можно добавлять новые методы и слоты.

Помимо подменю Image, для выбора класса нового объекта можно использовать инструментальную панель, которая отображается в окне сеанса (Session) при переводе его в режим конструктора (Layout). Кроме этого, объекты можно создавать и редактировать в окне просмотра объектов (Object Browser) или программно с помощью функций MakeImage и MakeInstance интерпретатора языка KAL.

Для создания нового объекта необходимо перейти в режим конструктора, выбрать соответствующий компонент и щелчком мыши указать его место в окне сеанса. Для редактирования объекта также необходимо перейти в режим конструктора и выбрать требуемый объект, щелкнув на нем кнопкой мыши. При двойном щелчке левой кнопкой мыши открывается панель редактирования свойств выбранного объекта.

9.1. Стандартные компоненты интерфейса пользователя

9.1.1. Компонент Button

Объекты данного класса представляют собой кнопку, при нажатии которой выполняются те или иные действия, определенные пользователем. На рисунке 9.1 показана панель свойств объекта типа Button, имеющих следующее назначение:

- поле Title (заголовок) определяет название кнопки, которое будет отображаться на ней;
- поле Action (действие) содержит имя функции, которая будет активироваться при нажатии на кнопку;
- поле RightAction аналогично полю Action, но содержит имя функции, активирующейся при нажатии правой кнопки мыши;
- если на кнопку требуется поместить рисунок, то в поле FileName необходимо указать полный путь к файлу, содержащему этот рисунок;
- поле FuncKey определяет клавишу быстрого, или так называемого «горячего», вызова (функциональные клавиши F1 — F12), нажатие которой в процессе выполнения KAPPA-программы будет равнозначно нажатию на рассматриваемую кнопку;
- флажок ShowBorder определяет режим отображения рамки вокруг кнопки.

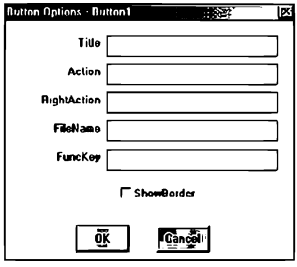


Рис. 9.1. Панель свойств объекта класса Button

9.1.2. Компонент Text

Объекты данного типа позволяют отображать статический текст. Если необходимо вывести заранее подготовленный текст большого объема, можно воспользоваться слотом Filename (имя файла) объ-

екта Text, указав в нем полный путь к требуемому текстовому файлу. Размер поля текста и его положение на экране можно изменять стандартными средствами Windows. Панель свойств объектов класса Text (рис. 9.2):

- поле Title содержит текст, который необходимо отобразить;
- поле RightAction привязывает к объекту действие, совершаемое при нажатии на нем правой кнопки мыши;
- флажок Transparent определяет прозрачность подложки текста при его отображении в окне.
- флажок ShowBorder определяет режим отображения рамки вокруг текста;
- кнопки Fonts и Colors вызывают стандартные диалоги для выбора гарнитуры и кегля шрифта, а также цветов текста и фона;
- группа кнопок с зависимой фиксацией Justification задает выравнивание текста в поле: по левому краю (Left), по центру (Center) или по правому краю (Right).

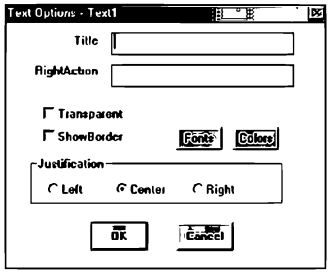


Рис. 9.2. Панель свойств объекта класса Text

9.1.3. Компонент Transcript

Этот компонент практически аналогичен предыдущему за исключением того, что содержимое его текстового поля можно изменять программно в процессе выполнения приложения (рис. 9.3).

Помимо уже описанных настроек, данный компонент позволяет добавлять к полю полосы вертикальной и горизонтальной прокрутки (Scroll Bars), что весьма полезно при большом объеме выводимой информации:

- On — включает соответствующую линейку;
- Off — выключает линейку;
- Auto — режим автоматического включения линейки при выходе текста за границы окна.

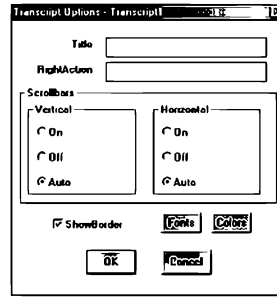


Рис. 9.3. Панель свойств объекта класса Transcript

Для вывода данных в поле предназначены специальные функции:

- ClearTranscriptImage (<имя объекта>) — очищает содержимое поля;
- DisplayText (<имя объекта>, <строка>) — выводит текст;
- DisplayFile (<имя объекта>, <имя файла>) — выводит в поле содержимое текстового файла.

9.1.4. Компонент Edit

Объекты этого класса обеспечивают пользователю возможность просмотра и изменения значения однозначного текстового слота (типа Single) в процессе работы KAPPA-программы. Для редактирования доступны следующие свойства объекта (рис. 9.4):

- поле Length определяет длину вводимой или выводимой строки в символах;
- поле Owner (владелец) задает имя класса или объекта, которому принадлежит редактируемый слот;
- в поле OwnerSlot (слот-владелец) приведено имя слота, значение которого будет редактироваться посредством объекта.

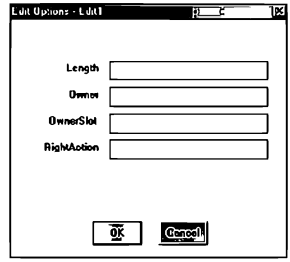


Рис. 9.4. Панель свойств объекта класса Edit

9.1.5. Компонент BitMap

Объекты класса BitMap позволяют импортировать в приложение растровые изображения из файлов формата *.bmp.

В поле FileName указывается полный путь к файлу, в котором хранится требуемая картинка (рис. 9.5).

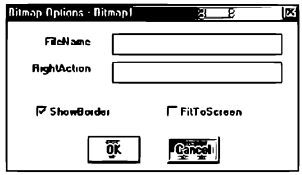


Рис. 9.5. Панель свойств объекта класса BitMap

Флажок FitToScreen позволяет задавать режим отображения. При его установке рисунок равномерно без учета пропорций заполняет отведенную область. При снятом флажке рисунок выводится без учета формы и размера поля вывода изображения.

9.1.6. Компонент Drawing

Объект класса Drawing позволяет пользователю программным способом создавать необходимую графическую картинку (рис. 9.6).

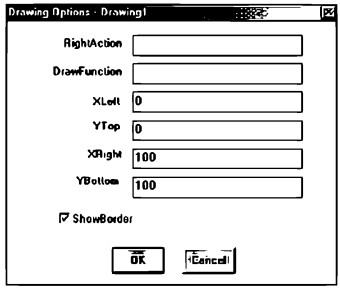


Рис. 9.6. Панель свойств объекта класса Drawing

Область рисования имеет прямоугольную форму и определяется координатами левой верхней (XLeft, YTop) и правой нижней (XRight, YBottom) точек в базисе окна сеанса. Для реализации процедуры рисования необходимо описать специальную KAL-функцию, являющуюся комбинацией стандартных функций рисования MoveTo, LineTo, MoveToRel, LineToRel и Ellipse.

Аргументами функций MoveTo (*x*, *y*) и LineTo (*x*, *y*) являются абсолютные координаты точки в базисе области рисования, в которую необходимо переместить курсор из текущего положения. Разница между этими функциями заключается в том, что первая из них лишь перемещает «кисть», тогда как вторая прорисовывает прямую линию, соединяющую исходную и конечную точки.

Функции MoveToRel (*dx, dy*) и LineToRel (*dx, dy*) аналогичны двум предыдущим. Отличие состоит в способе задания координат, значения *dx* и *dy* которых определяются относительно текущего положения «кисти», т.е. в ее базисе.

Функция Ellipse (xRadius, yRadius) позволяет нарисовать эллипс с полуосями xRadius, yRadius и центром, находящимся в точке текущего положения «кисти».

Идентификатор созданной KAL-функции необходимо указать в поле DrawFunction конкретного объекта Drawing Image.

9.1.7. Компонент StateBox

Объекты типа StateBox позволяют в процессе выполнения приложения демонстрировать значение указанного однозначного слота (рис. 9.7). Изменение значения слота в процессе выполнения программы отражается на состоянии объекта StateBox.

Свойства Owner и OwnerSlot описывают конкретный объект и слот, состояние которого необходимо отразить.

В поле Allowable Values необходимо перечислить все возможные значения, которые может принимать указанный слот.

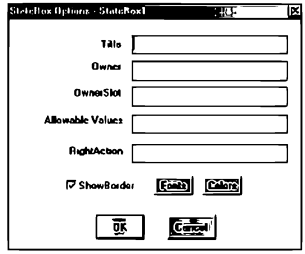


Рис. 9.7. Панель свойств объекта класса StateBox

9.1.8. Компонент Meter

Объекты этого класса позволяют просматривать изменение значения числового слота в процессе работы приложения (рис. 9.8). В данный момент значение слота установлено равным 55.

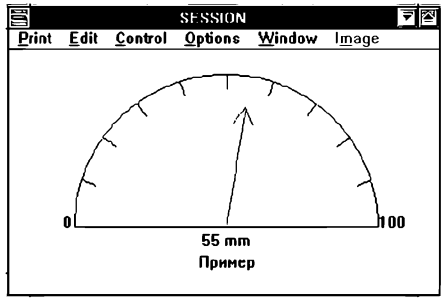


Рис. 9.8. Объект Meter

Пользователю для редактирования доступны следующие свойства объекта (рис. 9.9):

- поля MinValue и MaxValue задают минимальное и максимальное значение для отображения, т.е. границы шкалы. Следует отметить, что если для соответствующего числового слота уже заданы минимальное и максимальное значения, то они будут автоматически унаследованы объектом Meter;
- поле ValueFormat определяет формат выводимых значений (общее количество цифр и количество цифр после запятой);
- поле UnitsTitle задает единицы измерения.

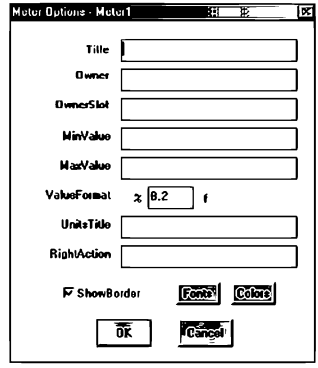


Рис. 9.9. Панель свойств объекта класса Meter

9.1.9. Компонент LinePlot

Объект типа LinePlot позволяет выводить в заданную область до шести графиков, отображающих значения многозначных слотов (рис. 9.10).

Поля панели свойств имеют следующее назначение:

- поле NumLines задает номер графика;
- в полях XTitle и YTitle указываются заголовки осей;
- поля XMin, XMax, YMin, YMax определяют границы шкал осей;
- установка флажка AutoScale включает режим автоматического определения масштаба шкал графика в зависимости от значений слотов;
- флажок LegendOn управляет отображением легенды графика.

Выводимые графики в свою очередь также являются объектами, относящимися к типу LinePlot Line (рис. 9.11).

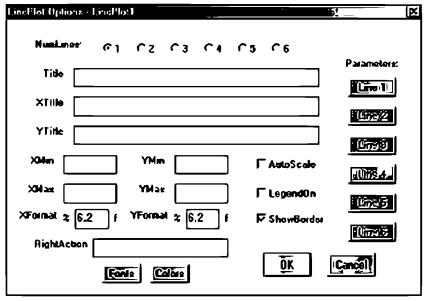


Рис. 9.10. Панель свойств объекта класса LinePlot

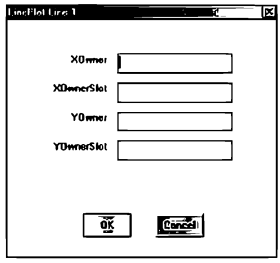


Рис. 9.11. Панель свойств объекта класса LinePlot Line

Для их редактирования необходимо вызвать панель свойств объекта класса LinePlot Line путем нажатия соответствующей кнопки:

Line 1, ... , Line 6. Для каждого графика должны быть заданы поля XOwner, XOwnerSlot, YOwner, YOwnerSlot, определяющие два многозначных слота. Указанные слоты являются источниками данных (координатами), по которым строится график.

9.1.10. Компонент Slider

Объект класса Slider (ползунок) позволяет пользователю в процессе работы KAL-программы изменять значение однозначного числового слота (рис. 9.12) в диапазоне MinValue — MaxValue с шагом, заданным в поле Interval.

При установке флажка Vertical ползунок будет иметь вертикальное положение, иначе — горизонтальное.

9.1.11. Компонент SingleListBox

Объекты этого типа используются для выбора одного значения из предложенного списка возможных значений, заданных в поле Allowable Values (рис. 9.13).

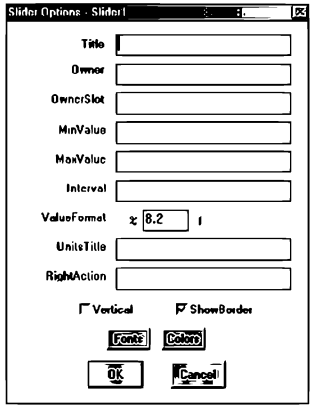


Рис. 9.12. Панель свойств объекта класса Slider

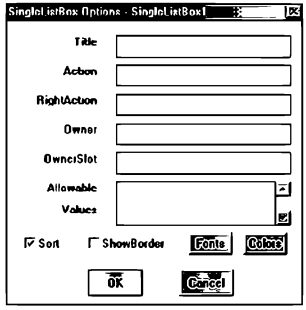


Рис. 9.13. Панель свойств объекта класса SingleListBox

Также допустимые значения могут быть представлены в описании слота-владельца, однако в этом случае строка Allowable Values должна быть пустой. Слот должен быть однозначным, числовым или текстовым.

При установке флажка Sort перечисленные значения сортируются в алфавитном порядке.

9.1.12. Компонент MultipleListBox

Этот компонент практически аналогичен предыдущему, однако позволяет осуществить выбор сразу нескольких значений из списка, или так называемый множественный выбор (рис. 9.14). Поставленный в соответствие многозначный слот должен быть числовым или текстовым.

9.1.13. Компонент CheckBox

Объект класса CheckBox представляет собой кнопку с независимой фиксацией (кнопку-флажок) и позволяет изменять значение однозначного булевого слота (рис. 9.15).

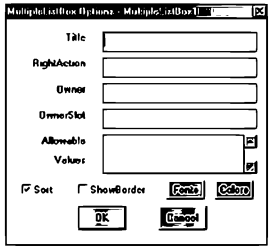


Рис. 9.14. Панель свойств объекта класса MultipleListBox

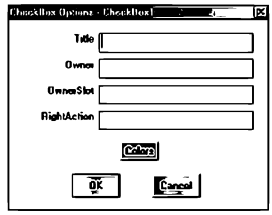


Рис. 9.15. Панель свойств объекта класса CheckBox

Примечание. Объекты данного класса доступны только из меню Image окна сеанса. При установке флажка значение слота равно true, а при его снятии (поле флажка пустое) — false. Если объект CheckBox отображается в сером цвете, то значение его слота не определено (NULL). Следует также отметить, что окно Session позволяет установить только значения true или false. Значение NULL можно установить лишь при помощи KAL-функции ResetValue.

9.1.14. Компонент CheckBoxGroup

Позволяет установить и просмотреть значения многозначного слота (рис. 9.16). Для отображения нового значения в Image-объекте нужно воспользоваться функцией ResetImage. Допустимые типы слота — текст, объект.

9.1.15. Компонент RadioButtonGroup

Объект класса RadioButtonGroup представляет собой группу кнопок с зависимой фиксацией, т.е. в конкретный момент в нажатом состоянии может находиться только одна кнопка из группы (рис. 9.17).

Он позволяет изменять значение однозначного слота, выбирая его из списка допустимых значений Allowable Values. Однако приоритет допустимых значений, определенных в соответствующем слоте, всегда будет выше, чем у допустимых значений, заданных в поле Allowable Values.

Примечание. Изменение значения посредством объекта мгновенно влечет за собой изменение в ассоциированном с ним слоте. Однако в обратном направлении это правило не действует. Если значение переменной изменено программно (например, с помощью функции SetValue), то сам объект изменится только после применения к нему функции ResetImage.

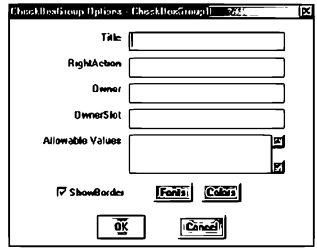


Рис. 9.16 Панель свойств объекта класса CheckBoxGroup

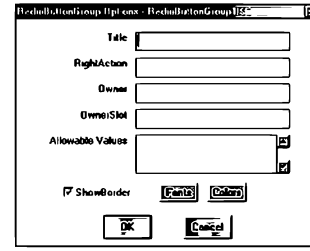


Рис. 9.17. Панель свойств объекта класса RadioButtonGroup

9.2. Особенности русификации ЭО КАРРА

В ЭО КАРРА нельзя использовать русскоязычные символы как в написании имен переменных или объектов, так и в качестве значений переменных. Тем не менее при оформлении пользовательского интерфейса в окне сеанса воспользоваться символами русского алфавита все-таки можно, причем при переносе КАРРА-продукта в формате *.bin настройки сохранятся.

Поскольку файл в двоичном формате, как правило, весьма громоздок, то в процессе отладки КАРРА-программы рекомендуется использовать только латинские символы. После завершения разработки последнюю версию KAL-файла необходимо транслировать в BIN-файл с добавлением русскоязычного интерфейса.

Контрольные вопросы

1. Перечислите компоненты ЭО КАРРА.
2. Назовите инструменты проектирования экспертных систем в ЭО КАРРА.
3. Объясните понятия «фрейм», «объект» и «слот».
4. Опишите необходимость использования методов при обработке значений слотов фрейма. Если необходимо, приведите пример.
5. Представление таксономии систем в ЭО КАРРА.
6. Каков принцип организации логического вывода в ЭО КАРРА?
7. Какие цепочки организации логического вывода вы знаете и как они реализуются в ЭО КАРРА?
8. Определите стратегии принятия решений в экспертных системах и в ЭО КАРРА.

9. Что включает агенда?
10. Опишите организацию работы с правилами в ЭО КАРРА.
11. Определите последовательность проектирования экспертных систем в ЭО КАРРА.
12. Как представляются декларативные и процедурные знания в ЭО КАРРА?
13. Опишите инструментарий создания интерфейса пользователя в приложении, разрабатываемом в ЭО КАРРА.
14. Чем отличаются файлы приложения, представленные с расширениями *.kal и *.bin?
15. Чем отличаются приложения, разработанные только на основе функций, и приложения, разработанные на основе правил?

10. ИНСТРУМЕНТАЛЬНАЯ ОБОЛОЧКА РАЗРАБОТКИ ЭС — CLIPS

10.1. Общие сведения об ИО CLIPS

Первая версия инструментальной оболочки CLIPS была разработана в 1985 г. Особое внимание было уделено созданию инструмента, совместимого с известными системами искусственного интеллекта. Таким образом, синтаксис CLIPS очень близок к синтаксису подмножества Borland C. CLIPS (C Language Integrated Production System) и является одним из распространенных инструментальных средств разработки экспертных систем. В оболочке реализуется продукционная модель знаний. В основе разработки три компонента: список фактов, база знаний в виде правил, блок логического вывода. Система полностью реализована на языке Borland C. ИО CLIPS поддерживает как процедурную, так и объектно-ориентированную парадигму программирования. Благодаря своей мобильности, расширяемости и большим возможностям ИО CLIPS находит широкое применение в правительстве, промышленности и университетах.

Желающим доступна как сама инструментальная оболочка CLIPS последней версии, так и ее исходный код, размещенные на официальном сайте http://www.ghg.net/clips/CLIPS. html

CLIPS состоит из интерактивной среды — экспертной оболочки со своим способом представления знаний, гибкого языка и нескольких вспомогательных инструментов.

10.2. Программирование в ИО CLIPS

10.2.1. Основные элементы программирования

CLIPS предоставляет три основных элемента для написания программ:

- простые типы данных;
- функции для манипулирования данными;
- конструкции для пополнения базы знаний.

Простые типы данных. Для представления информации в CLIPS предусмотрены следующие простые типы данных: float, integer, symbol, string, external-address, instance-name и instance-address.

Для представления числовой информации используются типы float и integer, символьной — symbol и string.

При записи числа могут использоваться только цифры 0—9, десятичная точка «.», знак «+» или « — » и символ «e» при представлении в экспоненциальном формате. Число сохраняется либо как целое, либо как действительное. Число, состоящее только из цифр и знака, сохраняется как целое.

Количество значащих цифр зависит от аппаратной реализации. В этой же связи могут возникать ошибки округления. Как в любом языке программирования, особенную осторожность необходимо проявлять при сравнении чисел с плавающей запятой как между собой, так и при сравнении с ними целых чисел.

Последовательность символов, которая не удовлетворяет требованиям к числовым типам, обрабатывается как тип данных symbol.

Тип данных symbol в CLIPS — это последовательность символов, состоящая из одного или нескольких любых печатных символов кода ASCII. Как только в последовательности символов встречается символ-разделитель, symbol заканчивается.

Разделителями могут служить следующие символы: любой непечатный ASCII-символ (включая пробел, символ табуляции, CR, LF), двойные кавычки, а также « () & | < ~;».

Символы-разделители не могут включаться в symbol за исключением символа «<», который может быть первым символом. Кроме этого, symbol не может начинаться с символа «?» или последовательности символов «$'?», поскольку эти символы зарезервированы для переменных. Заметим, что CLIPS различает регистр символов.

Переменные типа string представляют собой последовательность символов длиной от нуля и более печатных символов, заключенную в двойные кавычки. Если внутри строки встречаются двойные кавычки, то перед ними необходимо поместить символ «\». То же справедливо и для самого символа «\». Следует отметить, что, например, строка «abed» не эквивалентна строке abed. Они содержат одинаковые наборы символов, но являются экземплярами различного типа.

Пример.

Целое число:		237	15	+12	−32
Вещественное число:		237e3	15.09	+12.0 -	32.3e-7
Символ:	Hello	B76-HI	bad_value	742-42-42	@+=-%
Строка:	"foo"	"a and b"	"I number"	"a\"quote"	

Функции. Под функцией в CLIPS понимается фрагмент исполняемого кода, с которым связано уникальное имя, возвращающий полезное значение или имеющий полезный побочный эффект (например, вывод информации на экран).

Существует несколько типов функций. *Пользовательские* и *системные функции* — это фрагменты кода, написанные на внешних языках (например, на C++) и связанные со средой CLIPS. Системными называются те функции, которые были определены изначально внутри среды CLIPS.

Хотя ИО CLIPS и не ориентирована на вычислительные операции, в ней предусмотрен ряд стандартных арифметических и математических функций:

- + — сложение;
- – — вычитание;
- * — умножение;
- / — деление;
- ** — возведение в степень;
- Abs — определение абсолютного значения;
- Sqrt — вычисление квадратного корня;
- Mod — взятие по модулю;
- Min — нахождение минимума;
- Max — нахождение максимума.

Вызовы функций в CLIPS имеют префиксную форму, т.е. аргументы функции могут стоять только после ее названия. Вызов функции начинается с открывающейся скобки, за которой следует имя функции, затем идут аргументы, каждый из которых отделен одним или несколькими пробелами. Аргументами функции могут быть данные простых типов, переменные или вызовы других функций. В конце вызова ставится закрывающаяся скобка.

Пример.

```
(+ 3 4 5)
(* 5 6 0 2)
(+ 3 (* 8 9) 4)
(* 8 (+ 3 (* 2 3 4) 9) (* 3 4))
(** 3.5 6)
```

Конструкции. В CLIPS существует несколько видов описывающих конструкций: defmodule, defrule, deffacts, deftemplate, defglobal, deffunction, defclass, definstances, defmessage-handler и defgeneric.

При записи все они заключаются в скобки. Определение конструкции отличается от вызова функции главным образом по производимо-

му эффекту. Обычно вызов функции оставляет состояние среды CLIPS без изменений (за исключением случаев, когда речь идет о функциях сброса, очистки, открытия файла и т.п.). Определение конструкции, напротив, в точности направлено на изменение состояния среды путем внесения изменений в базу знаний CLIPS. В отличие от функций конструкции никогда не возвращают значений.

Конструкция deffunction позволяет пользователю определять новые функции непосредственно в среде CLIPS. Функции, определенные таким образом, выглядят и работают подобно остальным функциям, однако выполняются не напрямую, а интерпретируются средой CLIPS.

Все конструкции (за исключением defglobal) позволяют размещать комментарии сразу вслед за именем конструкции. Кроме того, комментарии могут вставляться в код CLIPS с помощью точки с запятой «;». Последовательность символов в строке, следующая за символом «;» до конца строки, игнорируется. Если «;» стоит первым символом в строке, то комментарием считается вся строка целиком.

10.2.2. Факты

Факты являются одной из основных форм представления информации в системе CLIPS. Каждый факт представляет собой фрагмент информации, помещенный в текущий список фактов, так называемый fact-list. Факт представляет собой основную единицу данных, используемую правилами.

Количество фактов в списке и объем информации, который может быть сохранен в факте, ограничивается только размером памяти компьютера. При попытке добавления к списку нового факта, идентичного одному из фактов, уже включенных в список, операция игнорируется (хотя такое поведение можно изменить).

Факт может описываться индексом или адресом. При каждом добавлении или изменении факта ему присваивается уникальный целочисленный *индекс*. Индексы фактов начинаются с нуля и для каждого нового или измененного факта увеличиваются на единицу. Каждый раз после выполнения команд reset и clear выделение индексов начинается с нуля.

Факт также может задаваться с помощью *адреса*. Адрес факта может быть получен путем сохранения возвращаемого значения команд, которые возвращают в качестве результата адрес факта (таких как assert, modify и duplicate), или путем связывания переменной с адресом факта в левой части правила (см. далее).

Идентификатор факта — это короткая запись для отображения факта на экране, состоящая из символа f и записанного через тире индекса факта. Например, запись f–10 служит для обозначения факта с индексом 10. Существует два формата представления фактов: позиционный и непозиционный.

Позиционные факты. Позиционные факты состоят из выражения символьного типа, за которым следует последовательность (возможно, пустая) из полей, разделенных пробелами. Вся запись заключается в скобки. Обычно первое поле определяет «отношение», которое применяется к оставшимся полям.

Пример.
(The pump is on)
(Altitude is 10000 feet)
(Grocery_list bread milk eggs)

Поля в позиционных фактах могут быть любого простого типа (за исключением первого поля, которое всегда должно быть типа symbol). На порядок следования полей также не накладывается никаких ограничений.

Ряд символьных выражений зарезервирован, и они не должны использоваться в качестве первого поля любых видов фактов, как позиционных, так и непозиционных: test, and, or, not, declare, logical, object, exists u forall.

Непозиционные факты. Для того чтобы обратиться к информации, содержащейся в позиционном факте, пользователь должен знать не только о том, какие данные содержатся в факте, но и то, в каком поле они хранятся. Непозиционные (шаблонные) факты дают возможность пользователю абстрагироваться от структуры факта, задавая имена каждому из полей факта. Для задания шаблона, который затем может использоваться при доступе к полям по именам, используется конструкция deftemplate. Она подобна структуре или записи в языках программирования Borland C и Pascal.

Конструкция deftemplate позволяет наряду с определением именованных полей, или слотов, вводить имя шаблона. В отличие от позиционных фактов слоты шаблонного факта могут быть ограничены по типу, значению, числовому диапазону. Кроме того, для любого слота можно определить значения по умолчанию. Слот состоит из открывающейся скобки, за которой следует имя слота, полей (могут отсутствовать) и закрывающейся скобки. Заметим, что слоты не могут использоваться в позиционных фактах, также как и позиционные

поля не могут использоваться в шаблонных фактах. Общая структура конструкции deftemplate имеет вид:

(deftemplate <имя шаблона>) (slot-1) (slot-2) ... (slot-N)

Пример. Шаблон со значениями слотов, заданными по умолчанию:

```
(deftemplate prospect)
(slot name
(default? DERIVE)
     (slot assets
(default rich)
     (slot age
(default 80)))
```

Шаблонные факты отличаются от позиционных первым полем в факте. Первое поле всех фактов должно быть типа symbol, но если это символьное выражение соответствует имени шаблона, то этот факт — шаблонный. За первым полем шаблонного факта следует список из нуля или более слотов. Как и позиционные, шаблонные факты заключаются в скобки.

Пример. Шаблонные факты:

```
(client (name "Joe Brown") (id X9345A))
(point-mass (x-velocity 100) (y-velocity -200))
(class (teacher "Martha Jones") (#-students 30)
(room "37A"))
(grocery-list (#-of-items 3) (items bread milk eggs))
```

Заметим, что порядок следования слотов в шаблонном факте неважен.

Манипуляции над фактами. Факты могут добавляться к списку фактов (с помощью команды assert), удаляться из него (с помощью команды retract), изменяться (с помощью команды modify) и дублироваться (с помощью команды duplicate) самим пользователем или программой, например (assert (light green)).

Кроме того, конструкция deffacts позволяет определить множество исходных, или априорных, знаний в виде набора фактов.

Пример.

```
(deffacts walk "Some facts about, walking"
     (status walking)
     (walk-sign walk))
```

Когда осуществляется сброс состояния среды CLIPS (с помощью команды reset), все факты, описанные в конструкции deffacts, добавляются к списку фактов. Кроме того, по этой команде в список фактов заносится исходный факт (initial-fact). Этот факт включается в список фактов всегда с идентификатором f–0. Его назначение будет рассмотрено далее.

10.2.3. Правила

Одним из основных методов представления знаний в CLIPS являются правила. Правила используются для представления эвристик, определяющих ряд действий, которые необходимо выполнить в определенной ситуации. Разработчик экспертной системы определяет совокупность правил, которые используются совместно для решения проблемы.

Правило состоит из двух частей: *антицедента* (условия), который является аналогом условия в if-then операторе и записывается слева, и *консеквента* (заключения), который является аналогом then-части этого оператора и записывается справа.

Левая часть правила представляет собой ряд условий (условных элементов), которые должны выполняться, чтобы правило было применимо. В CLIPS принято считать, что условие выполняется, если соответствующий ему факт присутствует в списке фактов. Одним из типов условных элементов может быть образец. Образцы состоят из набора ограничений, которые используются для описания того, какие факты удовлетворяют условию, определяемому образцом. Процесс сопоставления фактов и образцов выполняется блоком вывода CLIPS, который автоматически сопоставляет образцы исходя из текущего состояния списка фактов и определяет, какие из правил являются «применимыми». Если все условия правила выполняются, то оно активируется и помещается в список активных правил.

Если левая часть правила пуста, то для его активации необходимо наличие в списке фактов исходного факта (initial fact). Такие безусловные правила часто используются для того, чтобы инициировать работу программы. Поэтому перед запуском таких программ необходимо произвести сброс состояния среды CLIPS.

Правая часть правила представляет собой совокупность действий, которые должны быть выполнены, если правило применимо. Действия, описанные в применимых правилах, выполняются тогда, когда блок вывода CLIPS получает команду начать выполнение применимых правил. Если существует множество применимых правил, то для того чтобы выбрать правило, действия которого должны быть выполнены, блок вывода использует стратегию разрешения конфликтов.

Действия, описанные в выбранном правиле, выполняются (при этом список применимых правил может измениться), а затем блок вывода выбирает другое правило и т.д. Этот процесс продолжается до тех пор, пока не остается ни одного применимого правила,

т.е. пока список активированных правил не окажется пуст. Во многом правила похожи на операторы типа if-then процедурных языков программирования. Однако условие оператора if-then в процедурном языке проверяется только тогда, когда программа передает ему управление. С правилами ситуация иная. Блок вывода постоянно отслеживает все правила, условия которых выполняются, и таким образом, правило может быть выполнено в любой момент, как только оно становится применимым. В этом смысле правила подобны обработчикам исключительных ситуаций в процедурных языках, например в языке Ада.

Для определения правил используется конструкция defrule:

(defrule <имя правила> <комментарий>
(<шаблон_1>) (<шаблон_2>) ... (<шаблон_n>)
(<действие_1>) (<действие_2>) ... (<действие_m>))

Пример.

```
(defrule take-a-vacation
(work clone)
(money plenty)
(reservations made)
= >
(printout t "Let's go!" crlf))
```

10.2.4. Переменные

Как и в других языках программирования, в CLIPS для хранения значений используются переменные. В отличие от фактов, которые являются статическими, или неизменными, содержание переменной динамично и изменяется по мере того, как изменяется присвоенное ей значение.

Идентификатор переменной всегда начинается с вопросительного знака, за которым следует ее имя. В общем случае формат переменной выглядит следующим образом: ? <имя переменной>.

Пример.

```
?x
?sensor
?noun
?color
```

Перед использованием переменной ей необходимо присвоить значение. Все переменные, кроме глобальных, считаются локальными и могут использоваться только в рамках описания конструкции. К локальным переменным можно обращаться внутри описания, но они

не определены вне него. Обычно переменные описывают и присваивают им значения в левой части правила.

Пример.

```
(defrule make-quack
(duck-sound ?sound)
= >
(assert (sound-is ?sound))
```

Получив значение, переменная сохраняет его неизменным при использовании как в левой, так и в правой частях правила, если только это значение не изменяется в правой части с помощью функции bind.

Пример.

```
(defrule addition)
(numbers ?x ?y)
=>
(assert (answer (+ ?x ?y)))
(bind? answer (+ ?x ?y))
(printout t "answer is " ?answer crlf))
```

Кроме значения самого факта, переменной может быть присвоено значение адреса факта. Это может оказаться удобным при необходимости манипулировать фактами непосредственно из правила. Для такого присвоения используется комбинация «<-».

Пример. Присвоение переменой значения адреса факта:

```
(defrule get-married
?duck <- (bachelor Dopey)
=>
(retract? duck))
```

Для определения глобальных переменных, которые видны всюду в среде CLIPS, используется конструкция defglobal. К глобальной переменной можно обратиться в любом месте, и ее значение остается независимым от других конструкций. Глобальные переменные CLIPS подобны глобальным переменным в процедурных языках программирования, но они значительно слабее типизированы (на них не налагаются ограничения хранения данных только одного типа).

10.2.5. Дополнительные средства

CLIPS предоставляет ряд дополнительных средств, необходимых при написании программ. Основными из них являются: ограничения на значения полей; оператор проверки условия test; использование функций в правилах; использование процедурных знаний. Рассмотрим каждое из этих средств.

Ограничения на значения полей. Использование ограничений на значения полей позволяет ограничить значения, принимаемые образцами в левой части правила.

Рассмотрим три вида ограничений: «~», «|» и «&».

Ограничение первого типа действует на следующее прямо за ним значение и говорит о том, что поле не может принимать это значение.

Пример.

```
(defrule walk
(light -green)
=>
(printout t "Don't walk" crlf))
```

Ограничение второго типа указывает на то, что поле может принимать одно из следующих значений.

Пример.

```
(defrule cautious
(light yellow|blinking-yellow)
=>
(printout t "Be cautious" crlf))
```

Ограничение третьего типа используется только вместе с ограничениями первых двух типов и указывает на то, что должны удовлетворяться оба ограничения, соединяемых с его помощью.

Пример.

```
(defrule cautious
(light ?color&yellow!blinking-yellow)
=>
(printout t "Be cautious because light is " ?color
crlf))
```

Оператор проверки условия test. Оператор проверки условия test представляет собой мощное средство, позволяющее сравнивать числа, переменные и строки в левой части правила. Он записывается точно так же, как и образцы. Правило может считается выполненным только тогда, когда наряду с совпадением всех образцов, записанных в левой части правила, справедливо и условие, описанное в test.

Функция test имеет следующий синтаксис:

(test (<аргумент_сравнения> <аргумент_1><аргумент_2>)),

где <аргумент_сравнения> — параметр, по которому сравниваются два следующих аргумента.

В CLIPS существует ряд предопределенных аргументов сравнения:

- eq — равно (сравнивает тип и значение);
- neq — не равно;

- = — равно;
- <> — не равно;
- >= — больше или равно;
- > — больше;
- <= — меньше или равно;
- < — меньше.

Все аргументы, кроме eq и neq, используются только для сравнения чисел. При интерпретации выражения сравнения считается, что <аргумент_1> стоит слева от <аргумента_сравнения>, а <аргумент_2> — справа.

Использование функций в правилах. Функции могут использоваться и в левой, и в правой части правила.

Пример.

```
(defrule addition
(numbers ?x ?y)
(assert (answer (+ ?x ?y))))
```

При использовании функции в левой части правила перед ней должен стоять знак «=», указывающий CLIPS, что следующее выражение необходимо вычислить, а не использовать буквально.

Пример.

```
(defrule addition
(numbers ?x ?y)
(stock ?ID = (sqrt (+ (** ?x 2) (** y 2))))
=>
(printout t "stock ID = " ?ID crlf))
```

Использование процедурных знаний. CLIPS поддерживает также процедурную парадигму представления знаний, подобную принятой в обычных языках программирования (C, Паскаль). Конструкция deffunction позволяет пользователю определять новые функции, которые могут вызываться точно так же, как и встроенные функции CLIPS. Конструкция defmodule позволяет разбивать базу знаний на части.

10.3. Интерфейс ИО CLIPS

ИО CLIPS может работать в следующих режимах:

- интерактивном, с использованием простого текстового интерфейса командной строки;
- интерактивном, с использованием GUI-интерфейса;
- как ЭС, интегрированная в другие приложения.

Рассмотрим перечисленные режимы более подробно.

10.3.1 Интерфейс командной строки

Основным методом взаимодействия пользователя с CLIPS является ввод команд из командной строки CLIPS.

После появления на экране подсказки CLIPS> пользователь может ввести команду.

Командами могут быть вызовы функций, конструкции, глобальные переменные или константы. При вызове функции вычисляется ее значение, которое выводится на экран. Вызов функции в CLIPS имеет префиксную форму, т.е. аргументы функции стоят после ее названия. Если ввести определение, то будет создана конструкция соответствующего типа. В ответ на ввод глобальной переменной на экран будет выведено ее значение.

Пример.

```
CLIPS> (+ 3 4)
7
CLIPS> (defglobal ?*x* = 3)
CLIPS> ?*x*
3
CLIPS> red
red
CLIPS>
```

Примечание. В приведенном примере сначала была вызвана функция сложения с аргументами 3 и 4, которая вернула результат 7. Затем была определена глобальная переменная ?*x*, и ей было присвоено значение 3. Далее переменная ?*x* была введена с командной строки и было возвращено ее значение 3. В конце была введена и сразу возвращена константа red.

Стандартная процедура использования интерфейса командной строки содержит следующие последовательные операции:

1) создание и редактирование базы знаний с помощью любого текстового редактора;

2) сохранение базы знаний в одном или нескольких текстовых файлах;

3) выход из редактора и запуск ИО CLIPS;

4) загрузка базы знаний из текстовых файлов в ИО CLIPS.

Пример.

1. С помощью текстового редактора Notepad создадим текстовый документ следующего содержания:

```
(defrule one
(light green)
=>
```

```
(printout t "Go!!!" crlf))
```

2. Сохраним его в файле под именем TEST. CLP.
3. Запустим CLIPS (на экране появится главное диалоговое окно CLIPS).
4. Для загрузки правила с именем «one» в базу знаний CLIPS воспользуемся стандартной командой: (load "C:/CLIPSWIN/TEST.CLP").

10.3.2. Графический интерфейс пользователя

Как уже было отмечено ранее, CLIPS использует графический интерфейс пользователя, или так называемый GUI-интерфейс.

В таблице 10.1 приведено краткое описание наиболее часто используемых пунктов меню главного окна ИО CLIPS.

Таблица 10.1

Меню	Пункт	Сочетание клавиш	Назначение
File	Load Constructs...	Ctrl-L	Загрузка конструкций CLIPS из текстового файла. Эквивалентен команде load <имя файла>
	Load Batch	—	Загрузка командного файла. Эквивалентен команде batch <имя файла>
	Load Binary Image	—	Загрузка конструкций CLIPS, сохраненных в двоичном виде. Эквивалентен команде bioad <имя файла>
	Turn Dribble On	—	Начало записи протокола работы с CLIPS в текстовый файл. Эквивалентен команде dribble-on <имя файла>. После активации изменяется на Turn Dribble Off
	Turn Dribble Off	—	Завершение записи протокола работы с CLIPS в текстовый файл. Эквивалентен команде dribble-off
	Save Binary	—	Сохранение конструкций CLIPS в двоичном виде. Эквивалентен команде bsave <имя файла>
	Editor	—	Запуск встроенного редактора
	Quit	Ctrl-Q	Выход из CLIPS
Edit	Paste	Ctrl-V	Копировать содержимое буфера обмена в диалоговое окно (вставляемый текст всегда помещается в конце окна)

Продолжение

Меню	Пункт	Сочетание клавиш	Назначение
	Complete...	Ctrl-J	«Завершение» символа, вводимого в данный момент в диалоговом окне. Если не существует никакого возможного продолжения, подается звуковой сигнал. Если существует только одно возможное продолжение, то символ автоматически завершается. Если существует больше чем одно завершение, то список всех возможных продолжений выводится в диалоговом окне. Нажмите кнопку OK, чтобы завершить символ текущим выделением. Нажмите кнопку Cancel, чтобы закрыть диалоговое окно без завершения команды
Execution	Reset	—	Сброс. Эквивалентен команде reset. При сбросе сначала очищаются списки фактов и активированных правил, затем в список фактов заносятся исходный факт (initial-fact) и факты, описанные в конструкциях (deffacts)
	Run	Ctrl-R	Запуск. Эквивалентен команде run. В ходе выполнения программы изменяется на Halt
	Halt	Ctrl-C	Останов
Execution	Step	Ctrl-T	Пошаговое выполнение. Эквивалентен команде run<количество шагов>. Количество шагов задается в поле Step Rule Firing Increment панели настройки параметров CLIPS
	Watch	Ctrl-W	Смена режимов просмотра
	Preferences	—	Настройка параметров CLIPS
	Clear CLIPS	—	Очистка CLIPS. Система приводится к начальному состоянию
Browse	Module	—	Переключение между модулями программы. По умолчанию создается лишь один модуль — MAIN
	Defrule Manager	—	Просмотр и редактирование базы знаний
	Deffacts Manager	—	Просмотр и редактирование списков фактов

Окончание

Меню	Пункт	Сочетание клавиш	Назначение
Browse	Agenda Manager	—	Просмотр и редактирование списка активированных в данный момент правил
Window	Facts Window	—	Просмотр списка фактов
	Agenda Window	—	Просмотр списка активированных в данный момент правил
	Globals Window	—	Просмотр всех глобальных переменных и их значений
	All Above	—	Отображение всех окон из данного меню
	None	—	Закрытие всех окон
	Clear Dialog Window	—	Очистка диалогового окна

10.3.3. Интерфейс встроенного редактора

Редактор CLIPS представляет собой встроенный текстовый редактор, предназначенный для написания программ для CLIPS. В таблице 10.2 приведено краткое описание всех пунктов меню GUI-интерфейса редактора.

Таблица 10.2

Меню	Пункт	Сочетание клавиш	Назначение
File	New	Ctrl-N	Открывает новый файл для редактирования с именем untitled
	Open	Ctrl-O	Открытие текстового файла для редактирования. Открыт может быть только один файл
	Save	Ctrl-S	Сохранение редактируемого файла. Если файл не был назван, диалоговое окно запросит имя файла, чтобы сохранить файл
File	Save As	—	Сохранение редактируемого файла под новым именем
	Page Setup	—	Определение параметров страницы (используется при печати)

Окончание

Меню	Пункт	Сочетание клавиш	Назначение
File	Print	Ctrl-P	Печать активного окна редактирования
	Quit Editor	Ctrl-Q	Выход из редактора
Edit	Undo	Ctrl-Z	Отмена последней операции редактирования
	Cut	Ctrl-X	Удаление выделенного фрагмента в буфер обмена
	Copy	Ctrl-C	Копирование выделенного фрагмента в буфер обмена
	Paste	Ctrl-V	Копирование содержимого буфера обмена в окно редактирования
	Clear	—	Удаление выделенного фрагмента

10.4. Организация работы в ИО CLIPS

Порядок работы в ИО CLIPS целесообразно рассмотреть на примере решения несложной логической задачи.

10.4.1. Постановка задачи и составление программы

Постановка задачи. 1. Дан ребус: GERALD + DONALD = ROBERT.

2. В словах GERALD, DONALD и ROBERT вместо букв необходимо поставить цифры таким образом, чтобы получилось математически правильное выражение. Разным буквам должны соответствовать разные цифры.

3. Требуется написать программу решения ребуса.

Алгоритм решения. 1. Рассмотрим первые разряды всех трех чисел. Считаем, что первый разряд — это единицы, второй разряд — это десятки, третий — сотни и т.д. Очевидно, что нахождение множества пар (буква — цифра), входящих в решение задачи, должно удовлетворять условию:

$$(((D + D) \bmod 10) = T) \ \& \ (D? T). \qquad (10.1)$$

2. Рассмотрим второй разряд совместно с первым. Необходимо подобрать такие числа L и R, чтобы остаток отделения суммы (L*10 + L*10 + D + D) на 100 был равен (R*10 + T). При этом следует учесть, что L и R не равны между собой и не равны D и T.

Таким образом, среди множества пар (буква — цифра), удовлетворяющих первому условию, необходимо найти такие, чтобы:

$$(((L*10 + L*10 + D + D) \bmod 100) = (R*10 + T)) \& (D?\ T?\ L?\ R). \quad (10.2)$$

3. Продолжая действовать подобным образом, можно выписать конечное условие, которому должны удовлетворять пары (буква — цифра), входящие в решение данной задачи.

Решение. Сформулируем описанный ранее подход к решению в терминах CLIPS.

Представим комбинации из одной буквы и одной цифры в виде фактов:

(combination D 0), (combination D 1), (combination A 1) и т.д.

В первую очередь составим правило для нахождения комбинаций (буква — цифра), удовлетворяющих условию (10.1):

```
(defrule find_solution
(combination D ?d)
(combination T ?t&~ ?d)
(test (= (mod (+ ?d ?d) 10) ?t))
=>
(printout t "A solution is:" t t)
(printout t " D = " ?d t)
(printout t " T = " ?t t))
```

Данное правило выполнимо только в том случае, если в списке фактов существуют факты, удовлетворяющие условиям:

1) (combination D ?d) — все факты, в которых в первой и во второй позициях стоят «combination» и «D» соответственно, а в третьей позиции — любое значение, которое присваивается переменной ?d;

2) (combination T ?t&~ ?d) — все факты, в которых в первой и во второй позициях стоят «combination» и «T» соответственно, а в третьей позиции — любое значение, только не то, которое было присвоено переменной ?d, это значение присваивается переменной ?t. Знак «&» означает логическое «и» (and), а знак «~» — логическое «не» (not). Выражение «?t&~ ?d» означает, что переменная ?t может принимать значение, не равное значению переменной ?d;

3) (test (= (mod (+ ?d ?d) 10) ?t)) — это условие проверяет на равенство два выражения (mod (+ ?d ?d) 10) и (?t)).

Команда (printout t "A solution is:" t t) выводит на экран информацию, которая указывается после команды в полях, разделенных пробелами. Чтобы вывести текст, его необходимо заключить в кавычки, а перед кавычками поставить букву t. Эта буква указывает, что в сле-

дующем поле находится текст, взятый в кавычки, который необходимо напечатать. Если после буквы t нет текста в кавычках, то происходит перевод курсора на новую строку.

Перевести курсор на новую строку можно путем указания комбинации символов «crlf» без кавычек. Для вывода значения переменной необходимо просто указать ее имя.

Пример.

```
printout t " T = " ?t t)
```

Оператор выводит текст T= и значение переменной? t, а затем переводит курсор на новую строку.

Рассмотренное правило позволяет найти все комбинации (буква — цифра), удовлетворяющие условию для первых разрядов трех чисел. По аналогии с этим правилом составим конечное правило для нахождения решения:

```
(defrule find-solution
(combination D ?d)
(combination T ?t'&~ ?d)
(test (= (mod (+ ?d ?d) 10) ?t))
(combination L ?l&~ ?d&~ ?t)
(combination R ?r&~ ?d&~ ?t&~ ?l)
(test (= (mod (+ ?d ?d (* 10 ?1) (* 10 ?l) ?t)) 100)
(combination A ?a&~ ?d&~ ?t&~ ?l&~ ?r)
(combination E ?e&~ ?d&~ ?t&~ ?l&~ ?r&
(test (= (mod (+ ?d ?d (* 10 ?1) ((* 100 ?a) (* 100 ?a))
1000)
(+(* 100 ?e)(* 10 ?r) ?t)))
(combination N ?n&~ ?d&~ ?t&~ ?i&~ ?r&~ ?a&~ ?&e)
(combination B ?b&~ ?d&~ ?t&~ ?l&~ ?r&~ ?a&~ ?&e~ ?&n)
(test (= (mod (+ ?d ?d (* 10 ?1) (* 10 ?l)
(* 100 ?a) (* 100 ?a) (* 1000 ?r) (*1000 ?n)) 10000)
(+ (* 1000 ?b) (* 100 ?e) (* 10 ?r) ?t)
(combination O ?o&~ ?d&~ ?t&~ ?l~ ?r&~ ?a&~ ?e&~ ?n&~ ?b)
(combination G ?g&~ ?d&~ ?t&~ ?l&~ ?&~ ?a&~ ?e&~ ?n&~
?b~ ?o)
(test (= (+ ?d ?d
(* 10 ?l) (*10 ?l)
     (* 100 ?a) (* 100 ?a)
     (* 1000 ?r) (* 1000 ?n)
     (10000 ?e) (* 10000 ?o)
     (* 100000 ?q) (* 100000 ?d))
(+ (*100000 ?r) (* 10000 ?o) (* 1000 ?b) (* 100 ?e) (*
10 ?r) ?t)))
=>
(printout t "A solution is: " t)
(printout t "G = " ?g t)
```

```
(printout t "E = " ?e t)
(printout t "R = " ?r t)
(printout t "A = " ?a t)
(printout t "L = " ?l t)
(printout t "D = " ?d t)
(printout t "O = " ?o t)
(printout t "N = " ?n t)
(printout t "B = " ?b t)
(printout t "T = " ?t t)
(printout t " " ?g ?e ?r ?a ?l ?d t)
(printout t "+" ?d ?o ?n ?a ?l ?d t)
(printout t " " "---------" t)
(printout t " = " ?r ?o ?b ?e ?r ?t t t)
```

Теперь для поиска решения остается занести факты, соответствующие комбинациям из одной буквы и одной цифры, в список фактов. Это можно сделать командой assert:

```
(assert (combination D 0) (combination D 1)
(combination A 1)
(combination L 9)
...)
```

Однако, так как количество фактов равно 100, лучше поступить по-другому. Сопоставим буквам и цифрам следующие факты:

```
(number 0), (number 1), (number 2), ... , (number 9),
(letter G), (letter E), (letter R), (Letter A),
(letter L), (letter D), (letter O), (Letter N),
(letter B), (letter T).
```

Составим правило, по которому в список фактов будут заноситься факты, соответствующие комбинациям из одной буквы и одной цифры:

```
(defrule generate_combination
(number ?x)
(letter ?y)
=>
(assert (combination (?y ?x))))
```

Согласно данному правилу, если в списке фактов присутствуют факты, имеющие вид (number ?x), (letter ?y), где вместо переменных стоят вполне конкретные значения, в список фактов добавляется еще один факт — (combination ?y ?x), в котором вместо переменных стоят значения из фактов.

Наконец, составим правило, в соответствии с которым, во-первых, на экран будет выводиться условие задачи, а во-вторых, в список фактов будут заноситься факты, соответствующие буквам и цифрам. По-

скольку это правило должно выполняться всегда, то его условная часть будет пустой:

```
(defrule startup
=>
(printout t t "The problem is" t t)
(printout t " GERALD" t)
(printout t "+ DONALD" t)
(printout t "------------" t)
(printout t " = ROBERT" t t)
(assert
(number 0) (number 1) (number 2) (number 3) (number 4)
(number 5) (number 6) (number 7) (number 8) (number 9)
(letter G) (letter E) (letter R) (letter A) (letter L)
(letter D) (letter O) (letter N) (letter B) (letter T)))
```

Таким образом, рассмотренная программа состоит из трех правил: startup, generate_combination и find_solution. Первое из них выводит на экран условие задачи и добавляет в список факты, соответствующие отдельным буквам и цифрам. Второе правило, используя факты, введенные в список фактов первым правилом, заносит в список фактов факты, которые представляют собой всевозможные комбинации из десяти цифр и десяти букв. Третье правило позволяет найти среди этих фактов те, которые удовлетворяют условию задачи, и выводит на экран полученное решение.

Перейдем к порядку выполнения разработанной программы в ИО CLIPS.

10.4.2. Запуск ИО CLIPS

Система CLIPS реализована в виде исполняемого файла clipswin.exe, предназначенного для работы в операционных системах Windows 95, 98, NT 4.0, 2000, XP и их более поздних версиях. После запуска на экране появляется главное окно программы (рис. 10.1).

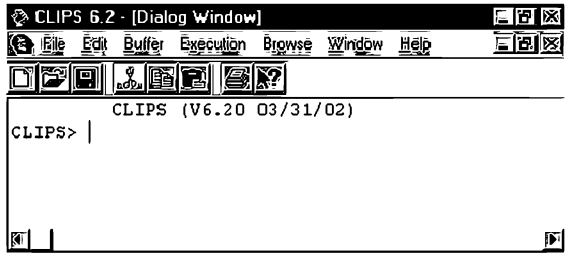


Рис. 10.1. Вид главного окна ИО CLIPS после запуска

Для возможности наблюдения за всеми изменениями, происходящими в состоянии CLIPS, выполните команду All Above (рис. 10.2). Данная команда открывает все окна. Окно Facts содержит факты из списка фактов. Окно Agenda содержит все правила из списка активных правил.

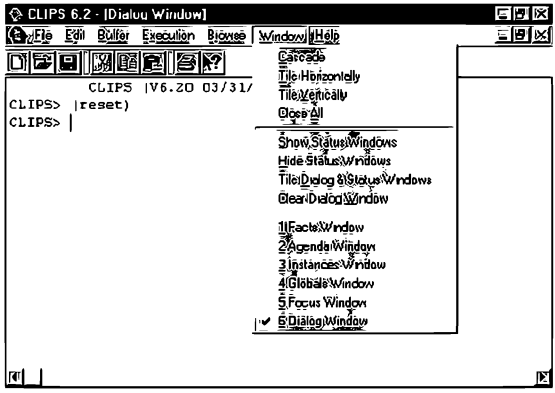


Рис. 10.2. Открытие всех окон

10.4.3. Ввод программы

Ввести программу в CLIPS можно непосредственно из диалогового окна, появившегося после запуска. Однако в этом случае все написанные правила после закрытия CLIPS будут потеряны. Чтобы этого не происходило, необходимо сохранить текст программы во внешнем файле. Для редактирования файлов в CLIPS имеется встроенный редактор.

Чтобы запустить редактор в режиме создания нового файла, необходимо в главном окне выбрать пункт меню File / New (рис. 10.3).

Внешний вид редактора представлен на рис. 10.4.

При запуске редактора автоматически создается новый файл с именем untitled. После того как ввод текста программы закончен, необходимо сохранить этот текст в каком-нибудь файле.

Сохранить текст в файле можно, выбрав в меню File редактора пункты Save или Save As (рис. 10.5). В появившемся диалоговом окне необходимо ввести имя файла (рис. 10.6).

По умолчанию файлы сохраняются с расширением *.clp. После сохранения текста программы в файле редактор можно закрыть.

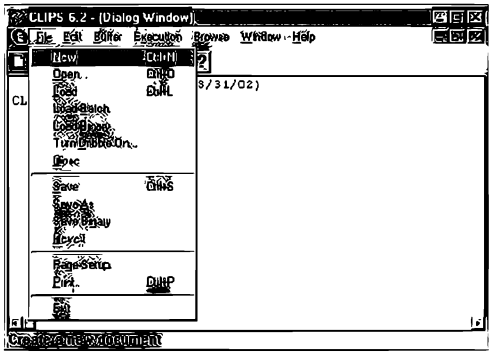


Рис. 10.3. Вызов редактора с созданием нового файла

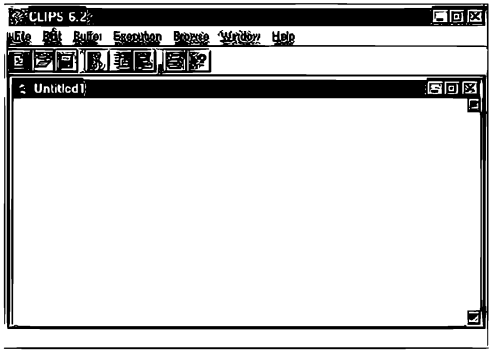


Рис. 10.4. Окно редактора

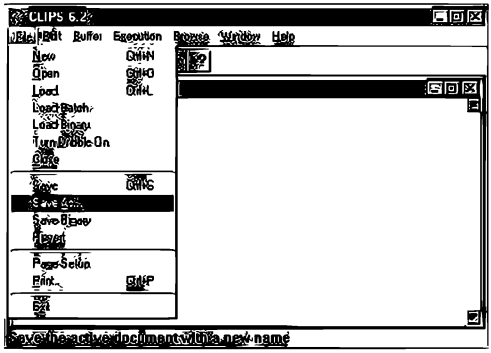


Рис. 10.5. Сохранение программы

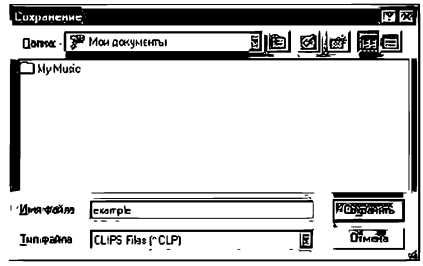


Рис. 10.6. Выбор файла для сохранения программы

10.4.4. Загрузка и запуск программы

Для загрузки в базу знаний CLIPS содержимого внешнего файла необходимо в главном окне выбрать пункт меню File/Load (рис. 10.7).

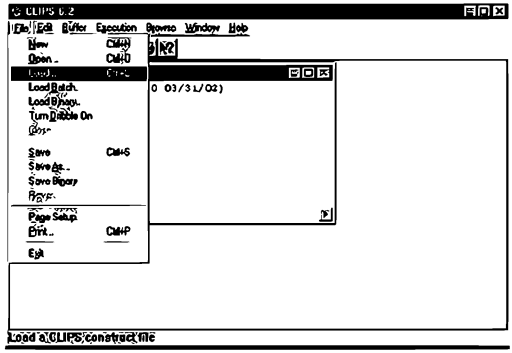


Рис. 10.7. Загрузка базы знаний из внешнего файла

В появившемся диалоговом окне необходимо выбрать имя требуемого этот файл и нажать кнопку «Открыть» (рис. 10.8).

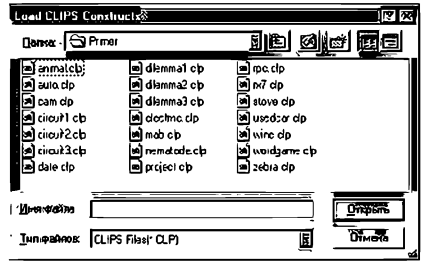


Рис. 10.8. Выбор файла для загрузки

Произойдет загрузка содержимого выбранного файла в базу знаний CLIPS. При наличии в списке фактов факта (initial-fact) CLIPS активирует правило без условий с идентификатором f–0. Занести этот факт в список можно, выбрав пункт меню Execution/Reset (рис. 10.9).

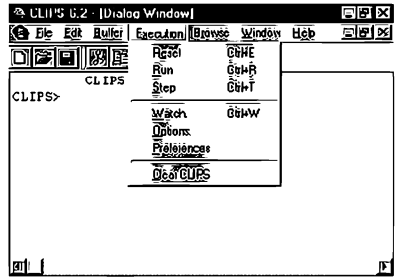


Рис. 10.9. Выбор команды Reset

Данная команда удаляет все факты из списка, а затем и включает в него исходный факт (initial-fact), а также все факты, описанные в конструкциях deffacts. Команда Reset не затрагивает самих правил, а лишь очищает список активных правил. После выполнения команды главное окно в режиме просмотра фактов будет выглядеть как показано на рис. 10.10.

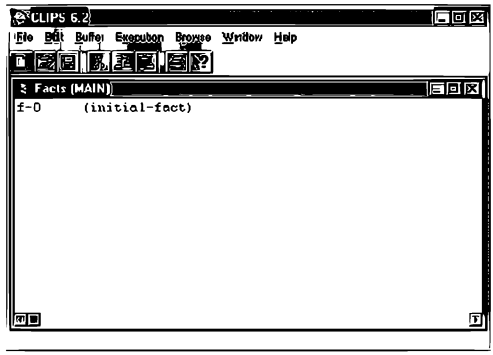


Рис. 10.10. Список фактов после сброса

Исходный факт активизирует правило, не содержащее условий, которое затем помещается в список активных правил — Agenda (рис. 10.11). В списке активных правил указывается, что это правило было активировано фактом с идентификатором f–0.

Итак, программа готова. Чтобы запустить ее на выполнение, необходимо выбрать команду Run (рис. 10.12).

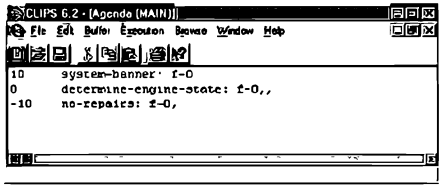


Рис. 10.11. Список активных правил

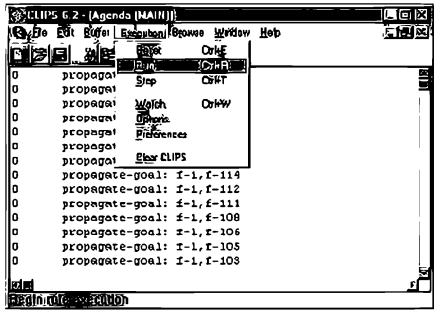


Рис. 10.12. Запуск программы

После этой команды CLIPS начнет выполнять все правила, которые находятся в списке активных правил. Выполнение программы завершается, когда в списке правил больше не останется ни одного активного правила, либо по прерыванию от пользователя с помощью сочетания клавиш «Ctrl-C».

Если после выполнения программы необходимо очистить базу знаний CLIPS, а также убрать все факты из списка фактов, т.е. привести CLIPS в начальное состояние, то необходимо воспользоваться командой Clear (рис. 10.13), очищающей CLIPS от всех правил и фактов.

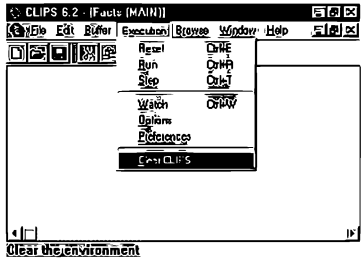


Рис. 10.13. Очистка текущего состояния ИО CLIPS

10.4.5. Работа программы

Рассмотрим подробнее, что происходит во время работы программы. Напомним, что в соответствии с рассматриваемым примером перед выполнением команды Run в списке активных правил было только одно правило с именем startup.

Сразу после запуска в результате выполнения данного правила в список фактов заносятся факты (рис. 10.14).

При этом в списке активных будет находиться сто правил, а окно примет вид, представленный на рис. 10.15.

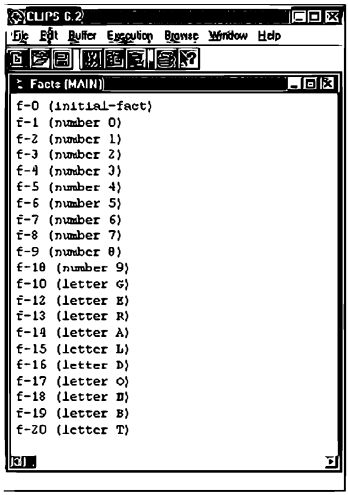


Рис. 10.14. Список фактов сразу после запуска

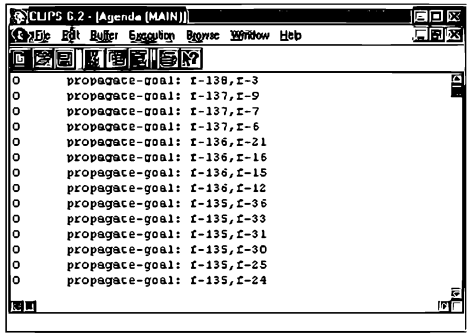


Рис. 10.15. Список активных правил

Это говорит о том, что в процессе выполнения программы правило generate-combination было активировано сто раз фактами из списка фактов.

После этого CLIPS начинает по порядку выполнять все правила из списка активных правил, проверяя при этом, не будут ли выполнены условия из третьего и первого правила. Очевидно, что первое правило больше выполняться не будет, а вот третье? Как только в списке фактов появятся факты, которые будут удовлетворять условиям третьего правила, оно незамедлительно будет помещено в список активных правил и выполнено. После выполнения всех ста шагов в списке фактов окажется еще сто фактов (рис. 10.16).

После этого в списке активных правил не останется ни одного правила, на экран будет выведено множество решений ребуса и программа завершит работу (рис. 10.17).

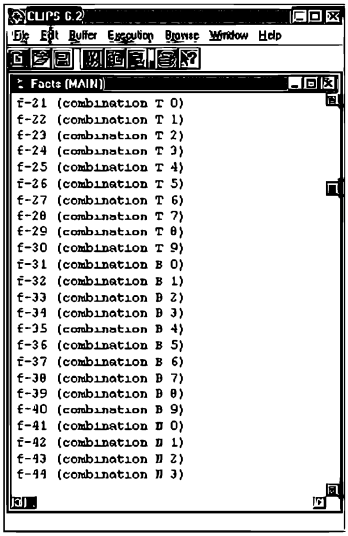


Рис. 10.16. Список фактов после выполнения ста шагов

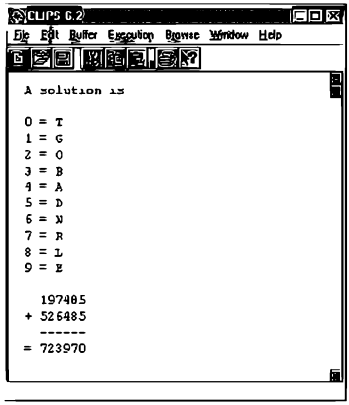


Рис. 10.17. Результат работы программы

10.4.6. Сохранение результатов работы

Для сохранения протокола и результатов работы необходимо сразу после запуска CLIPS выбрать пункт меню File/Turn Dribble On (рис. 10.18).

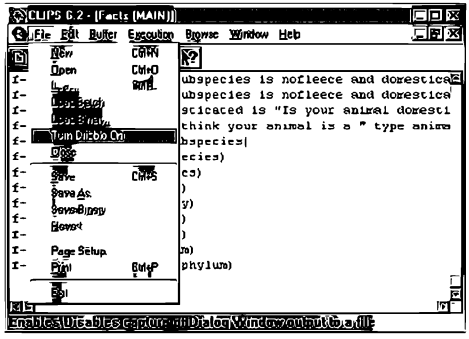


Рис. 10.18. Сохранение протокола работы

В появившемся диалоговом окне необходимо ввести имя файла, в котором будет сохранено содержимое главного окна CLIPS (рис. 10.19).

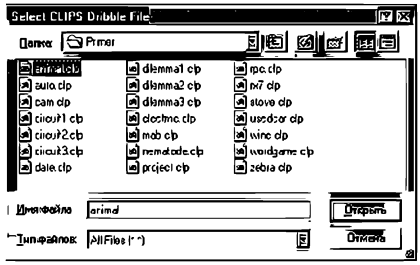


Рис. 10.19. Выбор файла для сохранения протокола работы

После подтверждения сохранения перед очисткой CLIPS необходимо выполнить команду Turn Dribble Off (рис. 10.20). По этой команде файл, в который записывалось содержимое главного окна CLIPS, будет закрыт.

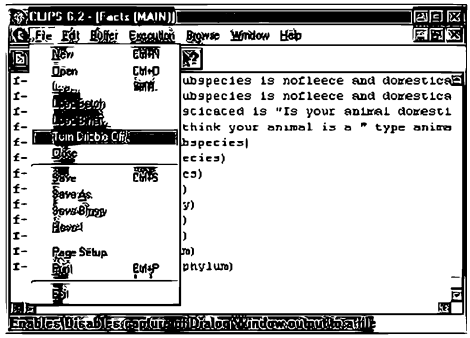


Рис. 10.20. Остановка записи протокола работы

Контрольные вопросы

1. Каков механизм организации логического вывода в ИО CLIPS?
2. Какие инструменты обеспечивают логический вывод в ИО CLIPS?
3. Как представлен интерфейс работы системы, разработанной в ИО CLIPS?
4. На каком языке программирования написана ИО CLIPS?

11. РАЗРАБОТКА ЭКСПЕРТНОЙ СИСТЕМЫ В ИО CLIPS

Инструментальная оболочка CLIPS в первую очередь предназначена для создания экспертных систем. В данном разделе будет рассмотрен пример создания хотя и не очень сложной, но полноценной экспертной системы диагностики неисправности двигателя автомобиля. Помимо этого разрабатываемая диагностическая экспертная система должна предоставлять пользователю соответствующие рекомендации по устранению неисправности. В программной реализации ЭС будут использованы управляющие команды CLIPS, такие как условный оператор (if-then-else) и оператор цикла с постусловием (while).

11.1. Подготовка исходных данных

Разработку любой экспертной системы следует начинать с выделения основных сущностей, имеющих значение при решении конкретной задачи и законов, скорее всего эмпирических, действующих над этими сущностями. В подавляющем большинстве случаев эту информацию получают при помощи эксперта — человека, хорошо знающего проблемную область и давно работающего в ней. Методы получения информации от эксперта и ее обработка выходят за рамки настоящей книги, но эта тема хорошо освещена в других работах.

Для решения выбранной конкретной задачи предположим, что в результате бесед с экспертом в области поиска неисправностей и ремонта автомобилей были установлены следующие эмпирические правила:

1. В один момент времени двигатель может находиться только в одном из трех технических состояний:
   - исправном, при котором ни один из параметров его функционирования не выходит за допустимые пределы, или, проще говоря, двигатель работает нормально;
   - работоспособном, при котором двигатель может функционировать, но один или несколько его параметров выходят за пределы допусков, т.е. двигатель работает ненормально;
   - двигатель не заводится.
2. Если двигатель работает нормально, это означает, что он и все его подсистемы (система зажигания, аккумулятор и т.д.) исправны и ремонт не требуется.

3. Если двигатель запускается, но работает ненормально, то это говорит, по крайней мере, о том, что исправен аккумулятор.

4. Если двигатель не запускается, то необходимо установить, происходит ли при запуске вращение вала двигателя. Если вал вращается, но двигатель при этом не заводится, то это может говорить о наличии плохой искры в системе зажигания. Если же вал двигателя не вращается, то это говорит об отсутствии искры, т.е. неисправности в самой системе зажигания.

5. Если двигатель не заводится, но вал вращается, необходимо проверить наличие топлива. При его отсутствии, вероятнее всего, автомобиль необходимо просто заправить.

6. Если двигатель не заводится, нужно также проверить, заряжен ли аккумулятор, и если нет, то зарядить его.

7. Если двигатель не заводится и существует вероятность плохой искры в системе зажигания, то необходимо проверить контакты. Контакты могут быть в одном из трех состояний: чистыми, опаленными и грязными. В случае опаленных контактов их необходимо заменить, в случае если контакты грязные, их достаточно просто очистить.

8. Если двигатель не заводится, искры нет и аккумулятор заряжен, то необходимо измерить электрическую проводимость катушки зажигания. В случае если ток не проходит через катушку, то ее необходимо заменить. Если катушка зажигания исправна, то необходимо заменить распределительные провода.

9. Если двигатель запускается, но при этом ведет себя инертно, не сразу реагирует на подачу топлива, то необходимо прочистить топливную систему.

10. Если двигатель запускается, но происходят перебои с зажиганием, то это говорит о наличии плохой искры в системе зажигания. Для устранения этой неисправности необходимо отрегулировать зазоры между контактами.

11. Если двигатель запускается, но в процессе работы стучит, то необходимо отрегулировать зажигание.

12. Если двигатель запускается, но не развивает нормальной мощности, то это может говорить об опаленных или загрязненных контактах (см. правило 7).

13. Возможны ситуации, когда состояние двигателя нельзя описать приведенными выше факторами и может потребоваться более глубокий анализ его технического состояния.

Имея эти данные, приступим к решению поставленной задачи.

11.2. Выделение сущностей

Из приведенных ранее правил можно выделить следующие сущности, имеющие значение при решении задачи:

1. Для решения задачи экспертной системе прежде всего необходимо знать, в каком ТС находился двигатель на момент начала диагностики (см. эмпирическое правило 1).

2. Большинство приведенных правил используют такой параметр, как наличие вращения двигателя. Согласно этим правилам двигатель может находиться в одном из двух ТС, которые определяются в зависимости от того, способен он вращаться (работать) или нет.

3. В правилах 4, 7, 8 и 10 используется оценка ТС системы зажигания. Система зажигания может быть в одном из трех ТС: исправном, ненормальной работы и нерабочего состояния.

4. Правила 6 и 8 используют понятие ТС аккумулятора. Аккумулятор может быть в одном из двух состояний: заряженным и разряженным.

В целях наглядности решения поставленной задачи воспользуемся для представления в CLIPS перечисленных выше данных упорядоченными фактами, зафиксированными в эмпирических правилах.

Пример. Факты, описывающие ТС двигателя и его подсистем:

```
;Группа фактов, описывающая ТС двигателя
working-state engine normal                ;двигатель исправен
working-state engine unsatisfactory        ;ненормальная работа
working-state engine does-not-start        ;двигатель не заводится

;Группа фактов, описывающая ТС вала двигателя
rotation-state engine rotates              ;вал двигателя вращается
rotation-state engine does-not-rotate      ;вал двигателя не вращается

;Группа фактов, описывающая ТС системы зажигания
spark-state engine normal                  ;зажигание исправно
spark-state engine irregular-spark         ;искра нерегулярна
spark-state engine does-not-spark          ;искры нет

;Группа фактов, описывающая ТС электрической системы
charge-state battery charged               ;аккумулятор заряжен
charge-state battery dead                  ;аккумулятор разряжен
```

Следует отметить, что факты, входящие в одну группу, являются взаимоисключающими, т.е. образуют полную группу несовместных событий. Наличие в системе сразу двух фактов из одной группы ли-

шено смысла, поскольку любая система не может одновременно находиться в двух различных состояниях.

Из постановки задачи следует, что разрабатываемая ЭС должна предоставлять пользователю рекомендации, позволяющие устранить обнаруженную неисправность. Из приведенных ранее правил можно выделить следующие рекомендации:

- добавить топливо (правило 5);
- зарядить аккумулятор (правило 6);
- заменить или почистить контакты (правило 7 или правило 12);
- заменить катушку зажигания или распределительные провода (правило 8);
- прочистить топливную систему (правило 9);
- отрегулировать зазоры между контактами (правило 10);
- отрегулировать зажигание (правило 11).

Необходимо помнить также о двух крайних случаях:

- двигатель исправен и ремонт не требуется;
- экспертная система не в состоянии диагностировать ТС.

Формализуем перечисленные рекомендации в виде фактов.

Пример. Факты, описывающие рекомендации по ремонту двигателя:

```
repair "Add gasoline"
repair "Charge the battery"
repair "Replace the points"
repair "Clean the points"
repair "Replace the ignition coil"
repair "Repair the distributor lead wire"
repair "Clean the fuel line"
repair "Point gap adjustment"
repair "Timing adjustment"
repair "No repair needed"
repair "Take your car to a mechanic"
```

Следует отметить, что одни и те же рекомендации могут выводиться различными правилами. Однако ТС двигателя при этом различны. Для обработки подобных ситуаций с помощью одного правила необходимо ввести два дополнительных факта.

Пример. Факты, описывающие мощность двигателя:

```
symptom engine not-low-output        ;мощность нормальная
symptom engine low-output            ;мощность мала
```

Кроме описанных ранее фактов, системе могут понадобиться факты, описывающие проявления неисправности. Однако в разрабатываемой версии экспертной системы таких фактов не будет, а о том, как без них обойтись, будет показано в следующем разделе. Приве-

денный ранее список фактов является достаточным для решения поставленной задачи. Приступим к следующему этапу — сбору исходной информации для диагностики.

11.3. Сбор информации

На ранних стадиях развития диагностических систем факты, описывающие проявление возникшей неисправности, вводились вручную самим пользователем. Однако в силу своей субъективности такой метод имеет ряд серьезных недостатков:

- пользователь может забыть о существенных деталях или, наоборот, указать ввести избыточную информацию, что может помешать нормальной работе системы;
- факты, описывающие проявление неисправности, должны иметь строго определенный формат, и случае ошибки со стороны пользователя система не сможет их обработать.

Для устранения указанных недостатков в разрабатываемой ЭС необходимо реализовать правила, которые в зависимости от той или иной ситуации будут задавать пользователю необходимые вопросы и получать ответ в строго заданной форме. Дальнейшая диагностика будет производиться с учетом предыдущих ответов на вопросы, заданных пользователю. Эти ответы будут формировать описание текущей ситуации с помощью фактов, приведенных ранее.

Для реализации подобной архитектуры необходимо реализовать функцию, задающую пользователю произвольный вопрос и получающую ответ из заданного набора корректных ответов.

Пример. Функция ask-question:

```
(deffunction ask-question (?question $ ?allowed-values)
   (printout t ?question)
   (bind ?answer (read))
   (if (lexemep ?answer)
      then
      (bind ?answer (lowcase ?answer)))
      (while (not (member ?answer ?allowed-values)) do
         (printout t ?question)
         (bind ?answer (read))
         (if (lexemep ?answer)
            then
            (bind ?answer (lowcase ?answer))))
   ?answer
)
```

Функция содержит два аргумента: простую переменную question, которая содержит текст вопроса, и составную переменную allowed-values с набором допустимых ответов. Сразу после вызова функция выводит на экран соответствующий вопрос и читает ответ пользователя в переменную answer. Если переменная answer содержит текст, то он будет принудительно приведен к верхнему регистру. После этого функция проверяет, является ли полученный ответ одним из заданных корректных ответов. Если нет, то процесс повторится до получения корректного ответа, иначе функция вернет ответ, введенный пользователем.

Будет также очень полезно определить функцию, задающую пользователю вопрос, допускающий ответ в виде «да/нет», так как это один из самых распространенных типов вопросов.

Пример. Функция yes-or-no-p:

```
(deffunction yes-or-no-p (?question)
   (bind ?response (ask-question ?question yes no y n))
   (if(or (eq ?response yes) (eq ?response y))
      then
         TRUE
      else
         FALSE)
)
```

Функция yes-or-no-p вызывает функцию ask-question с постоянным набором допустимых ответов: yes, no, y и n. В случае если пользователь ввел ответ yes или y, функция возвращает значение true, иначе — false. Обратите внимание, что поскольку функция yes-or-no-p использует функцию ask-question, то она должна быть определена после нее.

11.4. Диагностические правила

Для упрощения реализации разрабатываемой ЭС введем следующее ограничение: за один запуск система может предоставить пользователю только одну рекомендацию по исправлению неисправности. В случае если в машине несколько неисправностей, то систему нужно будет последовательно вызывать несколько раз, удаляя обнаруженную на каждом новом шаге неисправность. Таким образом, одним из образцов всех диагностических правил будет (not (repair?)), гарантирующий, что диагноз еще не поставлен.

Первым реализуем правило, определяющее ТС двигателя в целом (см. правило 1).

Пример. Правило determine-engine-state:

```
(defrule determine-engine-state ""
   (not(working-state engine?))
   (not(repair ?))
   =>
   (if (yes-or-no-p "Does the engine start (yes/no)?")
      then
      (if (yes-or-no-p "Does the engine run normally
      (yes/no)? ")
         then
            (assert(working-state engine normal))
         else
             (assert(working-state engine
             unsatisfactory)))
      else
         (assert(working-state engine does-not-start)))
)
```

Условный элемент (not (working-state engine ?)) гарантирует, что ТС двигателя в целом еще не определено. Если это так, то пользователю задаются соответствующие вопросы и в систему добавляется факт, описывающий текущее ТС двигателя.

Реализуем правило, определяющее, вращается ли вал двигателя, в случае если он не заводится при попытке запуска.

Пример. Правило determine-rotation-state:

```
(defrule determine-rotation-state ""
   (working-state engine does-not-start)
   (not(rotation-state engine ?))
   (not(repair ?))
   =>
   (if(yes-or-no-p "Does the engine rotate (yes/no)? ")
      then
         (assert(rotation-state engine rotates))
         (assert(spark-state engine irregular-spark))
      else
         (assert(rotation-state engine does-not-rotate))
         (assert(spark-state engine does-not-spark)))
)
```

Это правило выполняется в случае, если ТС двигателя в целом определено и известно, что он не заводится. Кроме того, условный элемент (not (rotation-state engine ?)) гарантирует, что это правило еще не вызывалось. В зависимости от ответа пользователя правило добавляет соответствующий набор фактов (см. правило 4).

Реализуем правила 5 и 6, выполняемые действия которых понятны без дополнительных комментариев.

Пример. Правила determine-gas-level и determine-battery-state:

```
(defrule determine-gas-level ""
   (working-state engine does-not-start)
   (rotation-state engine rotates)
   (not (repair?))
   =>
   (if (not(yes-or-no-p "Does the tank have any gas in
   it (yes/no)? "))
      then
         (assert(repair "Add gas.")))
)
(defrule determine-battery-state ""
   (rotation-state engine does-not-rotate)
   (not(charge-state battery ?))
   (not(repair ?))
   =>
   (if(yes-or-no-p "Is the battery charged (yes/no)? ")
      then
         (assert(charge-state battery charged))
      else
         (assert(repair "Charge the battery."))
         (assert(charge-state battery dead)))
)
```

Следует отметить, что правило determine-battery-state, помимо определения возможной неисправности, также применяется для добавления в систему факта, описывающего текущее состояние аккумулятора, который может быть использован другими правилами.

При реализации правила 7 необходимо обратить внимание на то, что рекомендации, предоставляемые этим правилом, подходят для двух в корне отличающихся ситуаций: во-первых, в случае если двигатель не заводится и существует вероятность плохой искры в системе зажигания (правило 7); во-вторых, в случае если двигатель запускается, но не развивает нормальной мощности (правило 12). Выполним реализацию этих правил.

Пример. Правила determine-low-output и determine-point-surface-state:

```
(defrule determine-low-output ""
   (working-state engine unsatisfactory)
   (not(symptom engine low-output I not-low-output))
   (not(repair ?))
   =>
   (if(yes-or-no-p "Is the output of the engine low
   (yes/no)?")
      then
         (assert(symptom engine low-output))
      else
```

```
      (assert(symptom engine not-low-output)))
)
(defrule determine-point-surface-state ""
   (or(and(working-state engine does-not-start)
   (spark-state engine irregular-spark))
   (symptom engine low-output))
   (not(repair ?))
   =>
   (bind ?response(ask-question "What is the surface
state of
   the points (normal /burned /contaminated)?"
   normal burned contaminated))
   (if(eq ?response burned)
      then
         (assert(repair "Replace the points."))
      else
         (if(eq ?response contaminated)
            then
               (assert(repair "Clean the points."))))
)
```

Правило determine-low-output определяет, имеет ли место низкая мощность двигателя или нет. Правило determine-point-surface-state адекватно реагирует на условия, заданные в правилах 7 и 12. Обратите внимание на использование условных элементов or и and, которые обеспечивают одинаковое поведение правила в двух абсолютно разных ситуациях. Кроме того, правило determine-point-surface-state отличается от приведенных ранее тем, что непосредственно использует функцию ask-question вместо yes-or-no-p, так как в данный момент пользователю задается вопрос, подразумевающий три варианта ответа. Выполним реализацию оставшихся правил 8—11.

Пример.

```
(defrule determine-conductivity-test ""
   (working-state engine does-not-start)
   (spark-state engine does-not-spark)
   (charge-state battery charged)
   (not(repair ?))
   =>
   (if(yes-or-no-p "Is the conductivity test for the
   ignition
   coil positive (yes/no)? ")
      then
         (assert(repair "Repair the distributor lead
         wire."))
      else
         (assert(repair "Replace the ignition coil.")))
```

```
)
(defrule determine-sluggishness ""
   (working-state engine unsatisfactory)
   (not (repair ?))
   =>
   (if(yes-or-no-p "Is the engine sluggish (yes/no)? ")
      then
         (assert(repair "Clean the fuel line.")))
)
(defrule determine-misfiring
   (working-state engine unsatisfactory)
   (not(repair ?))
   =>
   (if(yes-or-no-p "Does the engine misfire (yes/no)? ")
      then
         (assert(repair "Point gap adjustment."))
         (assert(spark-state engine irregular-spark)))
)
(defrule determine-knocking ""
   (working-state engine unsatisfactory)
   (not(repair ?))
   =>
   (if(yes-or-no-p "Does the engine knock (yes/no)? ")
      then
         (assert(repair "Timing adjustment.")))
)
```

Из рассмотренного списка правил остались нереализованными правила 2, 3 и 13. В качестве реализации правила 13 используем правило no-repairs.

Пример. Правило no-repairs:

```
(defrule no-repairs ""
   (declare (salience-10))
   (not (repair ?))
   =>
   (assert (repair "Take your car to a mechanic."))
)
```

Обратите внимание на использование приоритета при определении этого правила. Все правила, приведенные в предыдущем подразделе, определялись с приоритетом, по умолчанию равным нулю. Использование для правила no-repairs приоритета, равного −10, гарантирует, что правило не будет выполнено, пока в плане решения задачи находится, по крайней мере, одно из диагностических правил. Если все активированные диагностические правила отработали и ни одно из них не смогло подобрать подходящую рекомендацию по устранению неисправности, то CLIPS запустит правило no-repairs, которое

просто порекомендует пользователю обратиться к более опытному механику.

Реализация правил 2 и 3 приведена далее.

Пример. Правила normal-engine-state-conclusions и unsatisfactory-engine-state-conclusions:

```
(defrule normal-engine-state-conclusions ""
   (declare(salience 10))
   (working-state engine normal)
   =>
   (assert(repair "No repair needed."))
   (assert(spark-state engine normal))
   (assert(charge-state battery charged))
   (assert(rotation-state engine rotates))
   )
(defrule unsatisfactory-engine-state-conclusions ""
   (declare(salience 10))
   (working-state engine unsatisfactory)
   (assert(charge-state battery charged))
   (assert(rotation-state engine rotates))
)
```

В этих правилах, наоборот, используется более высокий приоритет, что гарантирует их выполнение до выполнения любого диагностического правила (естественно, только в случае удовлетворения условий, заданных в левой части правил). Это избавит нашу систему от лишних проверок, а пользователя — от лишних вопросов.

Экспертная система фактически готова к работе. Добавим только вывод итоговой информации и правило, сообщающее пользователю о начале работы.

Пример. Правила system-banner и print-repair:

```
(defrule system-banner ""
   (declare (salience 10))
   =>
   (printout t crlf crlf)
   (printout t "*********************************" crlf)
   (printout t "* The Engine Diagnosis Expert System *"
crlf)
   (printout t "*********************************" crlf)
   (printout t crlf crlf)
)
(defrule print-repair ""
   (declare (salience 10))
   (repair ?item)
   =>
   (printout t crlf crlf)
```

```
    (printout t "Suggested Repair:")
    (printout t crlf crlf)
    (format t " %s%n%n%n" ?item)
)
```

11.5. Листинг программы

В данном разделе приведен полный листинг программы с подробными комментариями. Он дает целостную картину того, как работает созданная экспертная система и из каких частей она состоит.

Пример. Экспертная система диагностики двигателя:

```
;
; Приведенная далее экспертная система способна
; диагностировать некоторые неисправности автомобиля и
; предоставлять пользователю рекомендации
; по их устранению.
;
;=====================================================
; Вспомогательные функции
;=====================================================
; Функция ask-question задает пользователю вопрос,
; полученный в переменной ?question, и получает
; от пользователя ответ, принадлежащий списку допустимых
; ответов, заданному в $?allowed-values
(deffunction ask-question (?question $?allowed-values)
   (printout t ?question)
   (bind ?answer (read))
   (if(lexemep ?answer)
      then
         (bind ?answer (lowcase ?answer)))
         (while(not(member ?answer ?allowed-values)) do
            (printout t ?question)
            (bind ?answer (read))
            (if(lexemep ?answer)
               then
                  (bind ?answer (lowcase ?answer))))
    ?answer)
;-----------------------------------------------------
; Функция yes-or-no-p задает пользователю вопрос,
; полученный в переменной ?question, и получает
; от пользователя ответ yes(y) или; no(n). В случае
; положительного ответа функция возвращает значение
; TRUE, иначе – FALSE
(deffunction yes-or-no-p (?question)
   (bind ?response (ask-question ?question yes no y n))
```

```
      (if (or (eq ?response yes) (eq ?response y))
         then
            TRUE
         else
            FALSE))
;=======================================================
; Диагностические правила
;=======================================================
; Правило determine-engine-state определяет текущее
; состояние двигателя машины по ответам, получаемым
; от пользователя. Двигатель может находиться в одном
; из трех состояний: работать нормально (working-state
; engine normal), работать неудовлетворительно
; (working-state engine unsatisfactory) и не заводиться
; (working-state engine; does-not-start)
; (см. правило 1).
(defrule determine-engine-state ""
   (not(working-state engine ?))
   (not(repair ?))
   =>
   (if(yes-or-no-p "Does the engine start (yes/no)? ")
      then
         (if(yes-or-no-p "Does the engine run normally
            (yes/no) ? ")
         then
            (assert(working-state engine normal))
         else
            (assert(working-state engine
unsatisfactory)))
      else
         (assert(working-state engine does-not-start))))
;-------------------------------------------------------
; Правило determine-rotation-state определяет состояние
; вращения двигателя по ответу, получаемому
; от пользователя. Двигатель может вращаться
; (rotation-state engine rotates) или не вращаться
; (spark-state engine does-not-spark) (см. правило 4).
; Кроме того, правило делает предположение о наличии
; плохой искры или ее отсутствии в системе зажигания
(defrule determine-rotation-state ""
   (working-state engine does-not-start)
   (not(rotation-state engine ?))
   (not(repair ?))
   =>
   (if(yes-or-no-p "Does the engine rotate (yes/no)? ")
      then
         ; Двигатель вращается
         (assert(rotation-state engine rotates))
```

```
         ; Плохая искра
         (assert(spark-state engine irregular-spark))
      else
         ; Двигатель не вращается
         (assert(rotation-state engine does-not-rotate))
         ; Нет искры
         (assert(spark-state engine does-not-spark))))
;-------------------------------------------------------
; Правило determine-gas-level по ответу пользователя
; определяет наличие топлива в баке. В случае если
; топлива нет, пользователю выдается рекомендация
; по ремонту — машину необходимо заправить (repair "Add
; gas.") (см. правило 5). При появлении соответствующей
; рекомендации выполнение диагностических правил
; прекращается.
(defrule determine-gas-level ""
   (working-state engine does-not-start)
   (rotation-state engine rotates)
   (not (repair?))
   =>
   (if(not(yes-or-no-p "Does the tank have any gas in it
      (yes/no) ? "))
   then
      ; Машину необходимо заправить
      (assert (repair "Add gas."))))
;-------------------------------------------------------
; Правило determine-battery-state по ответу пользователя
; определяет, заряжен ли аккумулятор. В случае если это
; не так, пользователю выдается рекомендация
; по ремонту — Зарядите аккумулятор (repair "Charge the
; battery.") (см. правило 6). Кроме того, правило
; добавляет факт, описывающий состояние аккумулятора.
; Выполнение диагностических правил прекращается.
(defrule determine-battery-state ""
   (rotation-state engine does-not-rotate)
   ; Состояние аккумулятора еще не определено
   (not(charge-state battery ?))
   (not(repair ?))
   =>
   (if(yes-or-no-p "Is the battery charged (yes/no)? ")
      then
         ; Аккумулятор заряжен
         (assert(charge-state battery charged))
      else
         ; Зарядите аккумулятор
         (assert(repair "Charge the battery."))
         ; Аккумулятор разряжен
         (assert(charge-state battery dead))))
```

```
;-------------------------------------------------------
; Правило determine-low-output определяет, развивает ли
; двигатель нормальную выходную мощность или нет
; и добавляет в систему факт, описывающий эту
; характеристику (см. правило 12).
(defrule determine-low-output ""
   (working-state engine unsatisfactory)
   ; Мощность работы двигателя еще не определена
   (not(symptom engine low-output I not-low-output))
   (not(repair ?))
   =>
   (if(yes-or-no-p "Is the output of the engine low
   (yes/no)? ")
      then
         ; Низкая выходная мощность двигателя
         (assert(symptom engine low-output))
      else
         ; Нормальная выходная мощность двигателя
         (assert(symptom engine not-low-output))))
; -------------------------------------------------------
; Правило determine-point-surface-state определяет
; по ответу пользователя состояние контактов (см.
; правила 7, 12). Контакты могут находиться в одном
; из трех состояний: чистые, опаленные и загрязненные.
; В двух последних случаях пользователю выдаются
; соответствующие рекомендации. Выполнение
; диагностических правил прекращается.
(defrule determine-point-surface-state ""
   (or(and(working-state engine does-not-start)
   ;не заводится
      (spark-state engine irregular-spark))
      ;и плохая искра
      (symptom engine low-output));или низкая мощность
   (not (repair ?))
   =>
   (bind ?response (ask-question "What is the surface
   state of the points (normal /burned /contaminated)?"
      normal burned contaminated))
   (if (eq ?response burned)
      then
      ; Контакты опалены — замените контакты
      (assert (repair "Replace the points."))
   else
      (if (eq ?response contaminated)
   then
      ; Контакты загрязнены — почистите их (assert
      (repair "Clean the points."))))))
;-------------------------------------------------------
```

```
; Правило determine-conductivity-test по ответу
; пользователя определяет, пропускает ли ток катушка
; зажигания. Если нет, то ее следует заменить.
; Если пропускает, то причина неисправности —
; распределительные провода. Для нормальной работы
; правила необходимо убедиться, что аккумулятор
; заряжен и искры нет (см. правило 8)
; Выполнение диагностических правил прекращается.
(defrule determine-conductivity-test ""
   (working-state engine does-not-start)
   (spark-state engine does-not-spark) ;нет искры
   (charge-state battery charged) ;аккумулятор заряжен
   (not (repair ?))
   =>
   (if(yes-or-no-p "Is the conductivity test for the
ignition
      coil positive (yes/no)? ")
   then ; Замените распределительные провода
      (assert(repair "Repair the distributor lead
      wire."))
   else ; Замените катушку зажигания
      (assert(repair "Replace the ignition coil."))))
;-------------------------------------------------------
; Правило determine-sluggishness спрашивает
; пользователя, не ведет ли; себя машина инертно
; (не сразу реагирует на подачу топлива). Если такой
; факт обнаружен, то необходимо прочистить топливную
; систему (см. правило 9) и выполнение диагностических
; правил прекращается.
(defrule determine-sluggishness ""
   (working-state engine unsatisfactory)
   (not(repair ?))
   =>
   (if(yes-or-no-p "Is the engine sluggish (yes/no)? ")
      then
         ; Прочистите систему подачи топлива
         (assert(repair "Clean the fuel line."))))
;-------------------------------------------------------
; Правило determine-misfiring узнает — нет ли перебоев
; с зажиганием. Если это так, то необходимо
; отрегулировать зазоры между контактами (см. правило 10).
; Выполнение диагностических правил прекращается.
(defrule determine-misfiring ""
   (working-state engine unsatisfactory)
   (not(repair ?))
   =>
   (if(yes-or-no-p "Does the engine misfire (yes/no)? ")
      then
```

```
          ; Отрегулируйте зазоры между контактами
          (assert(repair "Point gap adjustment."))
          ; Плохая искра
          (assert(spark-state engine irregular-spark))))
;-------------------------------------------------------
; Правило determine-knocking узнает — не стучит ли
; двигатель. Если это так, то необходимо отрегулировать
; зажигание (см. правило 11). Выполнение диагностических
; правил прекращается.
(defrule determine-knocking ""
   (working-state engine unsatisfactory)
   (not(repair ?))
   =>
   (if(yes-or-no-p "Does the engine knock (yes/no)? ")
      then
          ; Отрегулируйте положение зажигания
          (assert(repair "Timing adjustment."))))
;=======================================================
; Правила, определяющие ТС подсистем автомобиля по ТС
двигателя
;=======================================================
; Правило normal-engine-state-conclusions реализует
; правило 2
(defrule normal-engine-state-conclusions ""
 (declare(salience 10))
 ; Если двигатель работает неудовлетворительно
 (working-state engine normal)
 =>
 ; то
 (assert(repair "No repair needed.")); ремонт не нужен
 (assert(spark-state engine normal)) ; зажигание в норме
 (assert(charge-state battery charged)); аккумулятор заряжен
 (assert(rotation-state engine rotates)); двигатель вращается
)
; -------------------------------------------------------
; Правило unsatisfactory-engine-state-conclusions
; реализует правило 3
(defrule unsatisfactory-engine-state-conclusions ""
 (declare(salience 10))
 ; Если двигатель работает нормально
 (working-state engine unsatisfactory)
 =>
 ; то
```

```
 (assert(charge-state battery charged)) ; аккумулятор заряжен
 (assert(rotation-state engine rotates)) ; двигатель вращается
)
;=======================================================
; Запуск и завершение
;=======================================================
; Правило no-repairs запускается в случае, если ни одно
; из диагностических правил не способно определить
; неисправность. Правило корректно прерывает выполнение
; экспертной системы и предлагает пройти более
; тщательную проверку (см. правило 13).
(defrule no-repairs ""
   (declare(salience-10))
   (not(repair ?))
   =>
   (assert(repair "Take your car to a mechanic.")))
;-------------------------------------------------------
; Правило print-repair выводит на экран диагностическое
; сообщение по устранению найденной неисправности.
(defrule print-repair ""
   (declare(salience 10))
   (repair ?item)
   =>
   (printout t crlf crlf)
   (printout t "Suggested Repair:")
   (printout t crlf crlf)
   (format t " %s%n%n%n" ?item))
;-------------------------------------------------------
; Правило system-banner выводит на экран название
; экспертной системы при каждом новом запуске.
(defrule system-banner " каждом новом запуске."
   (declare (salience 10))
   =>; каждом новом запуске,
   (printout t crlf crlf)
   (printout t "********************************" crlf)
   (printout t "* The Engine Diagnosis Expert System *"
   crlf)
   (printout t "********************************" crlf)
   (printout t crlf crlf))
```

Напомним, что среда CLIPS воспринимает только английский алфавит. Комментарии на русском языке даны для наглядности, однако при вводе программы в таком виде CLIPS выдаст сообщение об ошибке.

11.6. Выполнение программы

Для запуска программы наберите приведенный в примере листинг в текстовом редакторе (лучше использовать встроенный редактор CLIPS). Сохраните набранный файл, например, под именем auto. CLP.

После этого запустите CLIPS или, если он уже был у вас запущен, очистите его командой (clear). Загрузите созданный вами файл с помощью команды (load «auto. CLP»). При успешной загрузке файла конструкторов и отсутствии ошибок экран примет вид, показанный на рис. 11.1. Обратите внимание, что функция load вернула значение true. Если это не так, значит в синтаксисе определений функций или правил была допущена ошибка. При загрузке файла с помощью функции load* информация, отражающая процесс загрузки, на экран не выводится.

После удачной загрузки файла убедитесь, что все правила присутствуют в списке правил CLIPS, а функции — в списке функций. Проще всего это проконтролировать с помощью менеджеров правил и функций соответственно (рис. 11.2 и 11.3).

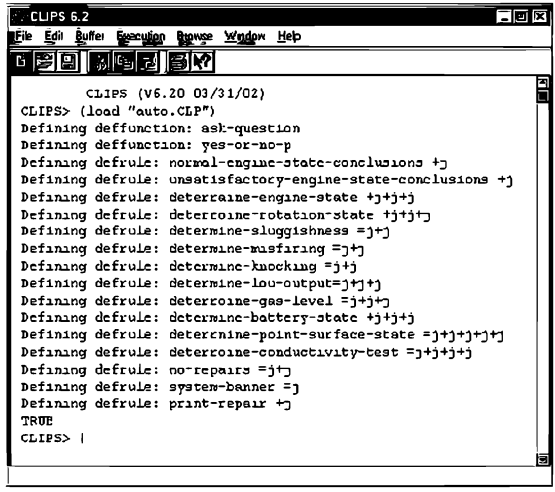


Рис. 11.1. Загрузка экспертной системы

Для запуска разработанной ЭС достаточно выполнить команду reset, которая добавит факт initial-fact, необходимый для правила system-banner, и команду run. После этого на экран выводится сооб-

щение «The Engine Diagnosis Expert System», которое означает, что система начала работать. Затем ЭС формирует серию вопросов, ответы на которые помогут ей оценить текущее ТС двигателя и подобрать соответствующую рекомендацию по ремонту.

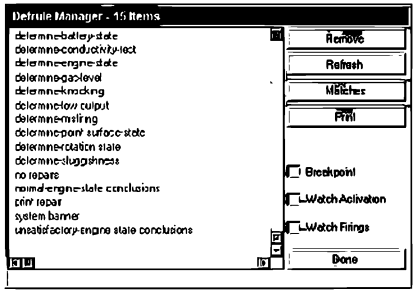


Рис. 11.2. Правила экспертной системы

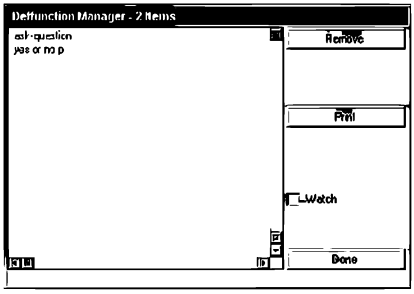


Рис. 11.3. Функции экспертной системы

Пример опроса пользователя системой показан на рис. 11.4, а результат ее работы — на рис. 11.5.

Если после завершения работы экспертной системы в списке фактов остаются факты, описывающие ТС двигателя, то их легко просмотреть с помощью команды Fact Window из меню Window.

Для повторного запуска экспертной системы необходимо еще раз выполнить команды reset и run.

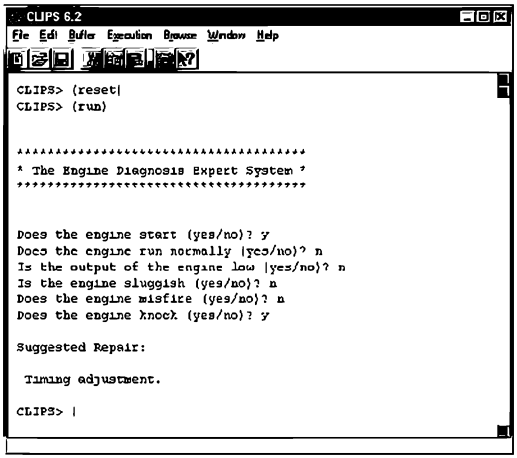


Рис. 11.4. Диалог с экспертной системой

Для того чтобы лучше понять механизм работы созданной ЭС и логический механизм вывода CLIPS, необходимо протестировать экспертную систему, по-разному отвечая на ее вопросы, а перед запуском системы сделать видимым окно фактов (Fact Window) и окно плана решения задачи (Agenda Window).

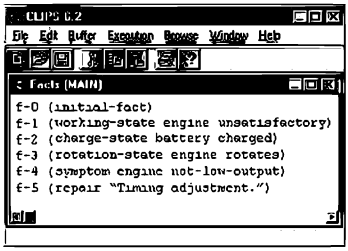


Рис. 11.5. Результаты работы экспертной системы

Рассмотренный пример доступен в Интернете по адресу: www.ghg.net/clips/download/executables/examples/auto. clp

В данном разделе был рассмотрен реальный пример построения экспертной системы в CLIPS. Он является наглядным подтверждением того, что с помощью фактов, функций и правил CLIPS можно построить работоспособную систему. Однако CLIPS содержит и более сложные конструкции, такие как родовые функции, объекты и модули, с помощью которых можно создавать гораздо более гибкие и мощные экспертные системы.

Контрольные вопросы

1. Определите основные структурные элементы программирования в ИО CLIPS.
2. Опишите особенности интерфейса ИО CLIPS.
3. Каковы особенности реализации логического вывода в ИО CLIPS?
4. К какому типу ИО относится CLIPS?
5. Опишите общую схему функционирования ЭС, разработанных с помощью ИО CLIPS.
6. Какие составляющие входят в структуру базы знаний CLIPS?
7. Какие стратегии принятия решений реализованы в ИО CLIPS?
8. Как проявляется свойство активности знаний?

12. СОЗДАНИЕ ПРОЕКТА ОНТОЛОГИИ С ПОМОЩЬЮ ИС PROTÉGÉ

Одной из основных проблем при построении баз знаний является организация большого объема данных таким образом, чтобы пользователи легко могли получить необходимую информацию. В связи с этим следует отметить, что онтологии представляют собой нечто гораздо большее, чем просто подход к описанию и классификации информации. Они могут использоваться для поддержки функционирования и развития распределенных интеллектуальных систем за счет следующих своих важнейших свойств:

- надежного семантического базиса в определении содержания;
- общей логической теории описания и классификации информации;
- дружественного интерфейса между пользователем и программой.

Protégé — это интегрированная инструментальная среда программирования разработки систем, основанных на знаниях, с использованием аппарата онтологий.

Protégé существенно облегчает работу с классами и экземплярами. Например, отдельный экземпляр может быть использован на уровне описания класса, а класс можно хранить, как экземпляр. Аналогичным образом слоты, которые в других оболочках используются только внутри классов, в Protégé могут рассматриваться в качестве самостоятельных классов.

Рассмотрим более подробно процесс создания проекта онтологии с помощью предложенной инструментальной среды и ее дальнейшего экспорта в ЭО CLIPS.

12.1. Создание нового проекта

Порядок создания нового проекта целесообразно рассмотреть в виде последовательности определенных действий.

Прежде всего необходимо запустить исполнимый файл Protege. exe, после чего на экране появится окно приглашения (рис. 12.1).

В этом диалоговом окне можно создать новый проект, открыть уже существующие проекты или получить необходимую справку, нажав соответствующие кнопки.

Выбираем создание нового проекта нажатием кнопки Create New Project. На экране появится окно мастера создания нового проекта (рис. 12.2), в котором необходимо выделить строку Protégé Files (.pont and.pins) и нажать кнопку Finish.

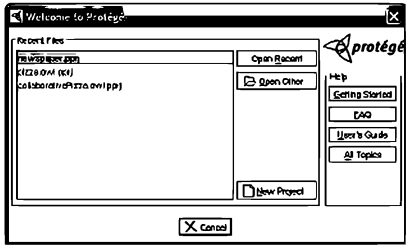


Рис. 12.1. Окно приглашения Protégé

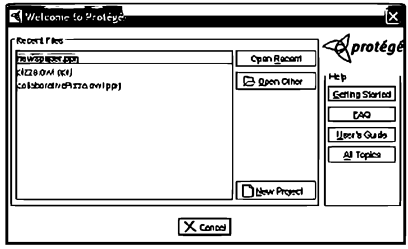


Рис. 12.2. Выбор типа нового проекта

В результате этого откроется главное окно ИС Protégé со стандартными вкладками (рис. 12.3), в котором будет отображена вновь созданная пустая база знаний, содержащая системные классы с корнем в классе: THING.

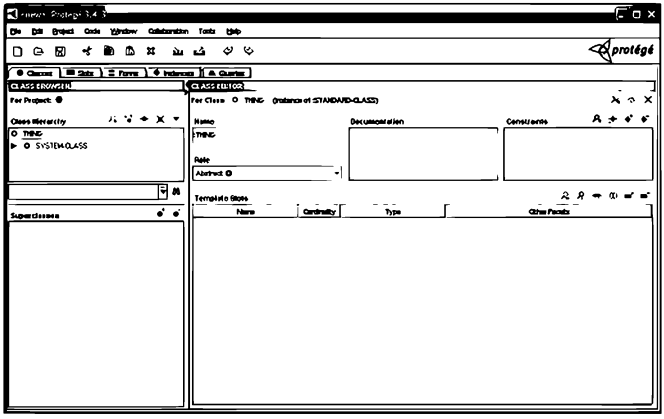


Рис. 12.3. Главное окно программы с открытым новым проектом

12.2. Структура проекта

Независимо от выбранного формата общая информация о проекте сохраняется в файле с расширением *.pprj (Protégé project). Создавать, открывать и сохранять проекты можно непосредственно с помощью

pprj-файла. При этом нет необходимости в присвоении имен другим файлам или обращения к ним, если только проект не импортируется.

Кроме файла проекта, Protégé сохраняет еще два файла, содержащих дополнительную информацию об онтологии и экземплярах, относящихся к данному проекту. При открытии проекта эти файлы загружаются автоматически. По умолчанию дополнительные файлы сохраняются в стандартном текстовом формате (Standard Text Files):

- текстовый файл с расширением *.pont (Protégé ontology), содержащий информацию о классах и слотах;
- текстовый файл с расширением *.pins (Protégé instances), содержащий информацию об экземплярах.

Однако при создании проекта можно выбрать и другие форматы:

- JDBC Database — создает проект как таблицу базы данных JDBC. Для этого эта база данных должна быть предварительно установлена и сконфигурирована;
- Resource Description Framework — проект сохраняется в формате RDF.

Если версия Protégé поддерживает другие дополнительные форматы, то они также будут отображены в списке, однако в любом случае для доступа к проекту также используется pprj-файл.

Перейдем к структурированию нового проекта, созданию классов и слотов.

12.3. Работа с классами

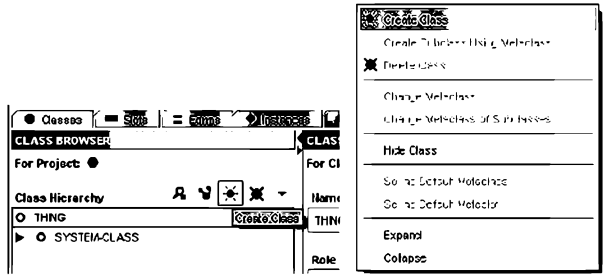
12.3.1. Создание нового класса

Для создания нового класса необходимо на панели Class Relationship (Отношения классов) выделить класс (рис. 12.4, *а*), который будет являться родительским для вновь создаваемого класса, а затем нажать кнопку Create Class (Создать класс).

Создать новый класс можно также, щелкнув правой кнопкой мыши по родительскому классу создаваемого класса и выбрав в выпадающем меню пункт Create Class (Создать класс) (рис. 12.4, *б*).

После выполнения одной из указанных ранее операций под выделенным классом добавится новый класс (рис. 12.5) с именем, сгенерированным системой случайным образом.

Для присвоения имени классу, выбора его роли, создания ограничений, а также создания и редактирования его слотов используется редактор классов, вид которого представлен на рис. 12.6.



а *б*

Рис. 12.4. Создание нового класса:
а — в панели отношений классов; *б* — через выпадающее меню

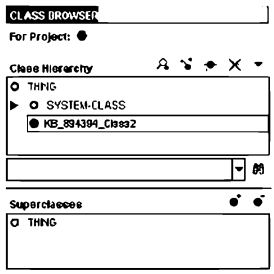


Рис. 12.5. Вновь созданный класс KB_894_Class2

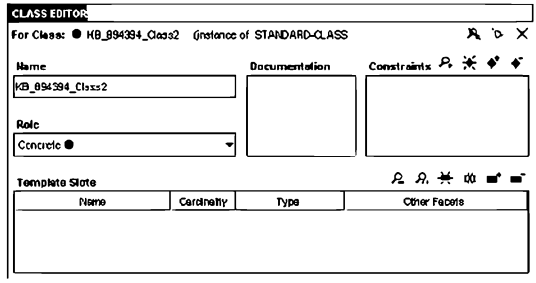


Рис. 12.6. Редактор классов

12.3.2. Создание экземпляра класса

Для создания или редактирования экземпляра прежде всего надо открыть вкладку Instances (Экземпляры) главного окна (рис. 12.7).

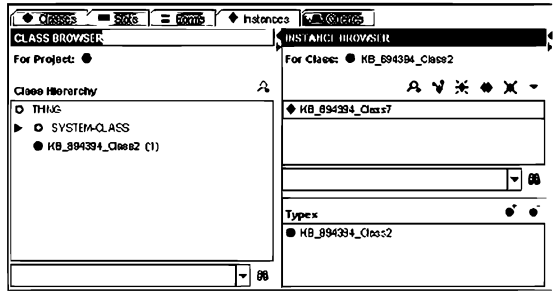


Рис. 12.7. Создание нового экземпляра класса

Затем, выделив класс, экземпляр которого требуется создать, нажимаем кнопку Create Instance (Создать экземпляр) на панели просмотра экземпляров (Instance Browser). Вновь созданный экземпляр с именем KB_894_Class7 отобразится в окне.

12.3.3. Инструменты работы с классами

Прежде чем рассмотреть порядок редактирования классов и их свойств, раскроем назначение кнопок на панели просмотра классов (Class Browser), поскольку, где бы ни появились кнопки, аналогичные этим, нажатие на них даст один и тот же результат (рис. 12.8).

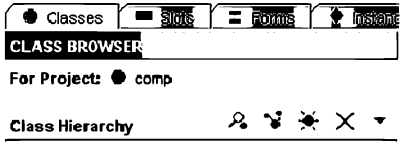


Рис. 12.8. Кнопки на панели просмотра классов

Нажатие кнопки View Class (Просмотр класса) открывает форму, позволяющую просматривать и изменять свойства класса (рис. 12.9).

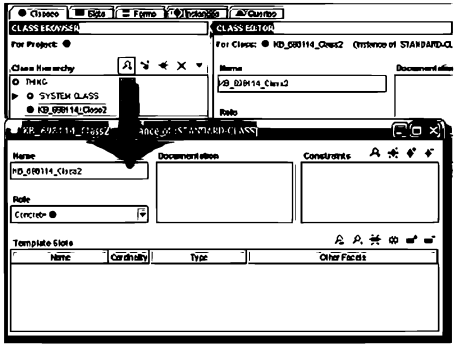


Рис. 12.9. Форма редактирования класса

Кнопка View References to Class (Просмотр связей класса) открывает окно, в котором отображено место рассматриваемого класса в общей иерархии, а также раскрыты значения его слотов (рис. 12.10).

Нажатие кнопки Create Class (Создать класс) приводит к созданию нового класса, как это было показано ранее (см. подраздел 12.4.1), в качестве подкласса выделенного класса.

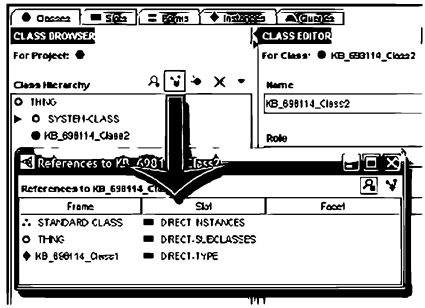


Рис. 12.10. Просмотр связей класса

Кнопка Delete Class (Удалить класс) позволяет удалить выделенный класс и все его подклассы.

Для определения и редактирования свойств класса, выбранного на панели просмотра, используется форма класса (см. рис. 12.9). Как только изменения внесены в форму классов, они сразу же вступают в действие, однако для сохранения изменений в проекте необходимо выбрать пункт меню Project/Save (Сохранить проект) в меню главного окна программы. В форме классов (см. рис. 12.9) имеется пять полей редактировании. Рассмотрим их более подробно.

Поле Name (имя класса) дает возможность присвоить имя классу. При создании класса по умолчанию ему присваивается имя, сгенерированное системой случайным образом. Имя нового или уже существующего класса можно изменить, выделив его и набрав новое. При написании имен классов следует помнить, что имя класса должно быть уникальным в пределах одной базы знаний, а система Protégé чувствительна к регистру. Кроме этого, каждое слово в имени класса рекомендуется начинать с большой буквы, остальные писать строчными буквами, а слова отделять подчеркиванием.

Выпадающий список Role (роль класса) позволяет выбрать роль класса: Concrete (конкретный) или Abstract (абстрактный). Конкретные классы могут иметь непосредственные экземпляры, тогда как абстрактные — нет. По умолчанию роль в ИС Protégé установлена в значение Concrete.

ИС Protégé не накладывает каких-либо ограничений на роли классов. Тем не менее существует правило, в соответствии с которым все классы нижнего уровня (классы-листья) должны быть конкретными, а классы верхнего уровня (внутренние, не листья) — абстрактными.

Поле Constraints (ограничения класса) позволяет создавать, редактировать, добавлять и удалять ограничения, накладываемые на класс. Помимо этого, ограничения класса можно определять программно.

Поле Documentation (информация о классе) позволяет вводить комментарий или текстовое описание класса, а также любые специальные примечания. Заполнять это поле необязательно, хотя делать это рекомендуется, особенно при разработке сложных и масштабных приложений.

В таблице Template Slots (список слотов) отображаются названия (Name) прямых и унаследованных слотов выбранного класса, их вид (Cardinality) — однозначный или многозначный, а также тип допустимых значений (Type).

Следует отметить, что перечисленные поля также доступны для просмотра и редактирования в окне редактора класса (см. рис. 12.6).

12.4. Работа со слотами

Создавать, редактировать, добавлять и удалять слоты класса в ИС Protégé можно как в таблице Template Slots (список слотов), расположенной в окне редактора класса (см. рис. 12.6), так и в аналогичной таблице формы редактирования класса (см. рис. 12.9).

Для создания нового слота к существующему классу (рис. 12.11) на форме редактирования класса необходимо нажать кнопку ▣ в разделе списка слотов.

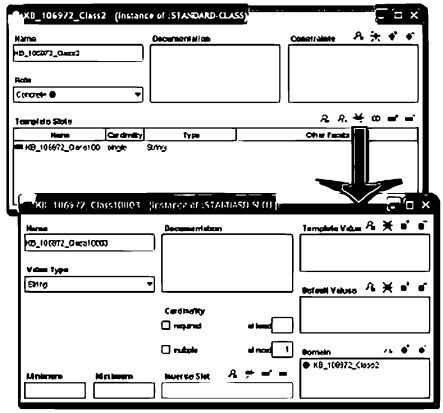


Рис. 12.11. Создание слота

На экран выводится форма редактирования слота, содержащая следующую информацию:

- Name — имя слота;
- Value Type — тип значения слота;
- Cardinality — мощность слота;
- Minimum — минимальное допустимое значение слота;
- Maximum — максимальное допустимое значение слота;
- Documentation — комментарий, пояснения к слоту;
- Inverse Slots — слоты, обратные редактируемому;
- Template Values — шаблонные значения, определенные для слота;
- Defaults — значения слота по умолчанию.

Редактирование слотов имеет ряд особенностей. Например, слот можно редактировать одним из двух способов: на верхнем уровне или в классе.

Редактирование слота на верхнем уровне влияет на слот везде, где бы он ни появился, включая вкладку Slots и любые классы, к которым этот слот присоединен.

Редактирование слота на уровне класса (переопределение) влияет только на слот этого класса и его подклассов. На вкладке Slots и любых неродственных классах слот останется неизменным.

Выбор режима редактирования слота осуществляется нажатием соответствующей кнопки на панели списка слотов Template Slots:

- View Top-Level Slot — для редактирования на верхнем уровне;
- View Slot Overrides — для редактирования в классе.

Также можно воспользоваться выпадающим меню, для чего необходимо выделить слот в списке, щелкнуть по нему правой кнопкой мыши и выбрать требуемый режим редактирования (рис. 12.12).

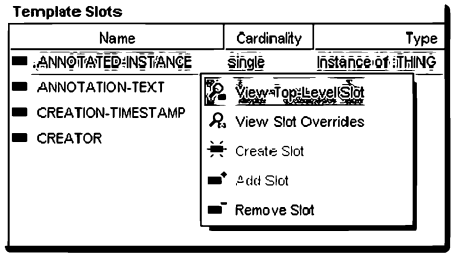


Рис. 12.12. Выбор режима редактирования слота

Для отдельных типов слотов под панелью Value Type появляется дополнительная панель:

- для типа Class — панель Allowed Parents (разрешенные родители);
- для типа Instance — панель Allowed Classes (разрешенные классы);
- для типа Symbol — панель Allowed Symbols (разрешенные символы).

Поле Cardinality (мощность слота) позволяет определить число значений, допустимых или необходимых для слота. По умолчанию слоту разрешено иметь только одно значение, т.е. он может иметь одно значение или вообще не иметь значений.

Для определенных типов слотов применяются некоторые ограничения.

Класс может иметь слоты, унаследованные от родительского класса. Если слот определен глобально или унаследован, то мощность слота можно поменять только с Multiple на Single, но не наоборот.

Слот типа Any может быть ограничен до одного из других типов (Boolean, Class, Float, Instance, Integer, String, или Symbol).

Для слота типа Class разрешенные родители могут быть изменены только на подклассы разрешенных классов в родительском классе.

Для слота типа Instance разрешенные классы можно изменить только на подклассы разрешенных классов в родительском классе.

Поля минимального и максимального значений (Minimum и Maximum) доступны только для слотов с типом значения Integer или Float.

Любой прямой слот, присутствующий в списке слотов Template Slots, может быть удален из класса. Для этого необходимо выделить требуемый слот и нажать кнопку Remove Slot (исключить слот). Слот будет исключен из класса, однако останется в базе знаний и будет отображаться в диалоговом окне выбора слотов Select Slots.

12.5. Сохранение проекта в формате RDF

Поздние версии Protégé поддерживают создание и редактирование данных в виде так называемой схемы описания ресурсов RDF (Resource Description Framework). ИС Protégé позволяет как разрабатывать схемы RDF и создавать данные о соответствующих экземплярах, так и просматривать и редактировать уже существующие RDF-файлы, созданные в других оболочках.

Формат RDF. Схема описания ресурсов RDF представляет собой стандартную модель обмена данными в сети Интернет. RDF обладает средствами, облегчающими объединение данных даже в случае, когда

лежащие в основе схемы данных отличаются. Кроме этого, RDF поддерживает эволюцию схем в течение времени, не требуя от потребителей данных внесения каких-либо изменений.

RDF расширяет структуру ссылок. Использование этой простой модели позволяет смешивать структурированные и слабоструктурированные данные, публиковать их, а также разделять их между разными приложениями.

Ссылочная структура RDF образует направленный, помеченный граф, в котором ребра представляют собой именованные ссылки между ресурсами, образующими в свою очередь узлы графа. Представление системы в виде графа является наиболее простой возможной ментальной моделью для RDF и часто используется в простых для понимания визуализированных объяснениях.

Основополагающим для RDF является понятие модели данных. Это есть набор фактов и семантических связей между ними, задаваемый конкретным RDF-документом. RDF обеспечивает последовательный стандартный способ описания и работы практически с любыми Интернет-ресурсами: от текстовых страниц и графиков до аудиофайлов и видеоклипов. Он предлагает синтаксические возможности для взаимодействия сетей и формирует базовый слой для создания семантической сети. RDF определяет управляемые графы связей, представленные тройками объект — атрибут — значение. RDF прекрасно подходит для представления данных, их структуры и связей.

Таким образом, применение специально разработанных RDF-схем в качестве средства описания онтологии предметной области позволяет представить информацию, относящуюся к различным разделам знаний, единым и понятным для различных Интернет-приложений способом.

Сохранить проект Protégé в формате RDF можно двумя способами:

- при создании проекта в диалоговом окне Select Format выбрать формат Resource Description Framework (RDF);
- при сохранении проекта, созданного ранее в другом формате, выбрать пункт меню Convert Project To Format..., а затем в диалоговом окне Select Format выбрать RDF Files (рис. 12.13).

После этого на экране отобразится диалоговое окно RDF Files (рис. 12.14). Для выбора или изменения имени и (или) местоположение pprj-файла необходимо ввести необходимую информацию в строку Project диалогового окна. Задать новое местоположение файла проекта можно также нажатием кнопки — Browse for File (Выбор файла). Если местоположение проекта не определено дополнительно,

то по умолчанию он будет сохранен в том же каталоге, где установлена ИС Protégé.

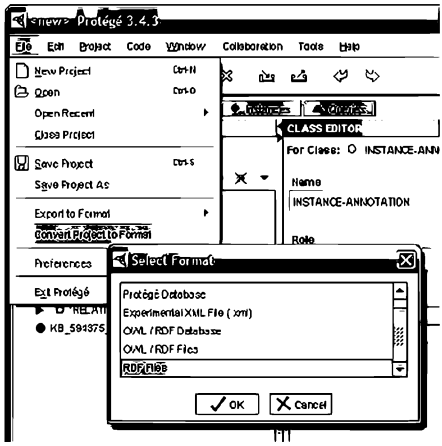


Рис. 12.13. Сохранение проекта в RDF



Рис. 12.14. Выбор файлов для сохранения в формате RDF

В результате ИС Protégé создаст два файла: один для схемы (классы и слоты), а другой для экземпляров. По умолчанию эти файлы используют имена существующего pprj-файла. Для изменения имени файла-схемы в строке Classes file name диалогового окна необходимо указать новое имя файла с расширением *.rdfs.

Для изменения имени файла экземпляров в строке Instances file name диалогового окна необходимо указать новое имя файла с расширением *.rdf.

В поле Namespace вводится пространство имен RDF проекта. Все классы, слоты и экземпляры находятся в этом пространстве имен.

После внесения изменений для их сохранения необходимо нажать кнопку OK.

12.6. Экспорт онтологии в формат ЭО CLIPS

Выберите пункт главного меню (рис. 12.15) File/Export to Format/CLIPS («Файл/Экспорт в Формат/CLIPS»).

В появившемся окне требуется выполнить ряд логически понятных действий, в итоге которых будут получены файлы *.pont и *.pins, первый из которых содержит созданные вами классы, представленные в формате CLIPS, а второй — объекты этих классов.

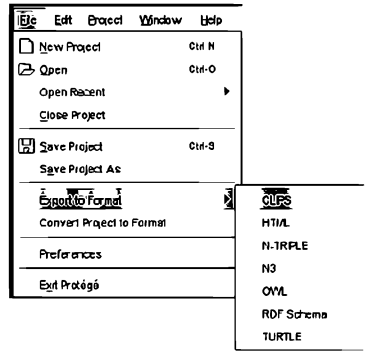


Рис. 12.15. Экспорт проекта онтологии в формат CLIPS

Контрольные вопросы

1. Как представить онтологию в Protégé?
2. Объясните назначение редактора классов для представления знаний.
3. В какие форматы можно передать онтологию, созданную в Protégé?
4. Для каких целей может потребоваться экспорт онтологий?

ЗАКЛЮЧЕНИЕ

За всю историю ИИ было предпринято множество исследований, целью которых было выяснить, каким образом и многообразие сведений об отдельных фактах, отношениях, и общие принципы построения окружающего нас мира можно использовать в компьютерной программе, которая ориентирована на построение логического рассуждения, направленного на достижение определенной цели.

Эти исследования включали использование конструкций следующих видов (чаще в чистом виде, но иногда и в комбинации):

- правил в форме «если имеет место это условие, то примени этот оператор»;
- сетей, в которых узлы соответствуют концепциям, а дуги — отношениям между ними;
- логических формул, представляющих отдельные факты и принципы, включая управляющую информацию о том, когда применить то или иное соответствие.

Как это представить в виде данных в машинных формах?

Действительно, при использовании инструментальных средств первого типа (языки программирования) в задачу разработчика входит программирование всех компонентов ЭС на языке довольно низкого уровня. Использование инструментальных средств второго типа (языки представления знаний и инженерии знаний) позволяет значительно повысить уровень языка, что, как правило, приводит к некоторому снижению эффективности. Инструментальные средства третьего типа (средства автоматизации разработки и проектирования систем ИИ) позволяют разработчику не программировать все или часть компонентов ЭС, а выбирать их из заранее составленного набора. При применении инструментальных средств четвертого типа (оболочки ЭС) разработчик ЭС полностью освобождается от работ по созданию программ, так как берет готовую пустую ЭС.

В пособии представлены декомпозиция проблемных областей, реализуемая с помощью различных инструментальных средств, и преемственность между этими представлениями. Кроме того, показано, каким образом проблемная область, представленная с помощью Protégé, экспортируется в формат ИС CLIPS.

Поэтому сегодня прототипы ЭС создаются с помощью оболочек или средств автоматизации разработки, так как основная цель на этой

фазе — минимизация трудозатрат, а промышленные или коммерческие ЭС разрабатываются на языках представления знаний или языках программирования, обеспечивающих более высокую эффективность реализации.

При разработке реальных экспертных систем также часто используются так называемые языки искусственного интеллекта типа LISP и Пролог. Однако в последнее время наметился переход к применению языков традиционного программирования (Borland C, C++ и др.), что позволяет упростить интеграцию создаваемой экспертной системы в эксплуатируемые информационно-аналитические системы и снизить требования к быстродействию и объему памяти.

Необходимо отметить, что современные ЭС еще слабо соответствуют многим из требований, предъявляемых к ним, но вывод о том, что они не обладают «пониманием» хотя бы в отдельной предметной области, также, на наш взгляд, спорен.

Материал, позволяющий сделать акцент на способах декомпозиции предметных областей, может являться основой следующего учебного пособия, где будут выделены особенности ООП при разработке современных программных систем, и в частности систем ИИ.

ГЛОССАРИЙ

Fuzzy Logic — нечеткая логика.

NP-задачи — недетерминированные, полиномиальные, слабо формализованные или плохо формализуемые задачи, для которых не существует точного решения, либо оно не может быть найдено за конечное количество шагов. Задачи этого типа возникают в условиях неполной, нечеткой, размытой, недостоверной или некорректной информации.

Агенда — список активных правил.

База знаний — вместилище знаний, необходимых для получения решений, сформированных на основе одной или нескольких моделей представления знаний и используемых в цепочках логического вывода систем ИИ; БЗ можно описать как совокупность правил логического вывода, неизменных в процессе отдельного эксперимента и представленных в установленных для системы формализмах (фреймы, семантические сети, предикаты первого порядка).

Блок логического вывода — элемент механизма использования правил для получения решений в архитектуре экспертной системы.

БПР (бизнес-процесс реинжиниринг) — исследование деятельности компании в целях выявления ее слабых мест и моделирование на основе этого ее функционирования с максимальной эффективностью.

Генотип — условный код в генетических алгоритмах для задания исходных данных решаемой задачи, которым она будет решена.

Гибридные системы — системы, обеспечивающие возможность комбинированного использования символического и субсимволического подходов.

Данные — описания и значения свойств объектов, представленных в заданной форме; в частности, данные — это значения констант и переменных.

Дедукция — метод получения новых знаний путем анализа понятий, способ логического вывода от общего к частному.

Декларативные знания — знания, сформулированные в виде фактов, правил и отношений.

Диагностика — процесс поиска неисправностей в системе или определения стадии заболевания в медицине, основанный на интерпретации данных; диагностика используется для установления связи между нарушениями деятельности в системе или организме и их возможными причинами в целях их устранения; основными проблемами, воз-

никающими при этом, являются: недоступность или малодоступность некоторых данных; сочетание не вполне совместимых частных моделей объектов или процессов; наложение симптомов других дефектов.

Знания — описания объектов, их характеристик в соответствии с фактами предметной области, представленные на естественном языке и отношениями каузальной (причинно-следственной) зависимости в целях констатации фактов или принятия решения.

Индукция — метод получения новых знаний путем анализа эмпирических фактов; способ логического вывода от частного к общему.

Интерпретация — анализ данных в целях определения их смысла; интерпретатор должен быть в состоянии обрабатывать информацию, представленную частично, выдвигать гипотезы о доверии данным; при ненадежных данных интерпретация также будет ненадежной, поэтому для достижения доверия необходимо определить, какая информация была неточной или неопределенной; так как цепочки рассуждений в ИнС могут быть достаточно длинными, интерпретатору необходимо располагать средствами объяснения того, как интерпретация обусловлена имеющимися данными.

Когнитолог — инженер по знаниям, специалист по инженерии знаний, работающий с экспертом, выявляя и формализуя экспертные знания.

Концептуализация — описание объектов, свойств, рассуждений.

Лингвистическая модель — модель, интерпретирующая информацию, получаемую от акустической модели, и отвечающая за представление результата распознавания потребителю.

Метаправила — правила работы с правилами.

Механизмы генетических алгоритмов — селекция (отбор), кроссинговер (скрещивание) и мутация (случайное изменение).

Мониторинг — непрерывное оповещение о состоянии системы или процесса при возникновении ситуаций, требующих вмешательства, основанное на интерпретации сигналов.

Мощность правила — апостериорная характеристика, определяемая как произведение коэффициента уверенности факта в условии и коэффициента уверенности самого правила.

Неопределенность — степень соответствия процесса или состояния характеристикам реального мира.

Обучение — обучение пользователя, а также самообучение системы как на этапе приобретения знаний, так и в процессе работы ИП.

Означивание — свойство знаний, характеризующее задержанный процесс, ассоциирующий организацию логического вывода.

Онтология — формальное описание понятий предметной области и отношений между ними в рассматриваемой предметной области, свойств каждого понятия, описывающих различные атрибуты понятия, а также ограничений, наложенных на слоты; онтология вместе с набором индивидуальных экземпляров классов образует базу знаний; в центре большинства онтологий находятся классы; классы описывают понятия предметной области.

Организация логического вывода — порядок интерпретации фактов из рабочей области в правилах в интересах формирования цепочки логических рассуждений.

Особь (хромосома) — заданная последовательность единиц в коде.

Отношение — наличие семантической зависимости между конкретными значениями данных с параметрами процедуры.

Отношения — совокупность семантических отношений для интерпретации правил в стратегиях поиска решений.

Персептрон — однослойная нейронная сеть, состоящая из нейронов с пороговой функцией активации.

Планирование — процесс составления планов; планировщик должен уметь делать пробные шаги и исследовать возможные планы, уметь сосредотачивать внимание на наиболее важных гипотезах, работать в условиях неопределенности; планирование должно быть условным, зависящим от поступления новых сведений.

Поколение — совокупность особей при реализации генетического алгоритма.

Правило — один из главных компонентов знания, определяющий стратегию поиска решения и имеющий форму представления в виде причинно-следственного отношения между фактами предметной области и получаемыми решениями в цепочках логического вывода.

Правила вывода — сложные предложения, определяющие новые правила и факты и предназначенные для создания новых стратегий с использованием новых фактов.

Предикат — это свойство объекта (в узком смысле) и описание отношений между свойствами объекта, которые могут быть определены или не определены (в широком смысле); свойства предиката описываются совокупностью его аргументов.

Предложение — совокупность правил в программе, соединенных логическими операторами and («и»), or («или»), not («не»).

Проблемная информация — информация, представляющая описание решаемой задачи, включает описание фактов и правил, является основой проектирования БЗ.

Прогнозирование — предсказание хода событий в будущем на основании модели прошлого и настоящего; ключевыми проблемами задачи является требование соединения в единое целое неполной имеющейся информации; прогнозирование рассматривает различные варианты будущего и указывает на их чувствительность к изменению входных данных; решаемая задача прогнозирования должна носить условный характер, поскольку вероятность определенных событий в будущем будет зависеть от более близких, но не предсказуемых событий.

Продукция — правило вывода.

Проектирование — использование экспертной системы для исключения профессионала из задачи проектирования или выполнение рутинных действий по обработке информации в конкретной прикладной системе; ключевыми проблемами проектирования являются: отсутствие исчерпывающей информации, позволяющей увязать ограничения проектирования с принимаемыми решениями; взаимодействие подзадач (одна оказывает влияние на другую); умение видеть картину в целом, чтобы уходить из точек в пространстве проекта, которые являются лишь локально-оптимальными; оценивание последствий принимаемых решений.

Процедурные знания — знания, формализованные в виде правил и процедур.

Свойства объектно-ориентированного подхода — абстракция данных, инкапсуляция, полиморфизм и наследование.

Семантическая сеть — в графическом виде представляет собой орграф, вершинами которого являются предложения или понятия естественного языка, отношения между ними, а дуги представляют степень влияния одной выделенной сущности на другую; дуга с инцидентными ей узлами является семантическим представлением правила.

Система-прототип — работающая модель программы, функционально эквивалентная подмножеству конечного продукта.

Три подхода к распознаванию символов — шаблонный, структурный и признаковый.

Унификация — операция приведения к единому значению в исчислении предикатов первого порядка.

Управляющая структура — механизм приложения правил в заданной модели в целях получения решения.

Утверждение — совокупность предложений, не связанных логическими операторами and («и»), or («или»), not («не»).

Факты — значения данных или описание понятий предметной области, предназначенные для интерпретации в правилах логического вывода.

Фенотип — определение экземпляра кода; каждый код представляет собой точку пространства поиска.

Формализация — представление информации в заданных структурах.

Фрейм — структура данных для представления стереотипных ситуаций.

Эксперт — человек, который является профессионалом высокой квалификации в данной проблемной области, для которой предназначена разработка экспертной системы; его знания лежат в основе системы.

Экспертная система — класс компьютерных систем, ориентированных на тиражирование опыта высококвалифицированных специалистов — экспертов в таких слабоформализованных областях, где качество принятия решений традиционно зависит от уровня экспертизы (например, медицина, экономика, юриспруденция, геология, военное дело и т.д.); это система, основанная на знаниях о заданной проблемной области, в которой знания слабо структурированы, в которой решаются сложные задачи (из области NP), осуществляется взаимодействие с естественным языком на основе рассуждений и комментирования своих действий в целях обучения пользователя при самообучении системы.

Элитизм — операция, при выполнении которой скрещиваемые хромосомы-родители или хромосомы предыдущего поколения переходят в следующее поколение без изменений.

СПИСОК ЛИТЕРАТУРЫ

1. *Адаменко А.Н., Кучуков А.М.* Логическое программирование и Visual Prolog. СПб. : БХВ-Петербург, 2003.
2. Базы данных. Интеллектуальная обработка информации / В.В. Корнеев, А.Ф. Гараев, С.В. Васютин, В.В. Райх. М. : Нолидж, 2001.
3. *Ветлугин А.Г., Сидоркина И.Г.* Логическое программирование. Йошкар-Ола : Изд-во МарГТУ, 1997.
4. *Гаврилова Т.А., Хорошевский В.Ф.* Базы знаний интеллектуальных систем. СПб. : Питер, 2000.
5. *Комарова Л.Г., Максимов А.В.* Нейрокомпьютеры : учеб. пособие для вузов. М. : Изд-во МГТУ им. Н.Э. Баумана, 2002.
6. *Круглов В.В., Борисов В.В.* Искусственные нейронные сети. Теория и практика. 2-е изд., стереотип. М. : Горячая линия — Телеком, 2002.
7. *Малпас Дж.* Реляционный язык Пролог и его применение. М. : Наука, 1990.
8. *Минский М.* Структура для представления знания. Психология машинного зрения / под ред. П. Уинстона. М. : Мир, 1978.
9. *Минский М.* Фреймы для представления знаний. М. : Энергия, 1979.
10. *Нильсон Н.* Принципы искусственного интеллекта : пер. с англ. М. : Радио и связь, 1985.
11. *Осовский С.* Нейронные сети для обработки информации / пер. с польского И.Д. Рудинского. М. : Финансы и статистика, 2004.
12. *Попов Э.В.* Экспертные системы. Решение неформализованных задач в диалоге с ЭВМ. М. : Наука, 1987.
13. *Розенблатт Ф.* Принципы нейродинамики. М. : Мир, 1965.
14. Системы искусственного интеллекта. Практический курс. М. : Бином, 2008.
15. Справочник по искусственному интеллекту : в 3 т. / под ред. Э.В. Попова, Д.А. Поспелова. М. : Радио и связь, 1990.
16. Статические и динамические экспертные системы / Э.В. Попов, И.Б. Фоминых, Е.Б. Кисель, М.Д. Шапот. М. : Финансы и статистика, 1996.
17. Труды конференций Интеллектуальные системы, Интеллектуальные САПР : в 3 т. М. : Физматлит, 2008.
18. *Элти Дж., Кумбс М.Э.* Экспертные системы. Концепции и примеры : пер. с англ. М. : Финансы и статистика, 1987.
19. *Barr A., Feigenbaum E.A.* The Handbook of Artificial Intelligence : Vol. 1. Los Altos CA : Morgan Kaufmann, 1981.

20. *Goldberg D.E.* Genetic algorithms in search, optimization and machine learning. Adison Wesley, Reading, MA, 1989.
21. *Holland J.H.* Adaptation in natural and artificial systems. University of Michigan, 1975.
22. http://codingrus.ru/readarticle.php? article_id=1750 — Унификация списков как аргументов предикатов.
23. http://kaon.semanticWeb.org/ — The KArlsruhe ONtology and Semantic Web tool suite.
24. http://protege.stanford.edu/doc/ — The Protégé Ontology Editor and Knowledge Acquisition System.
25. http://window.edu.ru/window_catalog/files/r26171/577.pdf — Николаев В.Г., Евсеева О.Н., Ярушкина Н.Г. Программирование на языке Турбо-Пролог 2.0.
26. http://www.basegroup.ru/library/analysis/fuzzylogic/fuzzy rule/ — Математические основы нечеткой логики.
27. http://www.matlab.ru/fuzzylogic/book1/index.asp — Штовба С.Д. Введение в теорию нечетких множеств и нечеткую логику.
28. http://www.swi-prolog.org/dl-stable.html — SWI-Prolog's home.
29. http://www.w^3.org/TR/rdf-syntax-grammar — RDF/XML Syntax Specification (Revised).
30. *Lenat Douglas.* EURISKO : A program that learns new heuristics and domain concepts. Artificial Intelligence, 1983.
31. *McCalloch W. S., Pitss W.H.* A logical calculus of ideas immanent in nervous activity // Bull. Math. Biophysics, 1943.
32. *Newell A.* The knowledge level. Artificial Intelligence, 1982.
33. *Newell A., Simon H.A.* Human Problem Solving. Englewood-Cliffs, NJ : Prentice-Hall, 1972.
34. *Schach S.R.* Software Engineering. Richard D. Irwin, 1993.