ВОЕННО-КОСМИЧЕСКАЯ АКАДЕМИЯ
имени А.Ф. Можайского

СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Учебное пособие

Под общей редакцией доктора технических наук,
профессора **А.Д. Хомоненко**

Санкт-Петербург
2013

Авторы:
А.Н. Адаменко, С.В. Войцеховский, Р.И. Компаниец, А.М. Кучуков, М.В. Полищук, А.Д. Хомоненко

Рецензенты:
доктор технических наук, профессор **В.А. Ходаковский**;
доктор технических наук, доцент **А.Г. Басыров**

Системы искусственного интеллекта: учебное пособие / А.Н. Адаменко, С.В. Войцеховский и др.; под общ. ред. профессора А.Д. Хомоненко. – СПб.: ВКА имени А.Ф. Можайского, 2013. – 291 с.

Учебное пособие предназначено для курсантов и слушателей, проходящих обучение по специальности «Применение и эксплуатация автоматизированных систем специального назначения» для специализации «Программное и математическое обеспечение систем управления летательными аппаратами», а также для обучающихся по другим специальностям, предусматривающим изучение дисциплины «Системы искусственного интеллекта».

Подписано к печ. 24.12.2012 Формат печатного листа 445Х300/8
Гарнитура Times New Roman Авт. печ. л. 18,25
Уч.-печ. л. 37 Заказ 2455 Бесплатно

Типография ВКА имени А.Ф. Можайского

ОГЛАВЛЕНИЕ

СПИСОК СОКРАЩЕНИЙ

XML (Extensible Markup Language) – расширяемый язык разметки документов;

АС – автоматизированная система;
АСУ – автоматизированная система управления;
БЗ – база знаний;
ИИ – искусственный интеллект;
ИППП – интеллектуальный пакет прикладных программ;
ИС – инструментальное средство;
ЛП – лингвистическая переменная;
ЛТ – лингвистический терм;
ПО – предметная область;
ПОМ – предположение об открытости мира;
СИИ – система искусственного интеллекта;
СПЗ – система представления знаний;
СППР – система поддержки принятия решения;
СС – семантическая сеть;
ФП – функция принадлежности;
ЭВМ – электронно-вычислительная машина;
ЭС – экспертная система;
ЯПЗ – язык представления знаний.

ВВЕДЕНИЕ

Системы искусственного интеллекта представляют собой одну из интенсивно развивающихся отраслей знаний современного общества и важнейших составляющих в модели подготовки специалиста в области информационных технологий. В настоящем пособии рассматривается достаточно типичный набор вопросов, как правило, входящих в состав примерных программ по дисциплине «Системы искусственного интеллекта». Прежде всего, это основные понятия искусственного интеллекта и характеристика инструментальных средств, используемых для разработки систем, основанных на знаниях.

Во-вторых, это проблемы и модели представления знаний. В пособии дается краткая характеристика основных моделей представления знаний: семантических сетей, фреймов и продукционных систем. Рассматриваются кратко также введение в логику и логические системы. Кроме того, в пособии приводится сравнительно нетрадиционный материал по представлению знаний, касающийся дескриптивной логики и онтологий. При этом приводятся общая характеристика, синтаксис и семантика дескриптивных логик, описываются онтологии и языки представления онтологий.

В-третьих, в пособии рассматриваются экспертные системы, как один из важнейших представителей систем, основанных на знаниях. При этом освещаются: структура экспертной системы, подсистема логического вывода, стратегии управления выводом (прямой и обратный вывод, повышение эффективности поиска решений). Здесь же затрагиваются вопросы проектирования и разработки экспертных систем (ЭС): проблемы разработки экспертных систем, выбор проблемы, технология быстрого прототипирования, развитие прототипа допромышленной ЭС, оценка, стыковка и поддержка экспертной системы.

В-четвертых, отдельные два раздела в пособия посвящены рассмотрению основ теории нечетких множеств (определение нечетких множеств и функций принадлежности, операции над нечеткими множествами, нечеткие и лингвистические переменные, нечеткие лингвистические высказывания логические операции с нечеткими высказываниями) и продукционным системам с нечеткими знаниями (этапы нечеткого вывода, правила нечетких продукций, введение нечеткости, агрегирование подусловий и активизация подзаключений, аккумулирование заключений, приведение результатов к четкости, алгоритмы нечеткого вывода).

При этом кратко рассматриваются отличительные особенности алгоритмов Мамдани, Ларсена, Цукамото и Такаги-Сугено. Здесь подраздел, посвященный лингвистической модификации функции принадлежности (задача уточнения

нечеткого вывода, оценивание степени размытости функции принадлежности, лингвистические модификаторы, влияние коэффициента лингвистической модификации функции принадлежности на степень уверенности СППР в наличии программных воздействий в АСУ), содержит результаты исследований авторов.

В разделе 7, посвященном искусственным нейронным сетям, кратко рассматриваются основные понятия нейронных сетей, модель нейрона, обобщенная модель искусственной нейронной сети, классификация искусственных нейронных сетей, характеристика процесса обучения нейронных сетей.

Для обеспечения практических занятий по дисциплине «системы искусственного интеллекта» в пособии дается достаточно подробное описание ряда инструментальных систем, в том числе: работа в системе логического программирования Visual Prolog, построение нечетких продукционных систем в системе MatLab, правила и объекты в системе Leonardo. Соответствующий материал сопровождается небольшими, наглядными примерами, иллюстрирующими технологию разработки систем основанных на знаниях.

При подготовке пособия использованы материалы, подготовленные с участием И.А. Белякова (раздел 7) и В.В. Рогальчука (подраздел 3.4).

Часть I ТЕОРЕТИЧЕСКИЕ ОСНОВЫ СИСТЕМ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

1 СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

1.1 Понятия искусственного интеллекта

Искусственный интеллект (ИИ) – это научная дисциплина, которая возникла в 50-х годах XX века на стыке кибернетики, лингвистики, психологии и программирования.

С самого начала исследования в области ИИ пошли по двум направлениям:

Первое (бионическое) – попытки смоделировать с помощью искусственных систем психофизиологическую деятельность человеческого мозга с целью создания искусственного разума.

Второе (прагматическое) – создание программ, позволяющих с использованием ЭВМ воспроизводить не саму мыслительную деятельность, а являющиеся ее результатами процессы. Здесь достигнуты важные результаты, имеющие практическую ценность. В дальнейшем речь будет идти об этом направлении.

Разработка интеллектуальных программ существенно отличается от обычного программирования и ведется путем построения системы искусственного интеллекта (СИИ). Если обычная программа может быть представлена в парадигме:

Программа = Алгоритм + Данные,

то для СИИ характерна другая парадигма:

СИИ = Знания + Стратегия обработки знаний.

Основным отличительным признаком СИИ является работа со знаниями. Если для обычных программ представление данных определяется на уровне описания языка программирования, то для СИИ представление знаний выливается в проблему, связанную со многими вопросами: что такое знания, какие знания хранить в системе в виде *базы знаний* (БЗ), в каком виде и сколько, как их использовать, пополнять и т. д.

В отличие от данных, знания обладают следующими свойствами:

- *внутренней интерпретируемостью* – вместе с информацией в БЗ представлены информационные структуры, позволяющие не только хранить знания, но и использовать их;
- *структурированностью* – выполняется декомпозиция сложных объектов на более простые и установление связей между ними;

- *связанностью* – отражаются закономерности относительно фактов, процессов, явлений и причинно-следственные отношения между ними;
- *активностью* – знания предполагают целенаправленное использование информации, способность управлять информационными процессами по решению определенных задач.

Все эти свойства знаний в конечном итоге должны обеспечить возможность СИИ моделировать рассуждения человека при решении прикладных задач – со знаниями тесно связано понятие процедуры получения решений задач (стратегии обработки знаний). В системах обработки знаний такую процедуру называют *механизмом вывода, логическим выводом* или *машиной вывода*. Принципы построения механизма вывода в СИИ определяются способом представления знаний и видом моделируемых рассуждений.

Для организации взаимодействия с СИИ в ней должны быть средства общения с пользователем, т. е. *интерфейс*. Интерфейс обеспечивает работу с БЗ и механизмом вывода на языке достаточно высокого уровня, приближенном к профессиональному языку специалистов прикладной области СИИ. Кроме того, в функции интерфейса входит поддержка диалога пользователя с системой, что дает возможность пользователю получать *объяснения* действий системы, участвовать в поиске решения задачи, пополнять и корректировать базу знаний.

Таким образом, основными частями систем, основанных на знаниях, являются:

1. База знаний.
2. Механизм вывода.
3. Интерфейс с пользователем.

Каждая из этих частей может быть устроена по-разному в различных системах, отличия эти могут быть в деталях и в принципах. Однако для всех СИИ характерно *моделирование человеческих рассуждений*. СИИ создаются для того, чтобы овеществлять в рамках программно-технической системы знания и умения, которыми обладают люди, чтобы решать задачи, относящиеся к творческой деятельности человека.

Знания, на которые опирается человек, решая ту или иную задачу, существенно разнородны, это, прежде всего:

- понятийные знания (набор понятий и их взаимосвязи);
- конструктивные знания (знания о структуре и взаимодействии частей различных объектов);
- процедурные знания (методы, алгоритмы и программы решения различных задач);

- фактографические знания (количественные и качественные характеристики объектов, явлений и их элементов).

Особенность систем представления знаний заключается в том, что они моделируют деятельность человека, осуществляемую часто в неформальном виде. Модели представления знаний имеют дело с информацией, получаемой от экспертов, которая часто носит качественный и противоречивый характер. Для обработки с помощью ЭВМ такая информация должна быть приведена к однозначному формализованному виду. Методологией формализованного представления знаний является логика.

Отличительной особенностью экспертных систем является их ориентация на решение *неформализованных* задач, обладающих одной или несколькими следующими характеристиками:

- невозможность задания в числовой форме;
- цели нельзя выразить в терминах точно определенной целевой функции;
- отсутствие алгоритмического решения задачи;
- невозможность использования существующего алгоритмического решения из-за ограниченности аппаратных ресурсов.

К числу особенностей неформализованных задач можно отнести также следующее:

- ошибочность, неоднозначность, неполнота и противоречивость исходных данных;
- ошибочность, неоднозначность, неполнота и противоречивость знаний о проблемной области и решаемой задаче;
- большая размерность пространства решения;
- динамически изменяющиеся данные и знания.

Экспертные системы и системы искусственного интеллекта, в отличие от систем обработки данных, используют в основном *символьное*, а не числовое, представление данных, символьный вывод и эвристический поиск решения, а не реализация известного алгоритма.

Решения, получаемые с помощью ЭС, отличаются «*прозрачностью*» в том смысле, что могут быть объяснены пользователю на качественном уровне. Это обусловлено способностью ЭС рассуждать о своих знаниях и умозаключениях.

Специалистами по системам программирования в последние несколько лет высказывались мнения, что применение экспертных систем и систем искусственного интеллекта не дало ожидаемого эффекта и их ждет постепенное отмирание. Отправной точкой таких суждений являлось рассмотрение ЭС в качестве альтернативы традиционному программированию и предположение, что ЭС полностью решают поставленные задачи в изоляции от других программных средств. Действительно, на раннем этапе развития ЭС

особенности используемых в них языков программирования, технологии разработки программ и аппаратных средств позволяли предположить высокую сложность интеграции их с традиционными программными системами.

В настоящее время инструментальные средства разработки ЭС создаются в соответствии с современными достижениями технологии программирования. При этом технология ЭС находит свое применение при разработке интегрированных приложений во многих областях.

Рассматривая нынешнее состояние СИИ и ЭС, многие авторы отмечают коммерческие успехи их применения. В качестве причин такого успеха современных инструментальных средств искусственного интеллекта (ИС ИИ) указывают следующие их свойства:

- *интегрированность*, означающая простоту использования ИС ИИ совместно с другими информационными технологиями и средствами;
- *открытость и переносимость*, означающую соответствие стандартам, обеспечивающих открытость и переносимость;
- *использование языков традиционного программирования и рабочих станций*. Первое позволило упростить обеспечение интегрированности, снизить требования приложений ИИ к аппаратным ресурсам по быстродействию и объему основной памяти. Второе существенно увеличило круг приложений, выполняемых с использованием ИС ИИ;
- *проблемно/предметно-ориентированные* ***ИС ИИ***. Переход к таким системам обеспечивает: сокращение сроков разработки приложений, увеличение эффективности использования ИС, упрощение и ускорение работы эксперта, повторную используемость информационного и программного обеспечения (объекты, классы, правила, процедуры).

Для примера укажем некоторые предметные области, в которых технологии ЭС получили успешное коммерческое применение:

- определение конфигурации вычислительной системы VAX (ЭС XCON/XSEL);
- управление трубопроводом (ИС G2 фирмы Gensym);
- выявление и устранение неисправностей в нефтехимической промышленности (ИС G2 фирмы Gensym);
- моделирование страховых рисков (ИС G2 фирмы Gensym).

Отметим зарубежных и советских *ученых*, внесших заметный вклад в развитие и применение теории искусственного интеллекта. Основания названной теории связаны с работами Аристотеля, А.М. Тьюринга, Е. Поста и К. Геделя. Заметный вклад в развитие и применение теории СИИ внесли Дж. Робинсон (метод резолюций), А. Ньюэлл, М. Минский, Дж. Слэйгл,

Е. Файгенбаум и др. В числе советских ученых можно назвать Г.С. Поспелова, Д.А. Поспелова, Э.В. Попова и С.Ю. Маслова.

1.2 Направления работ и инструментарий искусственного интеллекта

В настоящее время исследования в области ИИ имеют следующую прикладную ***ориентацию***:

- общение на естественном языке и моделирование диалога;
- экспертные системы (ЭС);
- автоматическое доказательство теорем ;
- робототехника;
- интеллектуальные пакеты прикладных программ;
- распознавание образов;
- решение комбинаторных задач.

Наибольшие практические результаты достигнуты в создании ЭС и в сфере робототехники, которые получили уже широкое распространение и используются при решении практических задач.

Экспертная система представляет собой программный комплекс, содержащий знания специалистов из определенной предметной области, обеспечивающий консультациями менее квалифицированных пользователей для принятия экспертных решений. Основное отличие ЭС от обычных программ, также способных поддерживать экспертные решения, заключается в отделении декларативных знаний от манипулирующего знаниями процедурного компонента.

Интеллектуальный пакет прикладных программ (ИППП) представляет собой интегрированную систему, позволяющую пользователю решать задачи без программирования – путем описания задачи и исходных данных. Программирование осуществляется автоматически программой планировщиком из набора готовых программных модулей, относящихся к конкретной предметной области. В числе примеров ИППП можно назвать систему ПРИЗ, в которой пользователь формирует свою задачу на непроцедурном языке УТОПИСТ. Еще одним примером ИППП является система СПОРА, в которой формирование задачи пользователь выполняет на непроцедурном языке ДЕКАРТ.

К числу ИППП относятся решатели вычислительных задач. В качестве примера назовем решатель вычислительных задач TKSolver, с помощью

которого можно описывать и решать задачи вычислительного характера без программирования.

Инструментальные средства, используемые для разработки СИИ, в том числе ЭС, можно разделить на следующие типы:

- системы программирования на языках высокого уровня;
- системы программирования на языках представления знаний;
- средства автоматизированного создания ЭС;
- оболочки систем искусственного интеллекта – скелетные системы.

Системы программирования на языках высокого уровня (ЯВУ) таких, как C++, Паскаль, Фортран, Бэйсик, Forth, Refal, SmallTalk, Лисп и др., в наименьшей степени ориентированы на решение задач искусственного интеллекта. Они не содержат средств, предназначенных для представления и обработки знаний. Тем не менее, достаточно большая, но со временем снижающаяся, доля СИИ разрабатывается с помощью традиционных ЯВУ.

В приведенном перечне можно выделить языки Лисп и SmallTalk, как наиболее удобные и широко используемые для создания СИИ. В частности, широкое использование языка Лисп объясняется наличием развитых средств работы со списками и поддержкой механизма рекурсии, важных для характерной в СИИ обработки символьной информации. С помощью языка Лисп разработан ряд распространенных ЭС, таких как MYCIN (решение задач диагностики в медицине), DENDRAL (распознавание формул химических элементов), PROSPECTOR (поиск полезных ископаемых).

Системы программирования на языках представления знаний (ЯПЗ) имеют специальные средства, предназначенные для создания СИИ. Они содержат собственные средства представления знаний (в соответствии с определенной моделью) и поддержки логического вывода. К числу языков представления знаний можно отнести FRL (Frame Representation Language – язык представления фреймов), KRL (Knowledge Representation Language – язык представления знаний), OPS5 (Official Production System), Log Lisp, Пролог и др. Разработка СИИ с помощью систем программирования на ЯПЗ основана на технологии обычного программирования. От разработчика требуются соответствующие программистские навыки и квалификация. Наибольшее распространение из числа названных языков получили язык логического программирования Пролог и OPS5.

Средства автоматизированного создания ЭС представляют собой гибкие программные системы, допускающие использование нескольких моделей представления знаний, способов логического вывода и видов интерфейса, и содержащие вспомогательные средства создания ЭС. В качестве примеров рассматриваемого класса средств можно назвать следующие системы: EXSYS

(для создания ЭС классификационного типа), 1st-Class, Personal Consultant Plus, ПИЭС (программный инструментарий экспертных систем), GURU (интегрированная среда разработки ЭС), XiPlus, OPS5+. Построение ЭС с помощью рассматриваемых средств заключается в формализации исходных знаний, записи их на входном языке представления знаний и описании правил логического вывода решений. Далее экспертная система наполняется знаниями.

К рассматриваемому классу систем можно отнести также ***специальный программный инструментарий***. К примеру, сюда относятся библиотеки и надстройки над языком Лисп: KEE (Knowledge Engineering Environment – среда инженерии знаний), FRL, KRL и др. Они повышают возможности и гибкость в работе с заготовками экспертных систем.

Оболочки, или «пустые» экспертные системы представляют собой готовые экспертные системы без базы знаний. Примерами оболочек ЭС, получивших широкое применение, являются зарубежная оболочка EMYCIN (Empty MYCIN – пустой MYCIN) и отечественная оболочка Эксперт-микро, ориентированная на создание ЭС решения задач диагностики. Технология создания и использования оболочки ЭС заключается в том, что из готовой экспертной системы удаляются знания из базы знаний, затем база заполняется знаниями, ориентированными на другие приложения. Достоинством оболочек является простота применения – специалисту нужно только заполнить оболочку знаниями, не занимаясь созданием программ. Недостатком применения оболочек является возможное несоответствие конкретной оболочки разрабатываемой с ее помощью прикладной ЭС.

2 ЭКСПЕРТНЫЕ СИСТЕМЫ

Как отмечалось, **экспертная система** представляет собой программный комплекс, содержащий знания специалистов из определенной предметной области, обеспечивающий консультациями менее квалифицированных пользователей для принятия экспертных решений.

2.1 Структура экспертной системы

Структура экспертной системы зависит от ее назначения и решаемых задач. В состав экспертных систем (рис. 2.1) входят следующие *основные*

компоненты: база знаний, решатель, редактор базы знаний, подсистема объяснений и интерфейс пользователя.

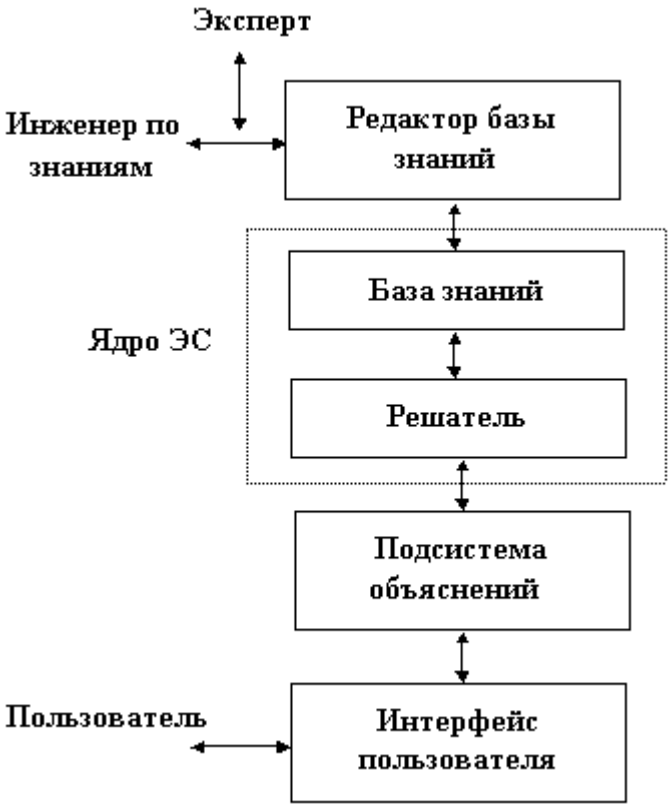


Рис. 2.1 Структура экспертной системы

Определение и взаимодействие компонентов ЭС может быть описано следующим образом.

База знаний представляет собой совокупность знаний о предметной области, организованных в соответствии с принятой моделью представления знаний.

Решатель, или подсистема логического вывода представляет собой программу, обеспечивающую автоматический вывод решения формулируемых пользователем или экспертом задач на основе знаний, хранящихся в базе.

База знаний и решатель вместе составляют основную часть – ***ядро ЭС***. В ряде источников к ядру ЭС относят только базу знаний.

Инженер по знаниям – специалист по искусственному интеллекту, помогающий эксперту вводить знания в базу знаний.

Эксперт – специалист в предметной области, способный принимать экспертные решения и формулирующий знания о предметной области для ввода их в базу знаний.

Редактор базы знаний – это программа, предназначенная для ввода в базу новых знаний о предметной области.

Пользователь ЭС является специалистом в данной предметной области, квалификация которого уступает квалификации эксперта.

Интерфейс пользователя есть комплекс программ, обеспечивающих удобный диалог с ЭС при вводе запросов на решение экспертных задач и получении результатов.

Подсистема объяснений – это программа, позволяющая пользователю трассировку логического вывода и получить мотивировку умозаключений на каждом этапе цепочки вывода.

Конкретная экспертная система создается в результате совместной работы инженера по знаниям и эксперта. Взаимодействие пользователя с ЭС осуществляется через интерфейс на близком к естественному или профессиональному языку предметной области непроцедурном языке. При этом производится трансляция предложений на язык представления знаний (ЯПЗ) экспертной системы. Описание запроса на ЯПЗ поступает в решатель, в котором на основе знаний из базы выводится решение поставленного запроса в соответствии с некоторой стратегией выбора правил. С помощью подсистемы объяснений отображают промежуточные и окончательные выводы, объяснение применяемой мотивировки.

Приведенную на рис. 2.1 структуру, согласно [23], называют структурой ***статической*** экспертной системы. Такие ЭС применяются в приложениях, в которых можно не учитывать изменения окружающей среды в процессе решения задачи. Получившие первое практическое использование экспертные системы относились к системам именно такого типа.

Существует большой класс приложений, в которых требуется учитывать изменения в окружающей среде за время исполнения приложения. Соответствующие экспертные системы называют динамическими. В состав динамических ЭС входят две дополнительные подсистемы: моделирования внешнего мира и сопряжения с внешним миром. Вторая подсистема обеспечивает связь с внешним миром через систему датчиков и контроллеров. Более того, входящие в состав обычной статической ЭС компоненты (база знаний и машина вывода) существенно изменены для отражения временной логики происходящих в мире событий.

2.2 Подсистема логического вывода

В составе получивших широкое распространение продукционных ЭС решатель, или подсистема логического вывода, выполняет следующие функции: во-первых, просмотр существующих фактов в рабочей памяти и правил из базы знаний и добавление в рабочую память новых фактов;

во-вторых, определение порядка просмотра и применения правил; в-третьих, управление процессом консультации, сохраняя для пользователя информацию о полученных заключениях, и запрашивает у него информацию, когда для срабатывания очередного правила в рабочей памяти не хватает данных.

В некоторых ЭС используется прямой порядок вывода – от фактов, находящихся в рабочей памяти, к заключению. В других системах вывод осуществляется в обратном порядке: заключения просматриваются последовательно до тех пор, пока не будут обнаружены в рабочей памяти или получены от пользователя факты, подтверждающие одно из них. Обычно решатель представляет собой небольшую программу. Основной объем памяти занимают правила.

Напомним, что решатель включает в свой состав два компонента: компонент вывода и управляющий компонент. Компонент вывода решает задачу, просматривая имеющиеся правила и факты из рабочей памяти и добавляя в нее новые факты при срабатывании какого-либо правила. Управляющий компонент определяет порядок применения правил.

Компонент вывода. Применительно к продукционным ЭС действие этого компонента основано на применении правила вывода, именуемого «модус поненс», суть которого состоит в следующем: пусть известно, что истинно утверждение А и имеется правило вида «ЕСЛИ А, ТО В», тогда утверждение В также истинно. Правила срабатывают, когда находятся факты, удовлетворяющие их левой части: если истинна посылка, то должно быть истинно и заключение.

Реализация такого с виду простого правила в ЭС может оказаться проблематичной в ситуациях, не представляющих трудностей для человека. Например, рассмотрим предложение:

«Сергей изучал орган».

Для слова «орган» возможны два смысловых значения: музыкальный инструмент и часть организма.

Еще более сложно интерпретировать факты, являющиеся составными частями продукций, которые используют правила «модус поненс» для вывода заключений. Например, вывод заключения вида:

ЕСЛИ	В туманную погоду плохо видно
И	Сегодня туманная погода
ТО	Сегодня плохо видно

не составляет труда для человека, но проблематичен для экспертных систем.

В общем человек способен вывести большое число заключений с помощью большой базы знаний, хранимой в его памяти; с помощью экспертных систем можно вывести сравнительно небольшое число заключений при заданном множестве правил.

Компонент вывода должен быть способен функционировать при недостатке информации. К примеру, в системе диагностики неисправностей автомобиля некоторые факты состояния отдельных его узлов могут отсутствовать. При этом механизм вывода должен провести рассуждения и при недостатке информации и найти решение, пусть не обязательно точное.

Управляющий компонент. Определяет порядок применения правил и устанавливает наличие фактов, которые могут быть изменены в случае продолжения консультации.

Управляющий компонент реализует следующие функции:

- сопоставление образца правила с имеющимися фактами;
- выбор правила по заданному критерию (разрешение конфликта) в ситуациях, допускающих применение нескольких правил;
- срабатывание правила при совпадении образца правила с фактами из рабочей памяти;
- выполнение действия путем добавления в рабочую память заключения сработавшего правила, также выполнение действия, указание на которое содержится в правой части правила.

Схема работы интерпретатора продукций (рис. 2.2) является циклической и состоит в следующем. В каждом цикле интерпретатор просматривает все правила для выявления посылок, совпадающих с известными в данное время фактами из рабочей памяти. Интерпретатор определяет также порядок применения правил. После выбора правило срабатывает, его заключение помещается в рабочую память и цикл повторяется.

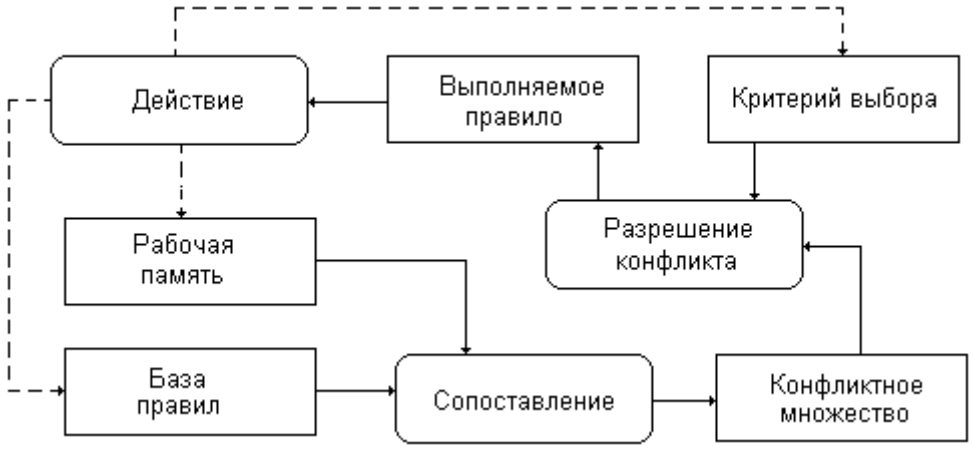


Рис. 2.2 Цикл работы интерпретатора продукций

В каждом цикле обрабатывается одно правило. Информация из рабочей памяти последовательно сопоставляется с посылками правил из базы для выявления успешного сопоставления. Отобранные таким образом правила образуют конфликтное множество.

Разрешение конфликта осуществляется на основе критерия выбора, с помощью которого выбирается одно выполняемое правило, которое срабатывает. При этом факты, образующие правило, помещаются в рабочую память или изменяется критерий выбора конфликтующих правил. Или выполняется действие, входящее в заключение правила.

2.3 Стратегии управления выводом

При проектировании управляющего компонента важным вопросом является определение *стратегии вывода*, т. е. выбор метода поиска решения. Процедура выбора заключается в определении направления поиска и способа его осуществления. Реализующие поиск процедуры обычно находятся внутри механизма вывода и поэтому в большинстве систем скрыты от инженеров знаний, а также недоступны для выбора и изменения.

При выборе стратегии управления выводом требуется решить два вопроса:

1. Определение исходной точки поиска в исходном пространстве состояний. От этого зависит метод осуществления поиска – в прямом или обратном направлении.

2. Повышение эффективности поиска решения. Для этого находят эвристики разрешения конфликтов, связанных с существованием нескольких возможных путей для продолжения поиска в пространстве состояний, чтобы повысить эффективность поиска, например, отбросить пути, не ведущие к искомому решению.

2.3.1 Прямой и обратный вывод

В системах с *прямым выводом* по известным фактам отыскивается заключение, которое из этих фактов следует (рис. 2.3). В случае отыскания такое заключение заносится в рабочую память. Прямой вывод называют также выводом, управляемым данными или управляемым антецедентами.

Например, пусть при движении автомобиля происходит снижение давления масла в двигателе. Требуется предсказать возможные последствия.

При возникновении определенного состояния (снижение давления масла) применяем соответствующие этой ситуации правила:

Правило1.

ЕСЛИ снизилось давление масла, **ТО** повысится трение.

Правило 2.

ЕСЛИ повысится трение, **ТО** двигатель заглохнет.

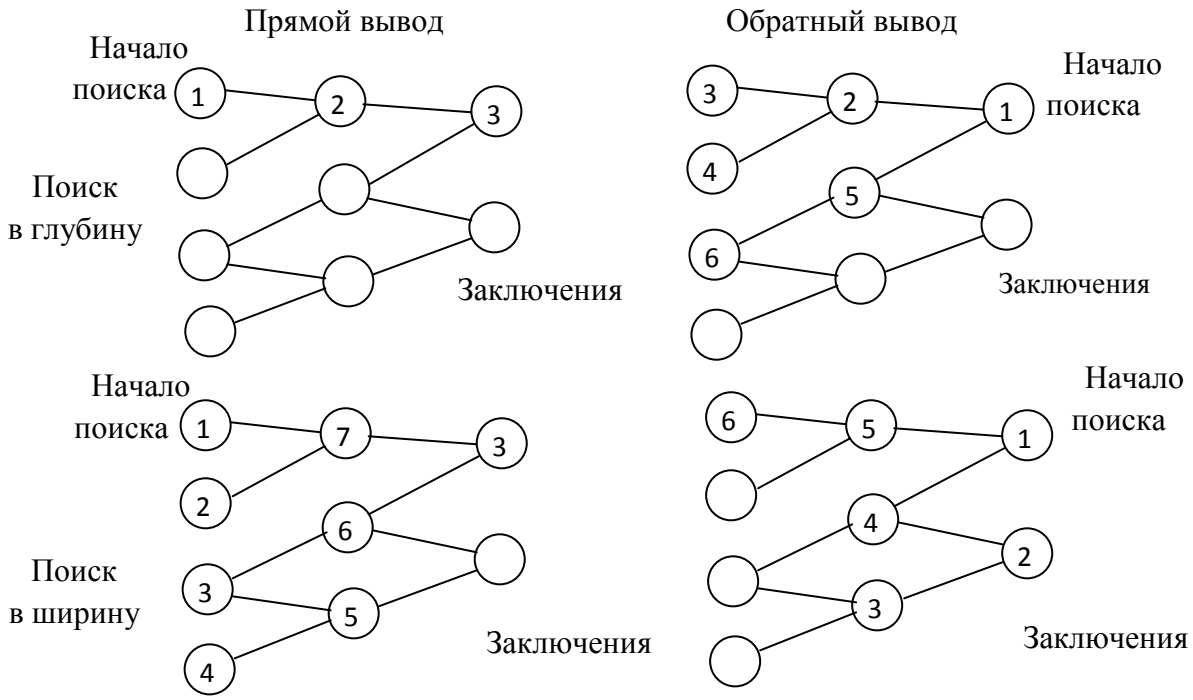


Рис. 2.3 Стратегии вывода решений

Сделать вывод о том, что двигатель заглохнет при недопустимом снижении давления масла можно, используя прямую цепочку рассуждений (прямой вывод). Здесь отправной точкой рассуждений служит возникшая ситуация (снижение давления масла). Сначала срабатывает первое правило, поэтому цепочка рассуждений продолжается. На основе срабатывания второго правила делаются выводы о возможных последствиях возникшей ситуации.

В системах с *обратным выводом* сначала выдвигается некоторая гипотеза, а затем механизм вывода от нее возвращается назад к фактам и пытается найти среди них те, которые подтверждают эту гипотезу. Если гипотеза оказалась правильной, то выбирается следующая гипотеза, детализирующая первую и являющаяся ее подцелью. Далее отыскиваются факты, подтверждающие истинность подчиненной гипотезы. Обратный вывод называют также выводом, *управляемым целями*, или *управляемым консеквентами*. Обратный вывод применяют в случаях, когда цели известны и их относительно немного.

Например, предположим, что двигатель не заводится. В более общей формулировке нам по известному результату требуется найти причины его возникновения.

Приведем возможные варианты правил, подходящих для рассматриваемого примера:

Правило 1. **ЕСЛИ** вышел из строя бензонасос, **ТО** неисправна подсистема питания.

Правило 2. **ЕСЛИ** неисправна подсистема питания, **ТО** двигатель не заводится.

Здесь для поиска причин того, что двигатель не заводится, сначала обращаемся к правилу 2, поскольку содержащийся в нем логический вывод «ТО двигатель не заводится» соответствует возникшей ситуации. Причина, по которой двигатель не заводится, содержится в условной части правила 2: «ЕСЛИ неисправна подсистема питания». Продолжая рассуждения, применяем правило 1. В условной части его записано: «ЕСЛИ вышел из строя бензонасос». Если это условие выполняется, то мы выявили причину, по которой двигатель не заводится. В противном случае нужно проверить другие правила, относящиеся к рассматриваемой ситуации.

Прямой вывод чаще применяется в системах планирования, обратный вывод более эффективен для применения в системах диагностики. В некоторых системах используется комбинированный метод, называемый *циклическим*, в котором вывод основан на сочетании названных методов – обратного и ограниченного прямого.

2.3.2 Повышение эффективности поиска

В интеллектуальных системах с очень большим количеством правил в базе знаний в желательно использование какой-либо стратегии управления выводом для минимизации времени поиска и соответственно повышения эффективности поиска решения. К таким *стратегиям* относятся:

- поиск в глубину;
- поиск в ширину;
- разбиение на подзадачи;
- альфа-бета алгоритм.

Применительно к обратному выводу поиск *в глубину* состоит в том, что при выборе очередной подцели в пространстве состояний предпочтение по возможности отдается подцели, которая соответствует следующему более детальному уровню описания задачи. Например, система диагностики состояния здоровья, сделав на основе имеющихся признаков предположение о болезни, будет запрашивать уточняющие ее признаки до тех пор, пока полностью не отвергнет или подтвердит гипотезу.

При поиске *в ширину* система сначала анализирует все симптомы на одном уровне пространства состояний, даже если они относятся к разным болезням, а затем перейдет к признакам следующего уровня детальности.

Целесообразность применения той или иной стратегии во многом зависит от пространства поиска, определяемого характером решаемой прикладной задачи. К примеру, программы для игры в шахматы строятся на основе поиска

в ширину, поскольку при поиске в глубину может потребоваться анализ очень большого числа ходов.

При использовании стратегии *разбиения на подзадачи* в исходной задаче выделяются подзадачи, решение которых рассматривается как достижение промежуточных целей. В частности, такая стратегия хорошо зарекомендовала себя при поиске неисправностей в автомобиле. Сначала выделяется отказавшая подсистема (электропитания, снабжения топливом и т. п.), тем самым существенно сужается пространство поиска. При правильном разбиении цели решаемой задачи на подцели можно добиться, что путь к ее решению в пространстве поиска окажется минимальным.

При использовании *альфа-бета алгоритма* уменьшение пространства состояний обеспечивается путем удаления ветвей, не перспективных для успешного поиска решения. Рассматриваются только те вершины, в которые можно попасть в результате следующего шага, затем неперспективные направления исключаются из рассмотрения. Названный алгоритм нашел широкое применение в основном в системах, ориентированных на игры, например в шахматных программах. Он может использоваться и в продукционных системах для повышения эффективности поиска путем определения оптимальной глубины поиска.

3 ПРЕДСТАВЛЕНИЕ ЗНАНИЙ

3.1 Проблемы представления и классификация знаний

Представление знаний – это соглашение о том, как описывать реальный мир. В естественных и технических науках принят следующий традиционный способ представления знания. На естественном языке вводятся основные понятия и отношения между ними. При этом используются ранее определенные понятия и отношения, смысл которых уже известен. Далее устанавливается соответствие между характеристиками (чаще всего количественными) понятий знания и подходящей математической модели.

Основная цель представления знаний – строить математические модели реального мира и его частей, для которых соответствие между системой понятий проблемного знания можно установить на основе совпадения имен переменных модели и имен понятий без предварительных пояснений и дополнительных неформальных соответствий. Представление знаний обычно выполняется в рамках некоторой системы представления знаний.

При организации представления знаний в экспертных системах, прежде всего, требуется решить следующие *проблемы*:

1. Что представлять. Определение состава представляемых знаний обеспечивает адекватное отображение моделируемых сущностей в системе.

2. Как представлять знания. От решения этой проблемы существенно зависит эффективность и принципиальная возможность представления знаний.

На состав представляемых знаний влияют следующие факторы:

- проблемная среда (предметная область);
- архитектура экспертной системы;
- потребности пользователей;
- язык общения.

Проблемная среда определяет сущности, знания о которых должны храниться и обрабатываться в экспертной системе. Состав знаний в ЭС существенным образом зависит от проблемной среды.

Архитектура экспертной системы. Для функционирования *статической* экспертной системы [18] требуются следующие знания:

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

Для *динамической* ЭС дополнительно требуются знания о методах взаимодействия с внешней средой и о модели внешнего мира [18].

Влияние *потребностей пользователя* на состав знаний определяют следующие факторы:

- решаемые задачи и исходные данные;
- предпочтительные способы и методы решения;
- ограничения на результаты и способы их получения;
- требования к языку общения и организации диалога;
- степень общности и конкретности знаний о проблемной области;
- цели пользователей.

Состав знаний о *языке общения* зависит от выбора языка общения и от требуемого уровня понимания.

Классификация знаний. Знания можно разделить на: интерпретируемые и не интерпретируемые. К *интерпретируемым* относятся знания, которые может анализировать интерпретатор. Остальные знания относятся к не

интепретируемым. Решатель не знает их структуры и содержания. Интерпретируемые знания можно разделить на знания о представлении, предметные и управляющие. Знания *о представлении* содержат информацию о том, в каких структурах представлены интерпретируемые знания.

Предметные знания содержат данные о предметной области и способах их преобразования при решении прикладных задач. В предметных знаниях можно выделить описатели и собственно предметные знания. Описатели содержат информацию о предметных знаниях, такую как коэффициент определенности правил и данных, меры важности и сложности. Собственно предметные знания включают факты и исполняемые утверждения. Факты определяют возможные значения сущностей и характеристик предметной области. Исполняемые утверждения представляют собой знания, задающие процедуры обработки в процедурной и в декларативной форме.

Управляющие знания можно разделить на фокусирующие и решающие. Фокусирующие знания описывают, какие знания следует использовать в определенной ситуации. Решающие знания содержат информацию, используемую для выбора способа интерпретации знаний в данной ситуации. Их применяют для выбора стратегий или эвристик, наиболее эффективных для решения конкретной задачи.

Не интерпретируемые знания подразделяются на вспомогательные и поддерживающие. *Вспомогательные* знания хранят информацию о лексике и грамматике языка общения, о структуре диалога. Они обрабатываются естественно-языковым компонентом. *Поддерживающие* знания используются при создании системы и выполнении объяснений. Они играют роль описаний интерпретируемых знаний и действий системы.

Улучшение количественных и качественных показателей экспертной системы можно добиться на основе использования метазнаний (знаний о знаниях).

К числу возможных назначений метазнаний можно отнести следующие:

- выбор необходимых (релевантных) правил с помощью стратегических метаправил;
- обоснование целесообразности применения правил из области экспертизы;
- обнаружение синтаксических и семантических ошибок в предметных правилах;
- адаптация к окружению путем перестройки предметных правил и функций;
- явное указание возможностей и ограничений системы.

3.2 Классификация моделей представления знаний

Системой представления знаний (СПЗ) называют средства, позволяющие: описывать знания о предметной области с помощью языка представления знаний; организовывать хранение и обработку знаний в системе (накопление, анализ, обобщение и структуризация знаний); вводить новые знания и объединять их с имеющимися; выводить новые знания; находить требуемые знания; удалять устаревшие знания; проверять непротиворечивость накопленных знаний; осуществлять интерфейс между пользователем и знаниями.

Центральное место в СПЗ занимает *язык представления знаний* (ЯПЗ). В свою очередь, выразительные возможности ЯПЗ определяются лежащей в его основе моделью представления знаний (иногда эти понятия отождествляют).

Модель представления знаний является формализмом, который отображает статические и динамические свойства предметной области (ПО), т. е. объекты и отношения ПО, связи между ними, иерархию понятий ПО и изменение отношений между объектами.

Модель представления знаний может быть универсальной (применимой для большинства ПО) или специализированной (разработанной для конкретной ПО). В СИИ используются следующие *основные универсальные модели представления знаний*:

- семантические сети;
- фреймы;
- системы продукций;
- объектно-ориентированный подход;
- логические модели;
- дескриптивная логика и онтологии.

Модели представления знаний делят также на логические (формальные) и эвристические (формализованные). В основе *логических* моделей представления знаний лежит понятие формальной системы (теории). Например, исчисление предикатов и любая конкретная система продукций. В логических моделях, как правило, используется исчисление предикатов первого порядка, дополненное эвристическими стратегиями. Соответствующие экспертные системы являются системами *дедуктивного* типа – в них используется получение вывода из заданной системы посылок с помощью фиксированной системы правил вывода. Дальнейшим развитием предикатных систем являются

системы индуктивного типа, в которых правила вывода порождаются на основе обработки конечного числа обучающих примеров.

В логических моделях представления знаний отношения между компонентами выражаются с помощью ограниченного набора средств, предоставляемых синтаксическими правилами используемой формальной системы.

К *эвристическим* моделям представления знаний можно отнести семантические сети, фреймы, системы продукций и объектно-ориентированный подход.

По сравнению с формальными моделями эвристические модели имеют разнообразный набор средств, учитывающих конкретные особенности предметной области. В связи с этим эвристические модели превосходят логические модели по возможности адекватного представления проблемной среды и по эффективности используемых правил вывода.

3.3 Модели представления знаний

3.3.1 Семантические сети

Семантические сети (СС) являются исторически первым классом моделей представления знаний. Здесь структура знаний о предметной области задается в виде ориентированного графа с размеченными вершинами и дугами. Вершины обозначают сущности и понятия ПО, а дуги – отношения между ними. Под *сущностью* понимают объект произвольной природы. Вершины и дуги могут снабжаться метками, представляющими собой мнемонические имена. Основными элементами СС, с помощью которых формируются понятия, являются:

- класс, к которому принадлежит понятие;
- свойства, выделяющие понятие из всех других понятий этого класса;
- примеры данного понятия.

На самой СС принадлежность элемента к некоторому классу или части к целому передается с помощью связок «IS A» и «PART OF» соответственно. Свойства описываются связками «IS» и «HAS» («является» и «имеет»). На рис. 3.1 приведен пример описания понятия с помощью СС.

С помощью СС можно описывать *события* и *действия*. Для этих целей используются специальные типы отношений, называемые падежами: агент –

действующее лицо, вызывающее действие; объект – предмет, подвергающийся действию; адресат – лицо, пользующееся результатом действия. Возможны и другие падежи типа: время, место, инструмент, цель, качество, количество и т. д. Введение падежей позволяет от поверхностной структуры предложения перейти к его смысловому содержанию.

Рис. 3.2 Пример фрагмента семантической сети

В СС понятийная структура и система отношений представлены однородно. Поэтому представление в них, например, математических соотношений графическими средствами неэффективно. СС не дают ясного представления о структуре ПО; они представляют собой *пассивные* структуры, для обработки которых необходима разработка аппарата формального вывода.

В чистом виде СС на практике почти не используются. При использовании СС обычно либо накладывают ограничения на типы объектов и отношений (примером таких сетей являются функциональные СС), либо расширяют СС специальными средствами для более эффективной организации вычислений (К-сети, пирамидальные сети и др.).

3.3.2 Фреймы

Метод представления знаний с помощью фреймов предложен М. Л. Минским. *Фрейм* – это структура, предназначенная для представления стереотипной ситуации. Каждый фрейм описывает один концептуальный объект, а конкретные свойства этого объекта и факты, относящиеся к нему, описываются в слотах – структурных элементах данного фрейма. Все фреймы взаимосвязаны и образуют единую фреймовую систему, в которой объединены и процедурные знания.

Фрейм определяется как структура следующего вида:

(ИМЯ ФРЕЙМА;
ИМЯ СЛОТА 1 (ЗНАЧЕНИЕ СЛОТА 1)
ИМЯ СЛОТА 2 (ЗНАЧЕНИЕ СЛОТА 2)
.....
ИМЯ СЛОТА N (ЗНАЧЕНИЕ СЛОТА N)).

Определим, например, фрейм для объекта «Служащий»:

(Служащий
ФИО (Петров И.П.)
Должность(инженер)
Категория(2)
.....).

Концептуальному представлению фрейма свойственна иерархичность: целостный образ знаний строится в виде единой фреймовой системы, имеющей иерархическую структуру. В слот можно подставить разные данные: числа или математические соотношения, тексты, программы, правила вывода или ссылки на другие слоты данного или других фреймов.

Если значения слотов не определены, то фрейм называют *фреймом-прототипом*, в противном случае – *конкретным фреймом* или *экземпляром фрейма.*

В теории фреймов ничего не говорится о методах реализации фрейма. В след за появлением теории фреймов появилось целое семейство систем программирования, поддерживающих концепцию фрейм-подхода: **KRL, GUS, FRL, OWL** и др.

Для большинства фреймовых языков свойственно иерархическое описание объектов предметной области с использованием типовых фреймов. При этом широко используется механизм наследования свойств одного объекта (представленных в виде значений слотов связанного с ним фрейма) другими объектами. Используются такие виды наследования, как класс-подкласс, класс-экземпляр класса. Это позволяет согласовать однотипную информацию различных объектов, а также в дальнейшем обеспечить соответствующее их поведение.

Фреймовые системы относят к процедуральной форме представления знаний. Управление выводом в них реализуется путем подключения *присоединенных процедур*, разрабатываемых пользователем.

Процедуры связываются со слотами и именуются демонами и слугами. ***Демон*** – это процедура, которая активизируется автоматически, когда в ее слот подставляется значение или проводится сравнение значений. ***Слуга*** – это процедура, которая активизируется по запросу – при возникновении определенного события.

С использованием присоединенных процедур можно запрограммировать любую процедуру вывода на фреймовой сети. Механизм управления выводом организуется так. Сначала запускается одна из присоединенных процедур некоторого фрейма, называемого образцом. Образец – это по сути фрейм-прототип, т. е. у него заполнены не все слоты, а только те, которые описывают связи данного фрейма с другими. Затем по мере необходимости, путем пересылки сообщений, последовательно запускаются присоединенные процедуры других фреймов, и таким образом осуществляется вывод.

Язык представления знаний, основанных на фреймовой модели, эффективен для структурного описания сложных понятий и решения задач, в которых в соответствии с ситуацией желательно применять различные способы вывода. В то же время на таком языке затрудняется управление завершенностью и постоянством целостного образа. В частности, существует опасность нарушения присоединенной процедуры, проблема зацикливания процесса вывода.

Замечание. Фреймовую модель без механизма присоединенных процедур, а следовательно, и механизма пересылки сообщений, часто используют как базу данных системы продукций.

3.3.3 Продукционные системы

Продукционные системы – это системы представления знаний, основанные на правилах типа **«УСЛОВИЕ-ДЕЙСТВИЕ»**.

Записываются эти правила обычно в виде

ЕСЛИ A_1, A_2, ..., A_n ТО B.

Такая запись означает, что «если выполняются все условия от A_1 до A_n (являются истинными), тогда следует выполнить действие **B**». Часть правила после **ЕСЛИ** называется *посылкой,* а часть правила после **ТО –** *выводом,* или *действием*, или *заключением*.

Условия A_1, A_2, ..., A_n обычно называют *фактами*. С помощью фактов описывается текущее состояние предметной области. Факты могут быть истинными, ложными либо, в общем случае, правдоподобными, когда истинность факта допускается с некоторой степенью уверенности.

Действие **B** трактуется как добавление нового факта в описание текущего состояния предметной области.

В упрощенном варианте описание ПО с помощью правил (продукций) базируется на следующих основных предположениях об устройстве предметной области. ПО может быть описана в виде множества фактов и множества правил.

Факты – это истинные высказывания (в естественном языке – это повествовательные предложения) об объектах или явлениях предметной области.

Правила описывают причинно-следственные связи между фактами (в общем случае и между правилами тоже) – как истинность одних фактов влияет на истинность других.

Описание ПО нетрудно ввести в ЭВМ – для этого достаточно снабдить его соответствующими средствами для хранения множества фактов, например, в виде базы фактов, для хранения правил, например, в базе правил, и построить интерпретатор базы правил, который по описанию текущего состояния ПО в виде предъявленных ему фактов осуществляет поиск выводимых из фактов заключений. На этой идее и построены системы продукций.

Как отмечалось, в экспертных системах продукционного типа используются два основных способа реализации механизма вывода: прямой вывод, или вывод от данных; обратный вывод, или вывод от цели.

В первом случае идут от известных данных (фактов) и на каждом шаге вывода к этим фактам применяют все возможные правила, которые порождают новые факты и так до тех пор, пока не будет порожден факт-цель.

Во втором случае вывод идет в обратном направлении – от поставленной цели. Если цель согласуется с заключением правила, то посылку правила принимают за подцель или гипотезу, и этот процесс повторяется до тех пор, пока не будет получено совпадение подцели с известными фактами.

Рабочая память представляет собой информационную структуру для хранения текущего состояния предметной области. Обмен информацией в продукционной системе осуществляется через рабочую память. К примеру, из одного правила нельзя переслать какие-либо данные непосредственно в другое правило, минуя рабочую память. Состояние рабочей памяти целиком определяет подмножество применимых на каждом шаге вывода правил.

Например, возможная формулировка правил продукций в экспертной системе диагностики автомобиля имеет вид

Если (горит_лампа_датчика_давления_масла
и уровень_масла_норма
и обороты_двигателя_норма
и масляный фильтр_не_засорен)
То (проверить масляный насос).

Приведенное правило позволяет принять решение по ремонту системы смазки автомобиля.

Достоинством применения правил продукций является их модульность. Это позволяет легко добавлять и удалять знания в базе знаний. Можно изменять любую из продукций, не затрагивая содержимого других продукций.

Недостатки продукционныхсистем проявляются при большом числе правил и связаны с возникновением непредсказуемых побочных эффектов при изменении и добавлении правил. Кроме того, отмечают также низкую эффективность обработки систем продукций и отсутствие гибкости в логическом выводе.

Достоинства продукционных систем: простота задания правил ПО и их наглядность.

3.3.4 Введение в логику

Логика занимается изучением законов мышления, одной из главных ее задач является моделирование правильных человеческих рассуждений. Особый интерес к логике возник с появлением ЭВМ, с попытками научить машину рассуждать, т. е. делать логические заключения.

В логике выделяют следующие *формы мышления:* понятия, высказывания и рассуждения.

Понятие о предмете составляет совокупность мыслимых признаков предмета. Понятие выражается словом. Основными способами образования понятий являются:

- сравнение – установление сходства или различия в понятиях;
- анализ – мысленное расчленение целого на составные части;
- синтез – мысленное создание целого из некоторого числа составных частей (признаков, свойств, отношений);
- абстрагирование – мысленное выделение в понятии определенных признаков и отвлечение от других;
- обобщение – объединение различных объектов в однородные группы на основании общих признаков.

Всякое понятие обладает *содержанием* – совокупностью признаков предмета в этом понятии и объемом – совокупностью объектов, входящих в данное понятие.

По *содержанию* понятия делятся на положительные и отрицательные (присутствуют или нет определенные признаки в понятии), безотносительные и относительные, сравнимые и несравнимые.

В логическом отношении друг с другом находятся только *сравнимые* понятия, которые имеют общие признаки в содержании. Сравнимые понятия бывают совместимыми (с совпадающим объемом понятий) и несовместимыми. Выделяют три вида отношений *совместимости*: равнозначность, пересечение, подчинение объемов. Существуют и три вида отношений *несовместимости*: соподчинение, противоположность, противоречие.

Связь между объемом и содержанием понятий выражается в *законе обратного отношения:* если два понятия, сравнимы в логическом смысле, и содержание первого из них больше, чем второго, тогда объем второго понятия больше объема первого. С этим законом связаны способы *обобщения* (переход от понятия с большим объемом и меньшим содержанием к понятию с меньшим объемом и большим содержанием) и *ограничения* (переход от понятия с меньшим объемом и большим содержанием к понятию с меньшим содержанием и большим объемом) понятий.

Одной из основных логических операций над объемом и содержанием понятий является *деление* понятия. При ее выполнении различают: делимое понятие, основание деления (признаки), члены деления (множество видовых понятий по отношению к рассматриваемому). Деление бывает: по *видоизменению* признаков, *дихотомическое* (деление понятия на два класса с противоречивыми признаками).

Практическое применение операции деления понятий – процедура классификации. *Цель классификации* – приведение знаний о предметной области в определенную систему, при этом основание деления должно отвечать цели классификации. Выбор классификационного признака – нетривиальная задача. Классификация, особенно при дихотомическом делении, принимает форму дерева (впервые использовалась сирийским логиком в четвертом веке Порфирием, называется по его имени «дерево Порфирия»).

Другой фундаментальной логической операцией над понятиями является *определениепонятия*. Эта операция позволяет строго закрепить за объектом, обозначенным с помощью определяемого понятия, содержание, выраженное в зафиксированных в признаках, свойствах и отношениях.

Определение бывает номинальное – раскрытие смысла употребления слова и реальное – определение понятий о предметах или явлениях, а не терминов, их обозначающих. Типы определений: через ближайший род и видовое отличие (поиск места понятия в явной или неявной форме классификации).

Понятия являются исходным материалом для построения высказываний (суждений). С грамматической точки зрения, *высказывание* – это повествовательное предложение.

Сложные предложения строятся из выражений, обозначающих некоторые понятия, и логических связок. Слова и фразы «НЕ», «И», «ИЛИ», «ЕСЛИ... ТО», «ТОГДА И ТОЛЬКО ТОГДА», «СУЩЕСТВУЕТ», «ВСЕ» и некоторые другие называются *логическими связками* (операторами) и обозначают логические операции, с помощью которых из одних предложений строятся другие.

Предложения без логических связок являются *элементарными*, их нельзя расчленить на части, чтобы при этом каждая из частей была также

предложением. Элементарные предложения называют также *высказываниями* (суждениями). В высказываниях содержится информация о предметах, явлениях, процессах и т. д.

Элементарное высказывание состоит из субъекта (логического подлежащего) – того, о чем идет речь в высказывании, и предиката (логического сказуемого) – того, что утверждается или отрицается в высказывании о субъекте.

Таким образом, *высказывания* – это форма мышления, в которой утверждается или отрицается логическая связь между понятиями, выступающими в качестве субъекта и предиката данного высказывания. Соответствие или несоответствие этой связи реальности делает высказывание истинным или ложным.

Логическая связь между субъектом и предикатом высказывания выражается обычно в виде связки «ЕСТЬ» или «НЕ ЕСТЬ», в самом предложении эта связка может отсутствовать, а лишь подразумеваться. При этом субъект высказывания может выражаться не обязательно только подлежащим в предложении, также как и предикат не только сказуемым (это могут быть и другие члены предложения). Что считать в предложении субъектом, а что предикатом высказывания, определяется логическим ударением. Оно связано со смыслом, содержащимся в предложении, для говорящего и для слушающего.

По *форме* высказывания делятся на *простые* (имеют логическую форму «S есть P» или «S не есть P», где S – субъект, P – предикат) и *сложные* (грамматически выражаются сложными предложениями).

Простые высказывания позволяют выразить следующие типы высказываний:

- атрибутивные – выражают принадлежность или не принадлежность свойства объекту или классу объектов;
- об отношениях – говорят о наличии отношения между объектами;
- существования(экзистенциональные) – говорят о существовании объекта или явления.

В общем случае простые высказывания можно рассматривать как атрибутивные, понимая под существованием и отношениями вид свойств субъектов высказывания.

По *качеству* простые высказывания делятся на утвердительные и отрицательные. По *количеству* высказывания делятся на следующие группы:

- единичные – субъектом является предмет, существующий в единственном числе;
- частные – в высказывании говорится о пересечении класса предметов, к которому относится субъект, с классом предметов, к которым относится предикат);

- общие – в высказывании говорится о включении или не включении всего класса предметов, к которому относится субъект высказывания, в класс предметов, к которому относится предикат.

Классы предметов, к которым относятся субъект и предикат высказывания, будем обозначать буквами S и P соответственно.

Высказывания, одновременно общие по количеству и утвердительные по качеству называются *общеутвердительными* и имеют форму «Всякий S есть P», обозначается тип высказывания A .

Высказывания, общие по количеству и отрицательные по качеству, называются *общеотрицательными* и имеют форму «Всякий S не есть P», обозначается тип высказывания E.

По данной аналогии выделяют *частноутвердительные* высказывания («Некоторый S есть P», обозначается тип как I) и *частноотрицательные* высказывания («Некоторый S не есть P», обозначается тип как O).

Различают также высказывания сравнимые – имеют одни и те же субъект и предикат и несравнимые – различны субъекты и предикаты суждений.

Третья форма мышления – *рассуждения.* Простейшей формой рассуждений является умозаключение – получение из одного или нескольких высказываний нового высказывания. Принято считать, что из высказываний A_1, A_2, ..., A_n следует высказывание B, если B истинно по крайней мере всегда, когда истинны A_1, A_2, ..., A_n. При этом исходные высказывания A_1, A_2, ..., A_n, из которых делается логический вывод называются *посылками*, а новое высказывание B – *заключением, следствием.* Возможность вывода заключения из посылок обеспечивается логической связью между ними.

Правильные с позиций логики выводы формулируются в виде *правил логического следования* (*правил вывода*). Правила в логике обычно записываются в виде: A_1, A_2, ..., $A_n \Rightarrow B$, здесь $\Rightarrow$ – знак логического следования; записываются правила также в виде дроби

$$\frac{A_1, A_2, ..., A_n}{B}.$$

Таким образом, *рассуждения* – это процесс перехода от посылок к заключениям и далее от полученных заключений как новых посылок к новым заключениям. Выполняется этот процесс в виде элементарных актов, каждый из которых есть шаг вывода, на котором применяется соответствующее правило вывода и называется этот процесс ***логическим выводом***. Число посылок в выводе, как и число его шагов, может быть различным. Вывод за один шаг применения правила вывода называют *непосредственным выводом*.

Нульпосылочные выводы – высказывания, для выяснения истинности которых посылки не нужны, называют **логическими законами**. Наиболее важными являются законы: тождества, противоречия, исключенного третьего и достаточного основания.

Закон тождества: «Объем и содержание всякого понятия должны быть зафиксированы и оставаться неизменными в течение всего процесса рассуждения». Этот закон записывается с помощью формулы **A⇔A,** где ⇔ – знак эквивалентности, т. е. если два понятия тождественны, то они могут быть взаимозаменяемы в логическом контексте. Буквой **A** здесь обозначаем переменную для высказываний или высказывательную форму.

Закон противоречия: «Два противоречащих друг другу высказывания не могут быть одновременно истинными, по крайней мере, одно из них ложно». Формульная запись закона имеет вид: **¬(A ∧¬A)** – «неверно, что**A** и не **A**».

Закон исключенного третьего (латинская формулировка этого закона – *Tertium non datur* – «Третьего не дано») выражается формулой: **A∨¬A**, т. е. «истинно**A** или не **A**» или словесная формулировка – если два высказывания противоречат друг другу, то одно из них истинно, а другое ложно.

Закон достаточного основания предложен немецким философом Г. Лейбницем, он требует, чтобы ни одно утверждение не признавалось справедливым без достаточного основания. Закон в целом не выражается какой либо формулой, его требования носят содержательный характер. *Двухпосылочные выводы* называют *силлогизмами* Аристотеля.

Силлогизм – это рассуждение, состоящее из трех атрибутных высказываний: двух посылок и одного заключения, например:

«Всякий человек смертен, Сократ – человек =>Сократ смертен».

Конкретные типы силлогизмов называют *модусами*. В силлогистике для четырех типов высказываний **A, I, E** и **O** можно получить 256 различных модусов – правил вывода.

В общем случае различают следующие рассуждения:

- *индуктивные* – от частного к общему;
- *достоверные – дедуктивные*, от общего к частному;
- *правдоподобные* – от частного к частному.

Индуктивные рассуждения от частного к общему отражают путь познания окружающего мира. Общие утверждения возникают при обобщении частных, отражающих совокупность единичных фактов, полученных из опыта. Истинность общего утверждения очевидна, если частных утверждений, подтверждающих общий результат, достаточно много и нет опровергающих утверждений.

Полной индукцией называют рассуждения, в которых общее заключение о принадлежности некоторого свойства предметам данного класса делается на основании его принадлежности всем предметам этого класса. Полная индукция дает истинное знание при условии, что граница рассматриваемого класса объектов точно известна.

Неполная индукция – это перенос знаний о части объектов данного класса на все объекты этого класса. Она основывается на свойстве повторяемости признаков у сходных объектов. Здесь возможны ошибочные индуктивные заключения из-за применения второстепенных признаков в качестве существенных, т. е. в индуктивных рассуждениях из истинных посылок могут получаться ложные заключения.

Правдоподобны индуктивные рассуждения достигаются на основе выявления повторяемости признаков у объектов класса, а также их взаимосвязи и причинной зависимости между этими признаками и свойствами объектов.

В индуктивных выводах используются различные методы установления причинно-следственных отношений. Формулируются они в виде принципов, основными из которых являются ***принципы:*** *единственного различия, единственного сходства, единственного остатка, аналогии* и другие.

Например, формулировка *принципа единственного различия*: «Если после введения какого-либо фактора появляется, или после удаления его исчезает известное явление, причем мы не вводим и не удаляем никакого другого обстоятельства, которое могло бы изменить явление, и не производим изменения первоначальных условий явления, то указанный фактор и составляет причину явления».

Этот принцип можно описать следующим образом: «Пусть в серии из n опытов A,B,C вызывают Д; в другой серии из n опытов B,C не вызывают Д». Тогда на основании наблюдений можно сделать следующий вывод: «Вероятно, А является причиной Д».

Применяются также ***нечеткие выводы***, когда истинность посылок принимается с некоторой степенью уверенности и заключение также выводимо из таких посылок с определенной степенью достоверности (вероятности).

Основные идеи, лежащие в основе ***дедуктивных рассуждений***, восходят к работам Аристотеля и состоят в следующем:

1) исходные посылки рассуждения являются истинными;

2) правильно применяемые приемы перехода от посылок к вытекающим из них утверждениям и из посылок и ранее полученных утверждений к новым вытекающим из них утверждениям должны сохранять истинность получаемых утверждений – истинные посылки порождают истинные следствия.

Наиболее важные практические результаты в СИИ получены при использовании дедуктивных рассуждений, на основе которых построено большинство логических систем представления знаний.

3.3.5 Логические системы

В основе логических систем представления знаний лежит понятие ***формальной логической системы***. Оно является также одним из основополагающих понятий формализации. Основные идеи ***формализации*** заключаются в следующем:

1 Вводится множество базовых элементов (*алфавит*) теории.

2 Определяются правила построения правильных объектов (*предложений*) из базовых элементов.

3 Часть объектов объявляется изначально заданными и правильными по определению – *аксиомами*.

4 Задаются правила построения новых объектов из других правильных объектов системы (*правила вывода*).

Эта схема лежит в основе построения многих ***дедуктивных*** СИИ. В соответствии с ней база знаний описывается в виде предложений и аксиом теории, а механизм вывода реализует правила построения новых предложений из имеющихся в базе знаний. На вход СИИ поступает описание задачи на языке этой теории в виде запроса (предложения, теоремы), которое явно не представлено в БЗ, но если оно верно с позиции заложенных в нее знаний, то может быть построено из объектов БЗ путем применения правил вывода. Процесс работы механизма вывода называют ***доказательством запроса*** (теоремы).

Формальные языки, на которых записываются предложения (формулы) с использованием рассмотренных понятий, называют ***логическими языками***. С практической и теоретической точек зрения наиболее важными и изученными являются *язык логики высказываний* и *язык логики предикатов*.

В языке логики высказываний элементарные предложения рассматриваются как неделимые сущности, в языке логики предикатов делается расчленение предложения на субъект и предикат.

При математизации рассуждений различают два вида слов: ***термы*** – аналоги имен существительных и ***формулы*** – аналоги повествовательных предложений.

Для записи предложений используются стандартные формы высказываний. Это позволяет стандартизовать рассуждения, т. е. рассматривать определенные структуры посылок и заключений, а также ввести в термы *переменные* – именные формы, которые обращаются в имена после подстановки вместо переменных конкретных значений.

Формулы с переменными, обращающиеся в высказывания при подстановке значений, называют ***высказывательными формами*** или переменными высказываниями. Одна форма порождает множество истинных или ложных высказываний.

Не все предложения, содержащие переменные, являются высказывательными формами. Различают связанные и свободные переменные. Так, сложные предложения с переменными, содержащие логические связки «СУЩЕСТВУЕТ» или «ВСЕ», обозначают высказывания, а переменные, к которым они относятся, являются связанными.

Расчленение предложения на субъект и предикат в математической логике выполняется путем соотнесения предложения, выражающего свойства предмета, с функцией одной переменной **P(x)**. При этом сама функция **P** – логическая функция одной переменной, т. е. ***одноместный предикат***, а аргумент **x** – ***субъект***. Если же предложение описывает отношение между несколькими (**n**) субъектами, то с ним можно связать **n**-местную логическую функцию **P(x1,x2,...,xn)** – **n**-местный предикат.

Для построения сложных предложений (формул) используются логические связки «И», «ИЛИ», «НЕ» и т. д., которые соотносятся с операциями логики следующим образом:

неверно что – $\neg$ (знак отрицания);
и – $\wedge$ (знак конъюнкции);
или – $\vee$ (знак дизъюнкции);
если ... то – $\rightarrow$ (знак импликации);
тогда, когда – $\Leftrightarrow$ (знак эквивалентности).

Логические связки «ДЛЯ ВСЯКОГО», «СУЩЕСТВУЕТ» относятся к переменным в предложении и обозначают:

для всякого – $\forall$ – знак квантора общности;
существует – $\exists$ – знак квантора существования.

В различных логических системах используются разнообразные правила вывода. Приведем два наиболее распространенных из них.

Первое – «***правило подстановки***» имеет следующую формулировку. В формулу, которая уже выведена, можно вместо некоторого высказывания подставить любое другое при соблюдении условия: подстановка должна быть сделана во всех местах вхождения заменяемого высказывания в данную формулу.

Второе – «***правило заключения***» (латинское название ***Modus ponens*** – положительный модус) состоит в следующем: Если α и $\alpha \rightarrow \beta$ являются истинными высказываниями посылками, тогда и высказывание заключение β также истинно. Записывается правило в виде дроби

$$\frac{\alpha,\ \alpha \to \beta}{\beta} \quad .$$

Пример. Пусть имеются следующие истинные высказывания:

1 Если самолет проверен и заправлен, то он готов к вылету.

2 Если самолет готов к вылету и дано разрешение на взлет, то он либо взлетел, либо находится на взлетной полосе.

3 Если самолет взлетел, то он выполняет рейс.

4 Самолет ЯК-42 проверен и заправлен.

5 Самолет ТУ-134 проверен.

6 Самолет ИЛ-62 заправлен.

7 Самолету ЯК-42 дано разрешение на вылет.

8 Самолет ЯК-42 не находится на взлетной полосе.

Требуется *найти*, какой из самолетов в момент времени Т выполняет рейс.

Проведем анализ данных высказываний. Высказывания 1, 2, 3 являются сложными и построены с использованием логических связок $\to$ (импликация), $\wedge$ (И). Во всех элементарных высказываниях, из которых построены предложения 1, 2, 3, субъектом является понятие «самолет»; предикатами выступают сказуемые, описывающие свойства всех объектов, принадлежащих классу «самолет». Высказывания 4-8 являются фактами, истинными на момент времени Т. Они являются элементарными высказываниями, описывающими свойства конкретных объектов предметной области.

Для формального описания задачи введем следующие одноместные предикаты:

ПРОВЕРЕН(X) – самолет X проверен;
ЗАПРАВЛЕН(X) – самолет X заправлен;
ГОТОВ(X) – самолет X готов к вылету;
ДАНО_РАЗР(X) – самолету X дано разрешение на вылет;
ВЗЛЕТЕЛ(X) – самолет X взлетел;
НАХ_ВЗП(X) – самолет X находится на взлетной полосе;
НЕ_НАХ_ВЗП(X) – самолет X не находится на взлетной полосе;
ВЫП_РЕЙС(X) – самолет X выполняет рейс.

Тогда исходное описание на языке логики предикатов будет иметь вид:

1 $\forall$X(ПРОВЕРЕН(X)$\wedge$ ЗАПРАВЛЕН(X) $\to$ ГОТОВ(X)).

2 $\forall$X(ГОТОВ(X) $\wedge$ДАНО_РАЗР(X)$\wedge$НЕ_НАХ_ВЗП(X) $\to$ ВЗЛЕТЕЛ(X)).

3 $\forall$X(ГОТОВ(X) $\wedge$ДАНО_РАЗР(X)$\wedge\neg$ ВЗЛЕТЕЛ(X) $\to$ НАХ_ВЗП(X)).

4 $\forall$X(ВЗЛЕТЕЛ(X) $\to$ ВЫП_РЕЙС(X)).

5 ПРОВЕРЕН(ЯК-42).

6 ЗАПРАВЛЕН(ЯК-42).

7 ПРОВЕРЕН(ТУ-134).

8 ЗАПРАВЛЕН(ИЛ-62).

9 ДАНО_РАЗР(ЯК-42).

10 НЕ_ НАХ_ВЗП(ЯК-42).

Предложения 1-4, хотя и содержат переменную, являются высказываниями – переменная X связана квантором общности $\forall$. В дальнейшем квантор писать не будем, так как он присутствует во всех предложениях.

Чтобы найти, какой из самолетов в момент времени T выполняет рейс, подготовим запрос вида

$$M \rightarrow \text{ВЫП\_РЕЙС}(Z),$$

где M – множество предложений 1-10.

Вывод запроса можно представить следующей последовательностью шагов:

1 шаг.

Применив к предложению 1 подстановку X=ЯК-42, получим заключение

$$\text{ПРОВЕРЕН}(\text{ЯК-42}) \wedge \text{ЗАПРАВЛЕН}(\text{ЯК-42}) \rightarrow \text{ГОТОВ}(\text{ЯК-42}).$$

2 шаг.

Первая посылка: объединив предложения 5 и 6, получим

$$\text{ПРОВЕРЕН}(\text{ЯК-42}) \wedge \text{ЗАПРАВЛЕН}(\text{ЯК-42}) .$$

Вторая посылка: заключение шага 1

$$\text{ПРОВЕРЕН}(\text{ЯК-42}) \wedge \text{ЗАПРАВЛЕН}(\text{ЯК-42}) \rightarrow \text{ГОТОВ}(\text{ЯК-42}).$$

Применив правило ModusPonens, имеем

$$\frac{\alpha,\ \alpha \rightarrow \beta}{\beta},$$

для α = ПРОВЕРЕН(ЯК-42) $\wedge$ ЗАПРАВЛЕН(ЯК-42)
и β = ГОТОВ(ЯК-42), получим следующее заключение: ГОТОВ(ЯК-42).

3 шаг.

Первая посылка: объединив заключение шага 2, предложения 9 и 10, получим: ГОТОВ(ЯК-42) $\wedge$ ДАНО_РАЗР(ЯК-42) $\wedge$ НЕ_ НАХ_ВЗП(ЯК-42).

Вторая посылка: применив к правилу 2 подстановку X = ЯК-42, получим

$$\text{ГОТОВ}(\text{ЯК-42}) \wedge \text{ДАНО\_РАЗР}(\text{ЯК-42}) \wedge \text{НЕ\_НАХ\_ВЗП}(\text{ЯК-42}) \rightarrow \text{ВЗЛЕТЕЛ}(\text{ЯК-42}).$$

Применив правило ModusPonens, получим заключение ВЗЛЕТЕЛ(ЯК-42).

4 шаг.

Первая посылка: заключение шага 3 – ВЗЛЕТЕЛ(ЯК-42).

Вторая посылка: применив к правилу 4 подстановку X = ЯК-42, получим

$$\text{ВЗЛЕТЕЛ}(\text{ЯК-42}) \rightarrow \text{ВЫП\_РЕЙС}(\text{ЯК-42}).$$

Применив правило ModusPonens, получим заключение ВЫП_РЕЙС(ЯК-42).

Таким образом, в момент времени T рейс выполняет самолет ЯК-42. Остальные подстановки, например X = ИЛ-62, приводят к тупиковым

ситуациям. Логический вывод выполнялся нами в прямом направлении, при этом в процессе вывода трижды использовалось правило заключения.

3.4 Дескриптивная логика и онтологии

3.4.1 Характеристика дескриптивных логик

Дескриптивные логики (DL) – семейство логик, предназначенных для моделирования знаний и формального описания терминологий.

В отличие от множества других средств моделирования знаний (фреймов, семантических сетей, графических языков моделирования), дескриптивные логики имеют не только строгий синтаксис, но и формальную семантику, обеспечивающую возможность машинной обработки баз знаний.

В дескриптивных логиках используются два вида отношений – унарные (концепты) и бинарные (роли, свойства). Отношения могут быть как атомарными, так и сложными, построенными на основе других отношений с помощью операторов, разрешенных в диалекте логики.

Роли делятся на абстрактные (связывающие между собой объекты и концепты) и атрибутивные (связывающие объекты со значениями примитивных типов данных).

База знаний (онтология) в DL строится в виде множества утверждений-*аксиом*. Все утверждения в онтологии разделяют на две категории:

- *терминологические* аксиомы (**TBox**, сокр. от terminological box), описывающие концепты и роли;
- *экземплярные* аксиомы (**ABox**, assertional box), описывающие отношения «экземпляр класса» или «экземпляр свойства».

В TBox описывается структура понятий предметной области; в ABox – набор фактов о конкретных объектах (к каким классам они относятся и какими свойствами обладают).

Важной особенностью дескриптивных логик как формальных моделей является следование предположению об открытости мира и не следование предположению об уникальности имен:

1 *Предположение об открытости мира* (Open World Assumption, OWA) говорит о том, что истинность некоторого знания не зависит от известности наблюдателю (агенту) о его истинности; и наоборот − из факта невозможности совершения вывода из-за недостатка данных нельзя сделать вывод о ложности

некоторого утверждения. ПОМ ограничивает возможность вывода заключений теми, которые следуют из явно известных фактов. В подавляющем большинстве процедурных языков программирования и реляционных баз данных используется предположение о замкнутости мира (CWA, Closed World Assumption), из которого следует ложность любого утверждения, о котором не высказано обратного (например, при CWA отсутствие утверждения о продаже билета говорит о том, что билет не был продан).

2 *Предположение об уникальности имен* (Unique Name Assumption, UNA) говорит о том, что в базе знаний различные имена обязательно ссылаются на разные объекты. В большинстве диалектов дескриптивных логик это предположение **не** делается − при отсутствии явного утверждения об обратном, два имени *могут* рассматриваться как ссылающиеся на *один и тот же объект*. Для выражения утверждений о различии или совпадении объектов в языках, основанных на DL, вводятся специальные операторы (например, в языке OWL это owl:sameAs и owl:differentFrom). Однако, UNA может приниматься в диалектах без конструктора номиналов (задания класса через его экстенсионал).

3.4.2 Синтаксис дескриптивных логик

Диалекты дескриптивных логик отличаются между собой наборами допустимых выражений и, соответственно, выразительной мощью и вычислительной сложностью. Набор операторов, доступных в некоторой логике, отражается в ее названии. *Базовых логик*, на основании которых строятся все остальные, имеется три: *AL* (атрибутивный язык), *FL* (фреймовый язык) и *EL* (экзистенциальный язык). Язык *EL* наиболее прост – он допускает использование оператора пересечения и экзистенциального оператора. Язык *FL* допускает пересечение концептов, универсальное и экзистенциальное ограничение свойств, а также описание ролей. Наиболее широко используется язык *AL* и его расширение *ALC* – «Атрибутивный язык с комлементом» – *AL* с разрешенным оператором дополнения составных концептов.

Широко используется следующее соглашение об именовании, в котором добавление операций в логику добавляет в ее название соответствующий символ:

C – отрицание (комплемент) составных концептов;

U – объединение концептов;

I – инверсные роли (R` является обратным по отношению к R, если R`(x, y) эквивалентно R (y, x));

F – функциональные свойства (объявляемые только один раз для каждого объекта);

H – иерархия свойств;

E – разрешение использовать полноценное экзистенциальное ограничение свойств (с любыми классами, а не только с универсальным классом ⊤);

R – рефлексивность, нерефлексивность и дизъюнктивность свойств;

O – «номиналы» – возможность задания концептов через перечисление его экземпляров (в OWL – one Of, hasValue);

N – ограничение на размер области значений свойств без указания области значений (non-qualified cardinality restrictions, в OWL – cardinality, max Cardinality);

Q – ограничение на размер области значений свойств с указанием области значений (qualified cardinality restrictions);

(D) – разрешение на использование примитивных типов данных и атрибутивных свойств (с примитивными типами в качестве области значений).

Ввиду того, что подавляющее большинство логик строится на основе логики ALC, дополненной транзитивными свойствами, для сокращения записи названий используют вместо последовательности ALC+ символ *S*. Примеры широко используемых логик: *EL++* или *ELRO* – логика *EL* с расширенными возможностями описания свойств – лежит в основе языка OWLEL, в котором формализована крупнейшая онтология в области медицины SNOMEDCT; $SROIQ^{(D)}$ – логика, используемая в OWL 2; $SHOIN^{(D)}$ – OWL-DL.

Приведем описание синтаксиса логики *ALC*. Язык *ALC* содержит множество базовых символов, который состоит из трех множеств:

N_C – множество имен базовых (атомарных) классов и двух специальных классов – универсального класса ⊤ и пустого класса ⊥;

N_R – множество имен ролей;

N_I – множество имен объектов.

Сложные классы описываются с помощью операторов − «конструкторов». В *ALC* используется пять операторов:

- пересечение классов (конъюнкция): C ⊓D;
- объединение классов (дизъюнкция): C⊔D;
- дополнение класса (отрицание): ¬C;
- универсальное ограничение свойства: ∀R.C;
- экзистенциальное ограничение свойства: ∃R.C.

В ALC используются три формы логических формул – аксиом. Терминологические аксиомы записываются в форме описания вложенности классов (general concept inclusion, GCI): C ⊑D. Например, если мы имеем заданные классы *Мать* и *Отец*, класс *Родитель* описывается как вложенный в их объединение: *Родитель*⊑*Мать*⊔*Отец*. Если имеются две аксиомы C⊑D и

D⊑C, записывается аксиома эквивалентности классов C ≡ D. Набор терминологических аксиом называется TBox – «терминологический набор». Фактически, TBox может рассматриваться как иерархия понятий в выбранной для моделирования предметной области.

Аксиомы вида «экземпляр класса» имеют форму a: C, где a входит в множество имен объектов, C − в множество имен классов. Аксиомы «экземпляр свойства» имеют вид (a,b): P, где a и b являются именами объектов, а P является именем роли, входящим в N_R. Набор экземплярных аксиом формирует ABox.

3.4.3 Семантика дескриптивных логик

Семантика дескриптивных логик является модельно-теоретической: интерпретация языка определяется парой из его области интерпретации (*носителя*, или *домена*) Δ и интерпретирующей функции *I*.

Домен – конечное множество элементов; функция интерпретации определяется следующим образом:

I отображает концептуальный символ на некоторое подмножество домена, при этом I(⊤) = Δ и I(⊥)= ∅;

I отображает свойства на отношения в Δ (подмножество декартова произведения Δ × Δ);

I отображает каждый объект a на элемент в Δ.

Сложные классы интерпретируются следующим образом:

- интерпретация $I(C \sqcap D)$эквивалентна $I(C) \sqcap I(D)$;
- $I(C \sqcup D) = I(C) \sqcup I(D)$;
- $I(\neg C) = \Delta \setminus I(C)$;
- $I(\forall R.C) = \{x \in \Delta | \forall y \in \Delta, (x,y) \in I(R) \rightarrow y \in I(C)\}$;
- $I(\exists R.C) = \{x \in \Delta | \exists y \in I(C), (x,y) \in I(R)\}$.

Интерпретация I удовлетворяет аксиоме $C \sqsubseteq D$, если $I(C) \sqsubseteq I(D)$; I удовлетворяет аксиоме $a:C$, если $I(a) \in I(C)$; I удовлетворяет аксиоме (x,y):P, если $(I(x), I(y)) \in I(P)$.

Если интерпретация *I* удовлетворяет некоторой аксиоме A, то она называется *моделью* A. Интерпретация *I* удовлетворяет (или является моделью) TBox, если она является моделью для всех аксиом, входящих в TBox.

Задание модели и интерпретирующей функции обеспечивает возможность проведения логического вывода по соответствующей базе знаний.

Для логики ALC выделяют несколько *основных задач вывода*:

1 Проверка *согласованности* (непротиворечивости) онтологии. Онтология называется непротиворечивой, если для нее существует хотя бы одна модель −

то есть, существует способ интерпретации классов, объектов и свойств, не противоречащий ни одной из содержащихся в онтологии аксиом.

2 Проверка *корректности* (когерентности) определения *класса*. Класс является корректным, если существует хотя бы одна модель, в которой он не является пустым множеством.

3 Проверка *принадлежности* объекта некоторому классу.

4 Проверка *существования* роли между двумя объектами.

5 Вывод аксиом. Из онтологии O выводится (следует) аксиома A, если каждая модель O также является моделью A.

Задача проверки согласованности является ключевой среди всех задач, так как к ней сводятся остальные. Проверка когерентности класса C может выполняться путем добавления новой аксиомы a: C, где a – новое имя, и проверки согласованности полученной онтологии. Вывод аксиом может быть выполнен путем добавления обратных к выводимым аксиом и проверки того, что онтология перестает быть согласованной.

3.4.4 Онтологии

В философии «онтология» – термин, обозначающий знание о сущем, бытии, определенную структуру категорий, отражающую определенную точку зрения на окружающую действительность. В современности данный термин чаще используется в информатике, в области обработки естественных языков, поиска данных и представления знаний. Также понятие онтологии является одним из ключевых в проекте Semantic Web.

В информатике под онтологией понимается записанная на формальном языке концептуальная модель предметной области, разделяемая некоторым сообществом. Онтология содержит иерархию понятий (концептов) предметной области, дополненную иерархией связей между ними, а также множество утверждений, описывающих смысл понятий.

Формальной моделью онтологии является четверка:

$$O = \langle L, C, P, R, A \rangle,$$

где L – множество лексических символов – имен концептов и отношений, C – множество описаний концептов предметной области; P – множество описаний свойств концептов; R – множество отношений между концептами; A – множество аксиом – утверждений, построенных с использованием описанных концептов и отношений.

Онтологии по цели создания разделяют на онтологии верхнего уровня, предметных областей и прикладные. Также выделяют особую категорию *онтологий языка представления*, описывающих формализм, на котором записываются другие онтологии (например, описание понятий языка OWL).

Онтологии верхнего уровня (upper или top-levelontology) описывают знания, общие для множества предметных областей – сущность, категория, пространство, время, событие, процесс, объект и т. д. Примерами подобных онтологий являются DOLCE[1], SUMO[2], Cyc[3], UMBEL[4].

Онтология предметной области строится с целью описания общих концепций для множества принадлежащих одной области задач. Подобная онтология абстрагирована от конкретных задач, но может использоваться профессиональным сообществом в качестве разделяемого формального словаря, а также применяться с целью интеграции специализированного ПО. В качестве примера приведем онтологию SNOMEDCT[5], содержащую медицинскую терминологию, и UNSPSC[6] – общая в рамках ООН терминология в области услуг и товаров.

Онтология прикладного уровня описывает концептуальную модель конкретной задачи либо информационной системы.

Онтологии находят свое применение в нескольких типовых задачах:

1 **Построение общего словаря.** Термины словаря используются при описании информационных ресурсов и в общении внутри профессионального сообщества.

2 **Информационный поиск.** Онтологии применяются для разметки содержания индексируемых документов (концептуальное индексирование) и указания точного смысла слова в запросе. Это позволяет улучшить качество поиска, так как выбор документов производится на основе обнаружения запрошенного понятия, а не лексической формы. При использовании подобного подхода слова-синонимы соотносятся с соответствующим им концептом, а множество различных понятий может быть связано с многозначным словом.

3 **Интеграция источников данных**. В онтологии описываются схемы данных нескольких источников данных. Термины общей схемы используются для формулирования запросов к интегрирующей системе, та, в свою очередь, переформулирует запросы в термины хранилищ данных.

---

[1] http://www.loa.istc.cnr.it/DOLCE.html

[2] http://www.ontologyportal.org/

[3] В открытом доступе находится часть онтологии - OpenCychttp://www.opencyc.org/

[4] http://umbel.org/

[5] http://www.ihtsdo.org/snomed-ct/ - Systematized Nomenclature of Medicine – Clinical Terms

[6] http://www.unspsc.org/ - UnitedNationsStandardProductsandServicesCode

С различными примерами онтологий можно ознакомиться в библиотеках онтологий; наиболее популярными являются библиотека на сайте Protege[7] и библиотека проекта DAML[8].

3.4.5 Языки представления онтологий

Широчайшее применение онтологии нашли в проекте Semantic Web – направлении развития существующей сети WWW, которая обеспечит машинную обработку документов.

В рамках этого направления консорциумом W3C разработано несколько специализированных языков:

- RDF (Resource Description Framework) – язык описания представленных в WWW данных;
- OWL (Web Ontology Language) – язык представления онтологий в Web;
- SPARQL (SPARQL Protocol and RDF Query Language) – язык запросов к хранилищам RDF-описаний.

Язык **RDF**[9] предназначен для описания *ресурсов* в сети WWW. Под ресурсом понимается все, что имеет уникальный идентификатор, на который можно сослаться в сети Интернет. Моделью данных в языке RDF является направленный мультиграф, описываемый множеством *троек* (триплетов) – высказываний в форме «субъект»-«предикат»-«объект». В качестве вершин графа выступают субъекты и объекты дуги всегда ведут от субъекта к объекту (рис. 3.2):

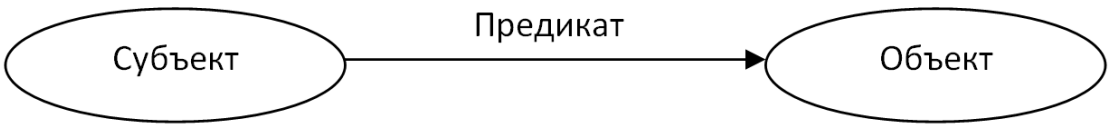


Рис. 3.2 Модель «тройки» RDF

Субъект и предикат в высказывании всегда идентифицируются при помощи URI. Объектом может выступать как ресурс, имеющий URI, так и константное значение – *литерал*. Предположим, что «ИВС» и «является кафедрой в» – два URI ресурсов, тогда на их основе можно построить триплет (рис. 3.3) «ИВС – является кафедрой в – `ПГУПС`», где `ПГУПС` – литерал:

---

[7] http://protegewiki.stanford.edu/wiki/Protege_Ontology_Library

[8] http://www.daml.org/ontologies/

[9] http://www.w3.org/RDF/

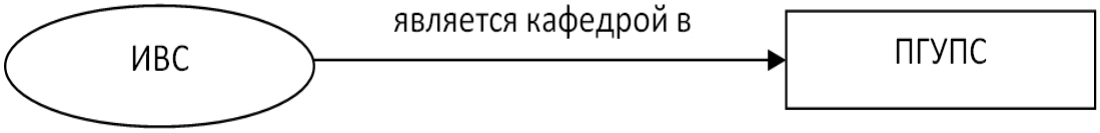


Рис. 3.3 Пример высказывания RDF с объектом-литералом

Язык RDF предоставляет минимальные возможности для описания модели данных. В нем предопределено всего восемь классов сущностей (класс литералов rdf:XMLLiteral, класс всех отношений – rdf:Propery, четыре контейнерных класса и класс, представляющий собой пустой контейнер и семь отношений.

Четыре из доступных свойств предназначены для моделирования высказываний с целью последующего их снабжения метаданными: rdf:Statement, rdf:subject, rdf:predicate, rdf:object. Свойства rdf:first и rdf:rest предназначены для описания элементов в контейнерных объектах. Свойство rdf:type предназначено для описания принадлежности субъекта классу сущностей, идентифицируемого объектом.

На рис. 3.4 приведен пример высказывания о том, что «ИВС» является объектом класса «Кафедра»:

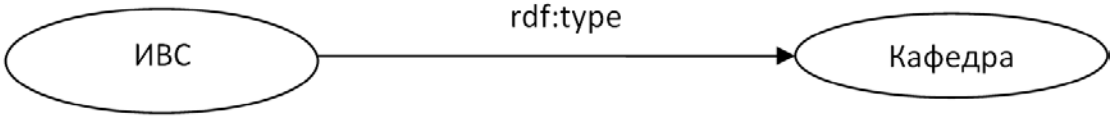


Рис. 3.4 Высказывание с предикатом rdf:type

Поскольку отношения в тройках являются строго бинарными, описание более мощных отношений производится либо с добавлением промежуточных объектов-ресурсов, либо с использованием не именованных «пустых» узлов (*blanknodes*).

Приведем пример (рис. 3.5):

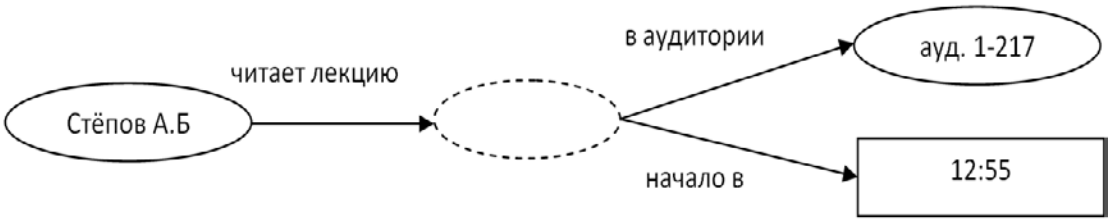


Рис. 3.5 Высказывание с анонимным узлом

Семантика RDF[10] достаточна для простого описания ресурсов, но недостаточно выразительна для построения сложных иерархий понятий, так

[10] http://www.w3.org/TR/rdf-mt/

как не позволяет описать отношения между классами ресурсов и свойствами. Для описания онтологий используются языки, «надстроенные» над моделью RDF.

Язык **RDFS**[11] (RDF Schema) является семантическим расширением RDF, добавляющим возможность описания иерархий классов и свойств.

В языке объявлены несколько специальных классов:

- rdfs:Resource – класс всех ресурсов;
- rdfs:Class – класс, включающий в себя все ресурсы, являющиеся классами;
- rdfs:Literal – класс всех литералов;
- rdfs:Datatype – класс типов данных, одновременно являющийся как экземпляром класса rdfs:Class, так и его подклассом;
- rdfs:Container – надкласс контейнерных классов языка RDF;
- rdfs:Container Membership Property – класс свойств, используемых для описания включения ресурса в контейнер (свойств с именами вида rdf:_nnn, где nnn – целое положительное число); каждый экземпляр этого класса является подтипом свойства rdfs:member.

На множестве классов определено отношение «класс-подкласс», описываемое свойством rdfs:sub Class Of. Семантика этого отношения состоит в том, что экстенсионал класса *D*, объявленного подклассом класса *C*, целиком содержится в экстенсионале класса *C* – каждый ресурс, объявленный как экземпляр класса *D*, является также экземпляром класса *C*.

Аналогично классам, на множестве свойств определено отношение включения rdfs:sub Property Of, позволяющего описывать иерархии свойств (например, свойство «обучается в ВУЗе» является подтипом свойства «обучается в организации»).

Помимо него, имеется несколько выражений, предназначенных для описания свойств и указания поясняющей информации:

- rdfs:domain – свойство предназначено для описания домена свойств. Доменом является некоторое множество классов, к которым свойство применимо;
- rdfs:range – диапазон значений свойств. Диапазон определяет допустимое множество значений свойства – ресурсов или литералов;
- rdfs:label – название ресурса, которое может прочесть человек;
- rdfs:comment – текстовое описание ресурса;
- rdfs:see Also – свойство предназначено для указания ссылки на ресурс, содержащего дополнительные данные по субъекту свойства;

---

[11] http://www.w3.org/TR/rdf-schema/

- rdfs:is Defined By – свойство ссылается на ресурс, хранящий определение субъекта;
- rdfs:member – свойство, являющееся базовым типом для всех свойств, описывающих включение ресурса в коллекцию.

Важно отметить, что модели RDF и RDFS опираются не на определение классов (как, например, фреймовая модель или объектная модель в программировании), а на описание свойств. Свойства в RDFS описываются как пары [домен; диапазон], а описание класса состоит из описания применимых к объектам класса свойств, и, вследствие этого, оно является открытым для дополнения.

RDFS расширяет возможности RDF, но не является полноценным языком для записи онтологий – его выразительная мощность недостаточна. В RDFS нет возможности указать ограничения на существование или мощность некоторого отношения, нет средств описания симметричных, обратных или транзитивных свойств.

Семейство языков **OWL**[12] (Web Ontology Language) содержит языки представления онтологий, основанные на дескриптивных логиках. В семействе присутствует несколько диалектов, различающихся по степени выразительности и вычислимости.

В первой спецификации языка OWL определены три диалекта:

- OWL Lite – подмножество языка, реализующее логику *SHIN* (без номиналов), за счет чего обеспечивает гарантированную разрешимость в линейное время;
- OWL DL – реализация более выразительной дескриптивной логики $SHOIN^{(D)}$ с некоторыми ограничениями (напр., запрет использования номиналов, запрет объявления класса экземпляром другого класса). Допускается использование типов данных XML. Этот диалект также является разрешимым, но вычислительная сложность его логики намного выше, чем для логики *ALC*;
- OWL Full – наиболее выразительный диалект, полностью реализующий дескриптивную логику $SHOIN^{(D)}$. Диалект полностью совместим с RDF – любой RDF-документ является корректным OWL Full документом. Для этого диалекта не гарантирована вычислимость – эффективных процедур обработки не существует.

Каждый из диалектов OWL является подмножеством следующего: онтология OWL Lite является корректной онтологией OWL DL, а онтология DL, в свою

---

[12] http://www.w3.org/TR/owl2-overview/

очередь, корректной онтологией OWL Full. Графически отношение диалектов между собой и RDF/S может быть изображено как показано на рис. 3.6.

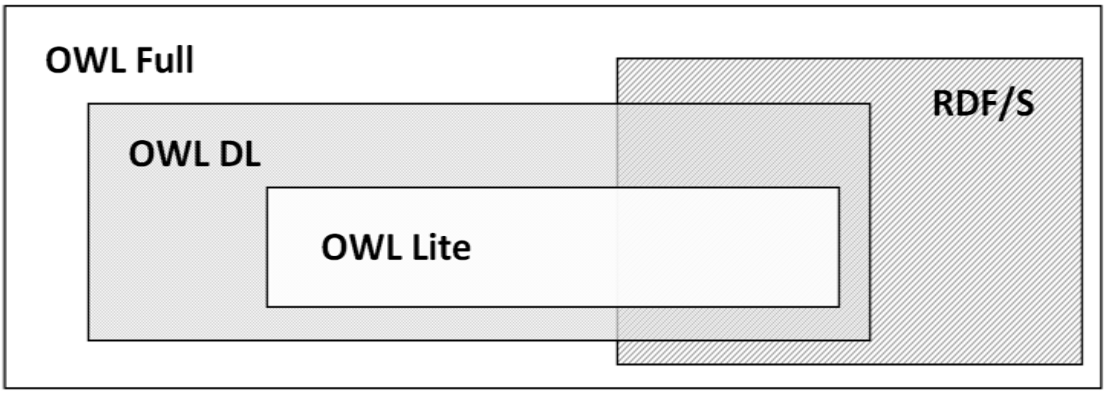


Рис. 3.6 Визуальное соотношение диалектов OWL 1 и RDF/S

Спецификация OWL 2, объявленная рекомендованной в октябре 2009 года, помимо добавления новых выражений, дополнила семейство языков еще тремя «профилями», каждый из которых содержит такое подмножество допустимых выражений, которое гарантирует полиномиальное время выполнения задач логического вывода над онтологией:

- OWL EL – простейший диалект языка, реализующий логику *EL* – используется для описания терминологий;
- OWL QL – подмножество языка, разработанное для решения задач интеграции источников данных. QL основан на логике DL-Lite, предназначенной для работы с большим количеством экземпляров, и спроектированной с учетом необходимости транслировать ее выражения в язык SQL;
- OWL RL – подмножество языка для записи правил (продукций), обрабатывающих RDF-графы. Подмножество запрещает использование дизъюнкций в описаниях классов, использование отрицаний в свойствах, а также налагает ограничения на место использования некоторых конструкций (например, в правой части определений языка запрещено использование экзистенциального квантора).

Несмотря на возможность представления OWL-онтологии в синтаксисе RDF, семантика этих языков не совпадает. В OWL для описания класса предлагается множество *выражений-конструкторов*:

- пересечение нескольких определений классов (owl:intersection Of);
- объединение определений классов (owl:union Of);
- дополнение класса (owl:complement Of);
- перечисление экземпляров класса (owl:one Of);
- ограничение на значение свойства:

- owl:some Values From – экзистенциальный квантор ∃r.C;
- owl:all Values From – универсальный квантор ∀r.C.

Именованный класс OWL создается только первым способом, остальные способы создают анонимные классы. Описание класса можно дополнить также утверждениями вида owl:equivalent Class (экстенсионалы классов совпадают) и owl:disjoint With (экстенсионалы классов не пересекаются).

В OWL свойства разделены на две категории:

- объектные свойства (экземпляры класса owl:Object Property) связывают между собой экземпляры классов;
- свойства-значения (экземпляры класса owl:Datatype Property) допускают в качестве области определения только литералы.

В языке доступно множество аксиом, используемых для описания свойств:

- owl:min Cardinality и owl:max Cardinality – описание мощности отношения – указание минимального и максимального «количества» связей;
- owl:equivalent Property – свойство, позволяющее указать эквивалентные свойства-синонимы;
- owl:inverse Of – позволяет указать обратное свойство;
- owl:functional Property – класс, в который входят функциональные свойства – свойства, имеющие только одно значение для одного ресурса (например, «имеет пол» – либо только мужской, либо только женский);
- owl:symmetric Property – класс, содержащий симметричные свойства;
- owl:transitive Property – класс, содержащий транзитивные свойства.

Для описания области определения объектного свойства используются выражения owl:some Values From и owl:all Values From.

Еще одна группа аксиом предназначена для описания фактов – индивидов классов и экземпляров свойств. Помимо возможности объявить о принадлежности ресурса некоторому классу и о значении некоторого свойства, возможно определять факты об идентичности или различности объектов, описываемых различными URI.

Необходимость в подобных утверждениях следует из особенности дескриптивной логики – соглашении об «открытости мира»:

- owl:same As – объектное отношение, указывающее, что два идентификатора ссылаются на один и тот же индивид;
- owl:different From – указание, что два идентификатора ссылаются на два разных индивида;
- owl:all Different – встроенный класс, использующийся для перечисления несовместных классов.

Онтологии, записанные на перечисленных выше языках, могут быть записаны в виде множества RDF-утверждений, для хранения и обработки

которых предназначены специализированные СУБД – *«хранилища троек»* (triplestore). Для взаимодействия с подобными базами данных используется язык запросов[13] и протокол передачи[14] данных **SPARQL**. Протокол передачи основан на протоколе HTTP, и описывает схему XSD и заголовки запросов и ответов, использующиеся для взаимодействия с *точкой доступа* хранилища данных.

Язык запросов предназначен для обнаружения в RDF-графе подграфов, соответствующих описанным в запросе шаблонам, например, поиск всех имен ресурсов, являющихся столицами стран, расположенных в Европе:

```
PREFIX pgups:<http://pgups.ru/онтология#>
SELECT ?capital ?country
WHERE {
?x pgups:названиеГорода ?столица ;
pgups:столицаСтраны ?y .
?ypgups:названиеСтраны ?страна ;
pgups:находитсяНаКонтиненте pgups:Европа .
}
```

В результате выполнения запроса в RDF-графе, сохраненном в базе данных, будут найдены все узлы, имеющие связь с типом «http://pgups.ru/онтология#названиеГорода» со строковым значением, и имеющие связь с типом «http://pgups.ru/онтология#столицаСтраны» с таким узлом, у которого имеются отношения с типом «http://pgups.ru/онтология#находитсяНаКонтиненте» с узлом, имеющим идентификатор «http://pgups.ru/онтология#Европа» (рис. 3.7). В результат запроса будут помещены значения литералов с названиями городов и стран.

Запрашиваемые переменные значения помечаются символом? Для того, чтобы в запросе не повторять одинаковые последовательности символов, стоящие в начале URI, допустимо использование префиксов (в примере объявлено сокращение pgups для последовательности http://pgups.ru/онтология#. Для разработчиков запросов к данным, использующим популярные онтологии, в сети существует сайт-справочник prefix.cc[15].

---

[13] http://www.w3.org/TR/rdf-sparql-query/

[14] http://www.w3.org/TR/rdf-sparql-protocol/

[15] http://prefix.cc/

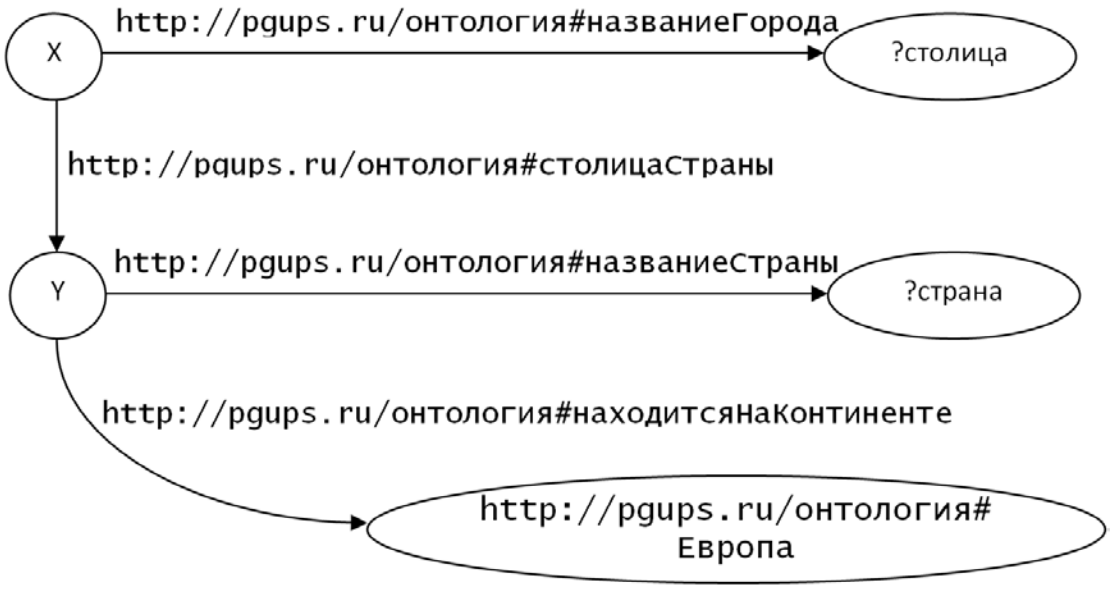


Рис. 3.7 Графовое представление шаблона запроса

Запросы SPARQL имеют четыре формы:

- SELECT – запрос строковых данных в виде таблицы (вместе с запросом точке доступа передается указание, в каком формате выдать таблицу – HTML, XML, Turtle, n3 и т. д.);
- CONSTRUCT – запрос, получающий новый сформированный RDF-подграф на основании данных, считанных из хранилища;
- ASK – запрос, возвращающий истинностное значение – true/false;
- DESCRIBE – запрос поясняющей информации, описывающей хранимые в базе данные.

Приведем пример запроса типа ASK, который может быть выполнен в точке доступа одного из крупных открытых хранилищ триплетов – dbpedia[16]: используется ли Российский рубль в качестве валюты хотя бы в трех странах:

```
ASK { { SELECT COUNT(*) as ?x WHERE {
  ?c a dbpedia-owl:Country;
dbpprop:currencyCode ?v.
FILTER( regex(?v, "RUB"))
} } FILTER(?x>= 3) }
```

Запрос возвращает true. Если выполнить запрос SELECT ?c без оператора COUNT, можно узнать, что в хранилище данных присутствуют три URI, для которых указан тип http://dbpedia.org/ontology/Country и имеется отношение currencyCode со значением RUB: http://dbpedia.org/resource/Abkhazia, http://dbpedia.org/resource/South_Ossetia и http://dbpedia.org/resource/Russia. Обратите внимание, что указанном примере запроса

[16] http://dbpedia.org/sparql

отсутствует определение префиксов dbpedia-owl (http://dbpedia.org/ontology/) и dbpprop (http://dbpedia.org/ property/), так как они присутствуют в списке предопределенных для этой точки доступа (http://dbpedia.org/sparql?nsdecl).

Возможности языка запросов SPARQL схожи с возможностями языка SQL – поддерживается фильтрация, сортировка данных, вызов функций обработки строк, чисел и дат. Версия языка SPARQL1.1 добавила возможность манипулирования данными – операции INSERT и DELETE – а также поддержку режима с логическим выводом для формирования более полного ответа на запрос.

4 ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ЭКСПЕРТНЫХ СИСТЕМ

4.1 Проблемы разработки экспертных систем

Обычно разработка экспертных систем проводится путем создания нескольких вариантов прототипов, на основе которых потом создается конечный продукт. Такой подход приемлем при создании экспертных систем в исследовательских целях, применительно к разработке коммерческих экспертных систем вряд ли его можно считать приемлемым ввиду высокой стоимости.

Процесс разработки промышленной экспертной системы можно представить в виде последовательности следующих этапов [6]:

1 Выбор проблемы.
2 Разработка прототипа ЭС.
3 Доработка до промышленной ЭС.
4 Оценка ЭС.
5 Стыковка ЭС.
6 Поддержка ЭС.

Приведенная последовательность этапов не является строго фиксированной. Более того, полученные на некотором этапе решения могут привести к необходимости пересмотра ранее принятых решений и доработке системы.

По мнению многих специалистов, затраты и время на разработку ЭС достаточно велики и не всегда оправдывают себя. В этой связи следует иметь в виду, что разработка ЭС может оказаться целесообразной в организациях, имеющих опыт по решению следующих задач обработки информации:

- создание корпоративных информационных систем;
- выполнение сложных расчетов;

- работа с компьютерной графикой;
- обработка текстов и автоматизированный документооборот.

Опыт решения подобного рода задач позволяет подготовить высококлассных специалистов в области информационных технологий и выделить задачи, требующие применения систем искусственного интеллекта от обычных алгоритмических и расчетных задач. Рассмотрим более подробно названные этапы разработки ЭС.

4.2 Выбор проблемы

На этапе выбора проблемы выполняются виды работы и деятельности, предшествующие решению о начале разработки ЭС. На этом этапе выполняется следующее [6]:

- определение проблемной области и задачи;
- поиск эксперта и определение коллектива разработчиков;
- определение предварительного подхода к решению проблемы;
- предварительная оценка расходов и доходов;
- подготовка подробного плана разработки.

Выбор подходящей проблемы во многом определяет эффективность разрабатываемой ЭС. Выбор проблемы должен быть таким, что разрабатываемая экспертная система действительно приводила к экономии при решении прикладной задачи и смогла окупить затраты на ее разработку.

При выборе проблемной области применения следует учитывать, что для работы с четко формулируемыми знаниями, требующими вычислительной обработки, не требуется разрабатывать экспертные системы, а лучше использовать обычные алгоритмические программы. Для работы с реляционными базами данных и электронными таблицами также целесообразно использовать соответствующие программные средства: системы управления базами данных и текстовые процессоры соответственно.

Экспертные системы имеет смысл разрабатывать в случае, если результативность задачи в рассматриваемой предметной области существенно зависит от знаний, носящих субъективный, изменяющийся, символьный или выводимый из соображений здравого смысла характер.

Обычно экспертные системы разрабатываются путем получения знаний от эксперта и ввода их в систему. При разработке и последующем расширении системы эксперт и инженер по знания работают вместе. Инженер по знаниям помогает структурировать знания, определить и формализовать понятия

и правила. В коллектив разработчиков целесообразно включить потенциальных пользователей и профессиональных программистов. Предварительный подход к программной реализации задачи определяется исходя из характеристик задачи и ресурсов, выделенных на ее решение. Инженер по знаниям выдвигает обычно несколько вариантов с учетов имеющихся программных инструментальных средств. Окончательный выбор осуществляется на этапе разработки прототипа.

После определения задачи подсчитываются расходы и прибыль от разработки экспертной системы. В расходы включаются затраты на оплату труда коллектива разработчиков. В дополнительные расходы включается стоимость программного инструментария, с помощью которого ведется разработка ЭС. Прибыль может быть получена путем снижения цены продукции, повышения производительности труда, расширения номенклатуры продукции и услуг, разработки новых видов продукции или услуг в области, в которой будет использоваться ЭС. Расходы и прибыль рассчитываются в привязке ко времени, в течение которого возвращаются затраченные на разработку средства.

Далее инженер по знаниям рассматривает ответы на следующие вопросы:

- возможность решения задачи с помощью ЭС;
- ЭС можно разработать с помощью доступных программных средств;
- имеется подходящий эксперт;
- предложенные критерии производительности можно удовлетворить;
- затраты и срок их окупаемости приемлемы заказчику.

При положительных ответах на эти вопросы инженер по знаниям составляет план разработки. В нем определяются шаги процесса разработки, необходимые затраты и ожидаемые результаты.

4.3 Технология быстрого прототипирования

Прототип системы представляет собой усеченную версию ЭС, предназначенную для проверки правильности кодирования фактов связей и стратегий рассуждения эксперта. Она дает возможность инженеру по знаниям привлечь эксперта к участию в процессе разработки ЭС. Обычно в состав прототипа входят несколько десятков правил, фреймов или примеров.

В создании прототипа можно выделить шесть основных стадий, приведенных в табл. 4.1, здесь же показан состав участников для каждой из указанных стадий и его результат. Приведенные в табл. 4.1 стадии

приближенно отражают реальный итерационный процесс разработки прототипа.

Таблица 4.1 **Стадии разработки прототипа ЭС**

Этап	Результат	Участники
Идентификация проблемы	Проблема	эксперт, инженер по знаниям, пользователь
Получение знаний (дополнительное извлечение)	Знания	эксперт, инженер по знаниям
Структурирование путем изменения полей	Поле знаний	инженер по знаниям
Формализация и переформализация	БЗ на языке представления знаний	инженер по знаниям, программист
Реализация прототипа (перепрограммирование)	Программа-прототип ЭС	программист
Тестирование		инженер по знаниям, эксперт, пользователь, программист

Дадим краткую характеристику каждому из этапов разработки прототипа ЭС.

Идентификация проблемы представляет собой знакомство и обучение членов коллектива разработчиков, а также создание неформальной формулировки проблемы Она включает уточнение задачи, планирование хода разработки прототипа экспертной системы. Средняя продолжительность этапа составляет 1-2 недели.

При этом определяется следующее:

- требуемые ресурсы (время, люди, ЭВМ и т. д.);
- источники знаний (книги, дополнительные эксперты, методики);
- имеющиеся аналогичные ЭС;
- цели (распространения опыта, автоматизация рутинных действий и др.);
- классы решаемых задач и т. д.

Структурирование или концептуализация знаний означает разработку неформального описания знаний о предметной области в виде графа, таблицы, диаграммы или текста, отражающего основные концепции и взаимосвязи между понятиями предметной области.

Она подразумевает выявление структуры знаний о полученной предметной области, при этом определяется следующее:

- терминология;
- список основных понятий и их атрибутов;
- отношения между понятиями;
- структура входной и выходной информации;
- стратегия принятия решений;
- ограничения стратегий и т. д.

Такое описание называется полем знаний. Средняя продолжительность этапа составляет 2-4 недели.

Формализация знаний представляет собой разработку базы знаний на языке представления знаний, который, с одной стороны, соответствует структуре поля знаний, а с другой – позволяет реализовать прототип системы на следующей стадии программной реализации. При формализации строится формализованное представление концепций предметной области на основе выбранного языка представления знаний.

Традиционно на этом этапе используются следующие методы и модели:

- логические методы (исчисления предикатов 1-го порядка и др.);
- продукционные модели с прямым и обратным выводом;
- семантические сети;
- фреймы;
- объектно-ориентированные языки.

Все чаще на этой стадии используется симбиоз языков представления знаний. Средняя продолжительность этапа составляет 1-2 месяца.

Реализация представляет собой разработку программного комплекса – прототипа, демонстрирующего жизнеспособность подхода в целом. Создание прототипа ЭС выполняется с помощью таких же средств, используемых для разработки самой ЭС, перечень средств мы называли в начале дисциплины. Средняя продолжительность этапа 1-2 месяца.

Тестирование представляет собой выявление ошибок в подходе и реализации прототипа и выработку рекомендаций по доводке системы до промышленного варианта. При тестировании проверяется работа программ прототипа с целью приведения в соответствие с реальными запросами пользователей.

Прототип проверяется по следующим вопросам:

- удобство и адекватность интерфейсов ввода/вывода (характер вопросов в диалоге, связность выводимого текста результата и др.);
- эффективность стратегии управления (порядок перебора, использование нечеткого вывода и др.);

- качество проверочных примеров;
- корректность базы знаний (полнота и непротиворечивость правил).

Средняя продолжительность этапа составляет 1-2 недели.

4.4 Развитие прототипа допромышленной экспертной системы

При неудовлетворительном функционировании прототипа эксперт и инженер по знаниям имеют возможность оценить, что именно будет включено в разработку окончательного варианта системы. Если первоначально выбранные объекты или свойства оказываются неподходящим, их требуется изменить. Возможно получение оценки общего числа эвристических правил, необходимых для создания окончательного варианта экспертной системы.

Иногда при разработке промышленной или коммерческой системы выделяют следующие дополнительные этапы для перехода:

- демонстрационный прототип – система решает часть задач, демонстрируя жизнеспособность подхода (несколько десятков правил или понятий);
- исследовательский прототип – система решает большинство задач, но неустойчива в работе и не полностью проверена (несколько сотен правил или понятий);
- действующий прототип – система надежно решает все задачи на реальных примерах, но требует много времени и памяти;
- промышленная система – система обеспечивает высокое качество решений при минимизации требуемого времени памяти (переписывается с использованием более эффективных средств представления знаний);
- коммерческая система – промышленная система, пригодная к продаже, хорошо документирована и снабжена сервисом.

Чаще реализуется плавный переход от демонстрационного прототипа к промышленной системе. Если программный инструментарий был выбран удачно, не обязательно даже переписывать окончательный вариант другими программными средствами,

Понятие коммерческой системы в нашей стране входит в понятие «промышленный программный продукт», или «промышленная ЭС».

Основная работа на данном этапе заключается в существенном расширении базы знаний, то есть в добавлении большого числа дополнительных правил, фреймов, узлов семантической сети или других элементов знаний. Эти элементы знаний обычно увеличивают глубину системы, обеспечивая большее число правил для трудно уловимых аспектов отдельных случаев. В то же время

эксперт и инженер по знаниям могут увеличить базу знаний системы, включая правила, управляющие дополнительными подзадачами или дополнительными аспектами экспертной задачи (метазнания).

После установления основной структуры ЭС знаний инженер по знаниям приступает к разработке и адаптации интерфейсов, с помощью которых система будет общаться с пользователем и экспертом. Необходимо обратить внимание на языковые возможности интерфейсов, их простоту и удобство для управления работой ЭС. Система должна обеспечивать пользователю возможность легким и естественным образом уточнять непонятные моменты, приостанавливать работу и т. д. В частности, могут оказаться полезными графические представления.

На этом этапе разработки большинство экспертов узнают достаточно о вводе правил, и могут сами вводить в систему новые правила. Таким образом, начинается процесс, во время которого инженер по знаниям передает контроль за системой эксперту для уточнения, детальной разработки и обслуживания.

4.5 Оценка, стыковка и поддержка системы

После завершения разработки промышленной экспертной системы нужно провести ее оценку путем тестирования по критериям эффективности. К тестированию привлекаются другие эксперты для проверки работоспособности системы на различных примерах. Экспертные системы оцениваются главным образом для проверки точность работы программы и ее полезности.

Оценку можно проводить, исходя из различных критериев, которые сгруппируем следующим образом [6]:

- критерии пользователей (понятность и «прозрачность» работы системы, удобство интерфейсов и др.);
- критерии экспертов (оценка советов-решений, предлагаемых системой, сравнение ее с собственными решениями, оценка подсистемы объяснений и др.);
- критерии коллектива разработчиков (эффективность реализации, производительность, время отклика, дизайн, широта охвата предметной области, непротиворечивость БЗ, количество тупиковых ситуаций, анализ чувствительности программы к незначительным изменениям в представлении знаний, весовых коэффициентах, применяемых в механизмах логического вывода, данных и т. п.).

На этапе стыковки осуществляется *стыковка* экспертной системы с другими программными средствами в среде, в которой она будет работать, и обучение обслуживаемых пользователей системы. Иногда это означает внесение существенных изменений. Такие изменения требуют вмешательства инженера по знаниям или какого-либо другого специалиста, который сможет модифицировать систему. Под стыковкой подразумевается также разработка связей между экспертной системой и средой, в которой она действует.

Когда экспертная система уже готова, инженер по знаниям должен убедиться в том, что эксперты, пользователи и персонал знают, как эксплуатировать и обслуживать ее. После передачи им своего опыта в области информационной технологии инженер по знаниям может полностью предоставить ее в распоряжение пользователей.

Для подтверждения полезности системы важно предоставить каждому из пользователей возможность поставить перед ЭС реальные задачи, а затем проследить, как она их решает. Чтобы система была одобрена, нужно представить ее как помощника, освобождающего пользователей от обременительных задач, а не как средство их замещения.

Стыковка включает обеспечение связи ЭС с существующими базами данных и другими системами на предприятии, а также улучшение системных факторов, зависящих от времени, чтобы можно было обеспечить ее более эффективную работу и улучшить характеристики ее технических средств, если система работает в необычной среде (например, связь с измерительными устройствами).

При перекодировании системы на язык, подобный С, повышается ее быстродействие и увеличивается переносимость, однако гибкость при этом уменьшается. Это приемлемо в случае, если система сохраняет все знания проблемной области, и это знание не будет изменяться в ближайшем будущем. Если экспертная система создана именно из-за того, что проблемная область изменяется, то необходимо *поддерживать* систему в ее инструментальной среде разработки.

5 ОСНОВЫ ТЕОРИИ НЕЧЕТКИХ МНОЖЕСТВ

В системах искусственного интеллекта зачастую приходится иметь дело с нечеткими знаниями, которые нельзя интерпретировать как полностью истинные или полностью ложные. При выводе решения в продукционных системах с нечеткими знаниями требуется решать проблемы представления

нечетких знаний и использования их в различных алгоритмах нечеткого вывода.

5.1 Нечеткие множества

При попытке формализовать человеческие знания возникла проблема, затруднявшая использование традиционного математического аппарата для их описания. Существует класс описаний, использующих качественные характеристики объектов (много, мало, сильно, слабо, очень сильно и т. п.). Эти характеристики обычно размыты, и их нельзя интерпретировать однозначно. В то же время они содержат важную информацию, например: «Одним из возможных признаков неисправности двигателя является низкое давление масла».

С другой стороны, в системах искусственного интеллекта часто пользуются неточными знаниями, которые нельзя интерпретировать как полностью истинные или полностью ложные. Существуют знания, достоверность которых выражается некоторым коэффициентом, например, 0,5.

Для формального представления таких знаний с учетом свойств их размытости и неточности американским ученым Лотфи Заде предложен математический аппарат нечеткой алгебры и нечеткой логики. Это направление получило широкое распространение, положив начало одной из ветвей теории искусственного интеллекта, именуемой мягкими вычислениями.

Неформально нечеткое множество можно определить как набор элементов произвольной природы, относительно которых нельзя однозначно утверждать, принадлежат ли отдельные элементы этого набора заданному множеству или нет.

Нечеткое множество *A*, формально определяется как множество упорядоченных пар $<x, \mu_A(x)>$, где x есть элемент универсального множества *X*, $x \in X$, $\mu_A(x)$ – функция принадлежности, принимающая значения на интервале [0, 1]. Эта функция указывает степень принадлежности элемента x нечеткому множеству *A*.

В частности, значение $\mu_A\{x\}=1$ означает, что элемент x точно принадлежит нечеткому множеству *A*, а значение $\mu_A\{x\}=0$, наоборот указывает на то, что элемент x точно не принадлежит нечеткому множеству *A*.

Конечные нечеткие множества обычно записывают в виде

$$A=\{<x_1, \mu_A(x_1)>, <x_2, \mu_A(x_2)>,\ldots, <x_n, \mu_A(x_n)>\}.$$

Кроме того, часто используют и другие варианты задания конечных нечетких множеств, например такой, как

$$A=\{<x_1/\mu_A(x_1)>+<x_2/\mu_A(x_2)>+,\ldots,+<x_n/\mu_A(x_n)>\}.$$

Здесь косая черта обозначает разделитель, а символ «+» служит для обозначения теоретико-множественного объединения отдельных элементов.

Пример. Нечеткое множество «юный» можно определить следующим образом:

«юный» = {11/0,6 +12/0,8+13/1+14/1+15/0,9+16/0,7+17/0,4+18/0,2}.

Такое определение нечеткого множества некоторым экспертом означает, что он с высокой степенью уверенности относит ребенка в возрасте 13-14 лет к юному ($\mu_A(x)=1$). Человека в возрасте 11-12 лет и 15-16 лет также относят к юному с меньшей степени уверенности ($0,6\leq \mu_A(x)\leq 0,9$), в возрасте 17-18 лет его называют юным достаточно редко. Таким образом, нечеткие множества позволяют учитывать разброс индивидуальных мнений.

Нечеткие множества можно задавать *графически*. Например, нечеткое множество «юный» графически можно задать, как показано на рис. 5.1*a* и *b*).

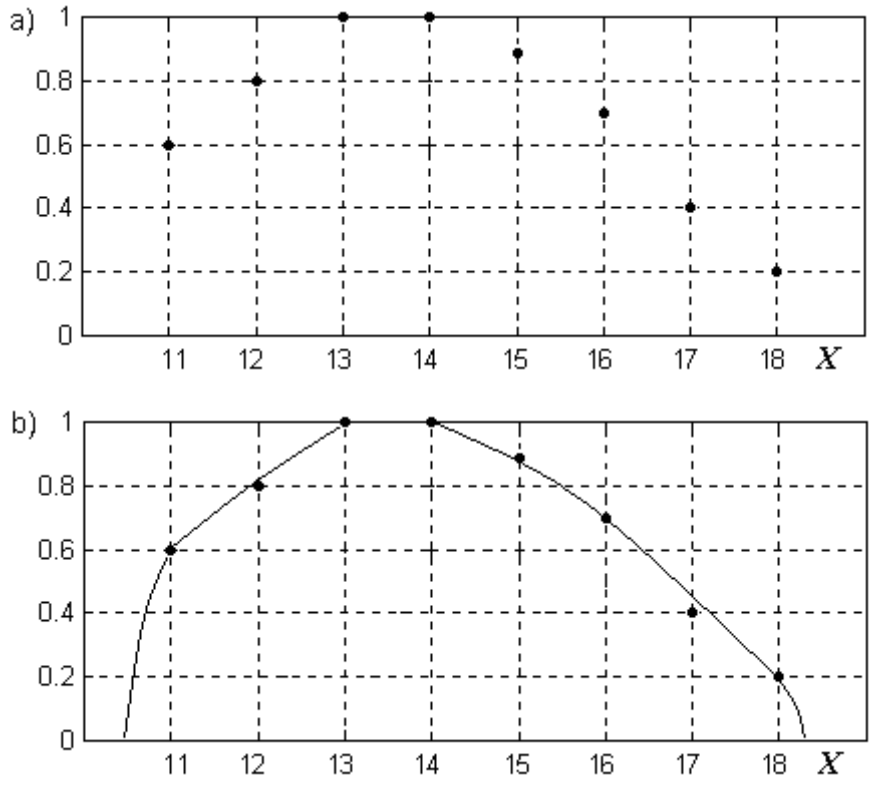


Рис. 5.1 Графическое представление нечеткого множества «юный»

Как видим на рис. 5.1, нечеткое множество можно задать с помощью дискретного (a) или непрерывного (b) графика функции принадлежности $\mu(x)$.

5.2 Определение функций принадлежности

Для определения функций принадлежности нечетких множеств могут быть использованы прямые и косвенные методы.

При использовании *прямых методов* эксперт либо просто задает для каждого $x \in E$ значение $\mu_A(x)$, либо определяет функцию совместимости. Прямые методы задания функции принадлежности обычно используются для измеримых понятий, таких как скорость, время, расстояние, температура и т. д., или когда выделяются полярные значения.

Во многих задачах при характеристике объекта можно выделить набор признаков и для каждого из них определить полярные значения, соответствующие значениям функции принадлежности, 0 или 1.

Например, в задаче определения характеристик и субхарактеристик качества программного обеспечения (согласно международному стандарту ISO 9126-1) можно выделить шкалы, приведенные в табл. 5.1.

Таблица 5.1 **Шкалы в задаче оценки характеристик качества ПО**

	характеристика	0	1
x_1	функциональные возможности	узкие	широкие
x_2	функциональная пригодность	низкая	высокая
x_3	правильность (корректность)	неполная	полная
x_4	способность к взаимодействию	слабая	сильная
x_5	защищенность	плохая	хорошая
x_6	надежность	низкая	высокая
x_7	практичность (применимость)	низкая	высокая
x_8	сопровождаемость	плохая	хорошая
x_9	мобильность	низкая	высокая

Для конкретного программного продукта A эксперт, исходя из приведенной шкалы, задает $\mu_A(x) \in [0; 1]$, формируя векторную функцию принадлежности $\{\mu_A(x_1), \mu_A(x_2), \ldots \mu_A(x_9)\}$.

При прямых методах используются также групповые прямые методы, когда, например, группе экспертов предъявляют конкретный программный продукт

и каждый должен дать один из двух ответов: «это ПО надежное » или «это ПО не надежное», тогда количество утвердительных ответов, деленное на общее число экспертов, дает значение $\mu_{надежное}$ (ПО).

Косвенные методы определения значений функции принадлежности используются в случаях, когда нет элементарных измеримых свойств, через которые определяется нечеткое множество. Как правило, это методы попарных сравнений. Если бы значения функций принадлежности были бы нам известны, например, $\mu_A(x_i) = \omega_i$, $i = 1, 2, ..., n$, то попарные сравнения можно было бы представить матрицей отношений $A=\{a_{ij}\}$, где $a_{ij}= \omega_i/ \omega_j$.

На практике эксперт формирует матрицу А, при этом предполагается, что диагональные элементы равны 1, а для элементов, симметричных относительно диагонали, $a_{ij} =1/a_{ij}$, т. е. если один элемент оценивается в *а* раз сильнее чем другой, то этот последний должен быть в *а* раз сильнее, чем первый. В общем случае задача сводится к поиску вектора w, удовлетворяющего уравнению вида $\mathbf{Aw} = \lambda_{max}\backslash\mathbf{w}$, где λ_{max} – наибольшее собственное значение матрицы **А**. Поскольку матрица **А** положительна по построению, решение данной задачи существует и является положительным.

Можно отметить еще два подхода:

• *использование относительных частот* по данным эксперимента в качестве значений функции принадлежности;

• *использование типовых форм* кривых для задания функций принадлежности с уточнением их параметров в соответствии с данными эксперимента.

5.3 Виды функций принадлежностей

Обычно используются следующие типовые формы функций принадлежности нечетких множеств: треугольная (trimf), трапецеидальная (trapmf), гауссова (gaussmf), двойная гауссова, обобщенная колоколообразная, сигмоидальная, двойная сигмоидальная, Z-функция, S-функция, Pi-функция.

Конкретный вид функций принадлежности определяется значениями параметров их аналитического представления

В работе [4] для реализации нечеткого вывода решения о том, как реагировать на факт обнаружения программных воздействий на автоматизированные системы (АС), с помощью экспертных оценок задается база правил нечеткого вывода системы поддержки принятия решений (СППР).

При формировании простых нечетких высказываний в предпосылках и заключениях продукционных правил необходимо задать функции принадлежностей (ФП) соответствующих нечетких множеств. Для удобства экспертам предлагается набор ФП, покрывающих по базовым множествам пространства входных и выходных переменных, экспертам требуется скорректировать числовые параметры ФП (табл. 5.2) и при необходимости их количество.

Для применения в СППР в качестве основной ФП предложено использовать обобщенную колоколообразную ФП. Такой выбор ФП объясняется наличием всего трех параметров, что позволяет сократить время настройки системы нечеткого вывода, а наличие коэффициента лингвистического модификатора позволяет автоматизировано корректировать правила при наличии ошибок второго рода.

Обобщенная колоколообразная ФП описывается формулой [15, 25]:

$$A_i(u_j) = \frac{1}{1+(\frac{u_j-b_i}{a_i})^{2s}}\ , \qquad (5.1)$$

где a_i – половина λ-сечения нечеткого терм-множества;

b_i – координата максимума терм-множества (середина ядра терм-множества);

u_j – входные данные для переменной A;

j – порядковый номер переменной;

$A_i(u_j)$ – функция принадлежности A_i заключения правила n;

s – коэффициент ядра терм-множества (коэффициент лингвистического модификатора).

При необходимости построения обратной функции в формуле (5.1) нужно изменить s на $-s$.

На рис. 5.2 показан внешний вид обобщенной колокообразной ФП и ее параметры (формула (5.1)).

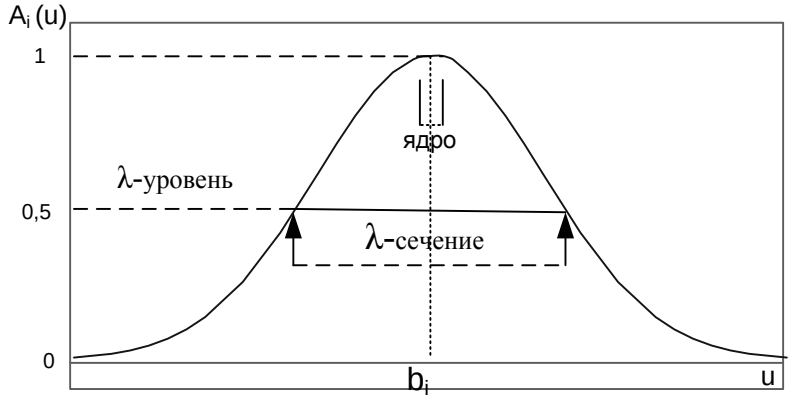


Рис. 5.2 Вид обобщенной колокообразной ФП

В качестве дополнительной в СППР предлагается использовать сигмоидальную ФП (рис. 5.3). Ее применяют тогда, когда нельзя заранее точно определить координату максимума терм-множества A_i (например, максимальное количество неудачных попыток входа в систему). Сигмоидальная ФП описывается формулой

$$A_i(u_j) = \frac{1}{1+e^{-s\,(u_j - h_i)}}, \qquad (5.2)$$

где: h_i – координата пересечения λ-сечения нечеткого терм-множества и функции принадлежности $A_i(u_j)$.

Для наблюдаемых состояний защищаемой АСУ функциями принадлежности входных переменных являются [4]:

$A_i(u_1)$ – количество попыток неудачного входа в систему за последний час;

$B_i(u_2)$ – количество попыток неудачного входа в систему за последние 24 часа;

$C_i(u_3)$ – средняя нагрузка на сетевой интерфейс (в % от максимальной нагрузки за последнюю минуту);

$D_i(u_4)$ – соответствие номеров открытых портов «эталону» (u_{04});

$E_i(u_5)$ – соответствие имен выполняемых процессов, пути к файлам и их контрольных сумм «эталону» (u_{05});

$F_i(u_6)$ – соответствие текущего времени суток рабочему (u_{06}).

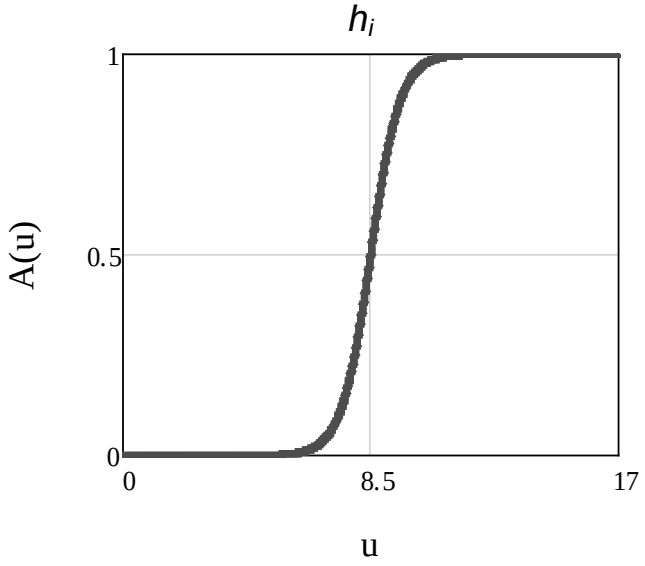


Рис. 5.3 Вид сигмоидальной ФП

Функцией принадлежности выходной переменной y является $G_i(y)$ – степень уверенности в наличии ПВ.

В табл. 5.2 приведены указанные выше переменные, соответствующие им типы ФП, лингвистические термы (ЛТ) и другие параметры для построения ФП СППР [4]

Таблица 5.2 **Основные параметры ФП СППР**

ФП	№ ЛТ	Тип формулы	ЛТ	Параметры				
				u_i	a_i	b_i	s_i	h_i
$A_i(u_1)$	1	(2)	незначительное	$0 \le u_1 \le 3.5$	3.5	0	2	-
	2	(2)	значительное	$3.5 \le u_1 \le 8.5$	2.5	6	2	-
	3	(3)	потенциально опасное	$8.5 \le u_1$	-	-	2	8.5
$B_i(u_2)$	1	(2)	незначительное	$0 \le u_2 \le 3.5$	3.5	0	2	-
	2	(2)	ниже среднего	$3.5 \le u_2 \le 6.5$	1.5	5	2	-
	3	(2)	выше среднего	$6.5 \le u_2 \le 9.5$	1.5	8	2	-
	4	(3)	потенциально опасное	$9.5 \le u_2$	-	-	2	9.5
$C_i(u_3)$	1	(2)	очень низкая	$0 \le u_3 \le 0.1$	0.1	0	2	-
	2	(2)	низкая	$0.1 \le u_3 \le 0.2$	0.05	0.15	2	-
	3	(2)	ниже среднего	$0.2 \le u_3 \le 0.4$	0.1	0.3	2	-
	4	(2)	выше среднего	$0.4 \le u_3 \le 0.6$	0.1	0.5	2	-
	5	(2)	высокая	$0.6 \le u_3 \le 0.8$	0.1	0.7	2	-
	6	(2)	очень высокая	$0.8 \le u_3 \le 1$	0.2	1	2	-
$D_i(u_4)$	1	(2)	соответствует	$0 \le u_4 \le 0.5$	0.5	0	4	-
	2	(2)	не соответствует	$0.5 \le u_4 \le 1$	0.5	1	4	-
$E_i(u_5)$	1	(2)	соответствует	$0 \le u_5 \le 0.5$	0.5	0	4	-
	2	(2)	не соответствует	$0.5 \le u_5 \le 1$	0.5	1	4	-
$F_i(u_6)$	1	(2)	соответствует	$9 \le u_6 \le 18$	0.5	0	3	-
	2	(2)	не соответствует	$18 \le u_6 \le 9$	0.5	1	-3	-
$G_i(y)$	1	(2)	очень низкая	$0 \le y \le 0.1$	0.1	0	2	-
	2	(2)	низкая	$0.1 \le y \le 0.25$	0.075	0.175	2	-
	3	(2)	ниже среднего	$0.25 \le y \le 0.5$	0.125	0.375	2	-
	4	(2)	выше среднего	$0.5 \le y \le 0.75$	0.125	0.625	2	-
	5	(2)	высокая	$0.75 \le y \le 0.9$	0.075	0.825	2	-
	6	(2)	очень высокая	$0.9 \le y \le 1$	0.1	1	2	-

На рис. 5.4 показан вид функций принадлежности $G_i(y)$ выходной переменной y, построенных в соответствии с заданными параметрами табл. 5.2.

Таким образом,

правило № 1 имеет вид

ЕСЛИ u_1 есть $A_1(u_1)$ **И** u_2 есть $B_1(u_2)$ **И** u_3 есть $C_1(u_3)$ **И** u_4 есть $D_1(u_4)$ **И** u_5 есть $E_1(u_5)$ **ТО** y есть $G_1(y)$; (5.3)

правило № 2 имеет вид

ЕСЛИ u_1 есть $A_1(u_1)$ **И** u_2 есть $B_1(u_2)$ **И** u_3 есть $C_2(u_3)$ **И** u_4 есть $D_1(u_4)$ **И** u_5 есть $E_1(u_5)$ **И** u_6 есть $F_1(u_6)$ **ТО** y есть $G_1(y)$. (5.4)

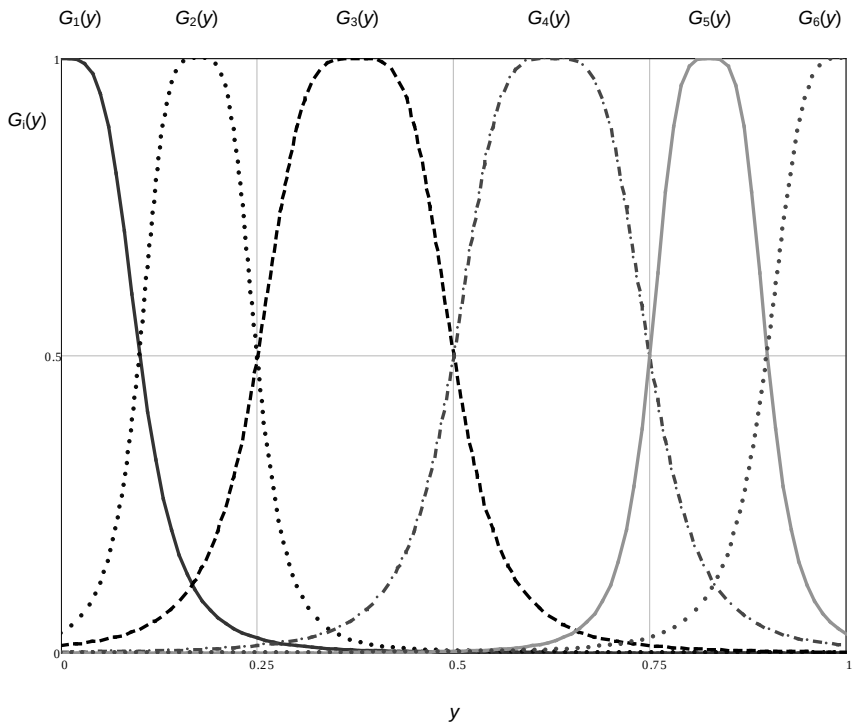


Рис. 5.4 Функции принадлежности $G_i(y)$ выходной переменной y

В правиле 1 (формула (5.3)) после оптимизации отсутствует предпосылка « u_6 есть $F_i(u_6)$», так как в любое время суток при любых входных данных (u_6) заключение правила одинаковое – «ТО y есть $G_1(y)$».

5.4 Операции над нечеткими множествами

Подчеркнем, что нечеткие множества являются обобщением обычных четких множеств. Поэтому любое определение некоторой операции над нечеткими множествами должно быть справедливым в случае, когда вместо нечетких множеств используются обычные множества. Для обеспечения возможности сравнения нечетких множеств и выполнения над ними различных операций соответствующие нечеткие множества должны быть определены на одном и том же универсуме.

Прежде всего, определим следующие два простейших отношения между нечеткими множествами.

Равенство. Два нечетких множества A={x, μA(x)} и B={x, μ_B(x)} считаются равными (A=B), если их функции принадлежности принимают равные значения на универсуме X:

$$\mu_A(x) = \mu_B(x) \text{ для любого } x \in X. \qquad (5.5)$$

Нечеткое подмножество. Нечеткое множество $A=\{x, \mu_A(x)\}$ является *нечетким подмножеством* нечеткого множества $B=\{x, \mu_B(x)\}$ (обозначают $A \subseteq B$) тогда и только тогда, когда выполняется следующее условие:

$$\mu_A(x) \le \mu_B(x) \qquad (\forall x \in \boldsymbol{X}). \tag{5.6}$$

Говорят, что нечеткое множество B *доминирует* нечеткое множество A, а нечеткое множество A *содержится* в нечетком множестве B. Нечеткое множество A называют также *несобственным подмножеством* множества B.

Если в определении нечеткого подмножества исключается равенство соответствующих нечетких множеств, то A называется *собственным* нечетким подмножеством B и обозначается: $A \subset B$. При этом нечеткое множество B с*трого* доминирует нечеткое множество A, а нечеткое множество A *строго* содержится в нечетком множестве B.

Приведем определения нескольких важных *логических операций* над нечеткими множествами.

Пересечением двух нечетких множеств A и *B* называют нечеткое множество C ($C=A \cap B$), заданное на этом же универсуме $\boldsymbol{X}$, функция принадлежности которого определяется по формуле

$$\mu_C(x) = \min\{\mu_A(x), \mu_B(x)\} \qquad (\forall x \in \boldsymbol{X}). \tag{5.7}$$

Пересечение $A \cap B$ есть наибольшее нечеткое подмножество C, которое содержится одновременно в нечетких множествах A и B.

Операцию пересечения нечетких множеств называют также min-пересечением или $\wedge$-пересечением (по определению логической операции «И», обозначаемой знаком «$\wedge$»).

Объединением двух нечетких множеств A и *B* называют нечеткое множество C ($D = A \cup B$), заданное на этом же универсуме $\boldsymbol{X}$, функция принадлежности которого определяется по формуле

$$\mu_D(x) = \max\{\mu_A(x), \mu_B(x)\} \quad (\forall x \in \boldsymbol{X}). \tag{5.8}$$

Объединение $A \cup B$ есть наименьшее нечеткое множество D, которое доминирует одновременно A и B. Операцию объединения нечетких множеств называют max- объединением или $\vee$-объединением (по определению логической операции «ИЛИ», обозначаемой знаком «$\vee$»).

Заметим, что эта же операция в терминах вероятностного подхода задается в виде

$$\mu_D(x) = \mu_A(x) + \mu_B(x) - \mu_A(x) * \mu_B(x).$$

Разностью двух нечетких множеств A и *B* называется некоторое третье нечеткое множество S (обозначается $S = A \backslash B$), заданное на этом же универсуме $\boldsymbol{X}$, функция принадлежности которого определяется по формуле

$$\mu_S(x) = \max\{\mu_A(x) - \mu_B(x), 0\} \qquad (\forall x \in \boldsymbol{X}), \tag{5.9}$$

где используется операция арифметической разности двух чисел.

При построении нечетких моделей сложных систем широко используются унарные операции умножения нечеткого множества на число и возведение нечеткого множества в степень.

Умножение нечеткого множества на число. Пусть A = $\{x, \mu_A(x)\}$; A – произвольное нечеткое множество, заданное на универсуме $\boldsymbol{X}$; a – положительное действительное число, такое, что $a \cdot h_A \leq 1$ (h_A – высота нечеткого множества A).

Результат операции *умножения нечеткого множества* A *на число* a определяется как нечеткое множество B = $\{x, \mu_B(x)\}$, заданное на этом же универсуме $\boldsymbol{X}$, функция принадлежности которого определяется по формуле

$$\mu_B(x)=a\cdot\mu_A(x) \quad (\forall x\in\boldsymbol{X}). \tag{5.10}$$

Эту операцию в дальнейшем будем обозначать через $a\cdot$A.

Возведение в степень. Пусть A = $\{x, \mu_A(x)\}$; A – произвольное нечеткое множество, заданное на универсуме $\boldsymbol{X}$; k – положительное действительное число. В этом случае формально можно определить операцию *возведения* нечеткого множества A *в степень* k как нечеткое множество B = $\{x, \mu_B(x)\}$, заданное на этом же универсуме $\boldsymbol{X}$, функция принадлежности которого определяется по формуле

$$\mu_B(x) = \mu_A(x)^k \quad (\forall x\in\boldsymbol{X}). \tag{5.11}$$

Примеры графического представления операции возведения нечеткого множества в степень приведены на рис. 5.5.

На основе операции возведения в степень определяются две специальные операции над нечеткими множествами: операция концентрирования и операция растяжения нечеткого множества.

Концентрирование. Пусть на универсуме $\boldsymbol{X}$ задано произвольное нечеткое множество A = $\{x, \mu_A(x)\}$.

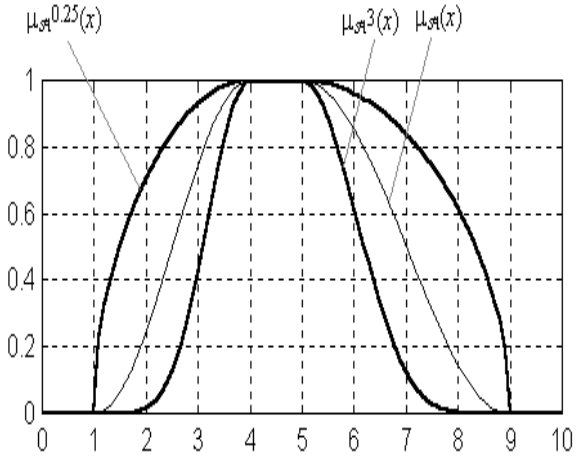


Рис. 5.5 Представление операций возведения в степень

Операция *концентрирования*, обозначаемая через CON (A), дает в результате нечеткое множество C = {x, $\mu_C(x)$}, функция принадлежности которого:

$$\mu_C(x) = \mu_A(x)^2 \qquad (\forall x \in \boldsymbol{X}). \qquad (5.12)$$

Очевидно, в этом случае CON (A) = A^2.

Например, для конечного нечеткого множества A = {<1, 1.0>, <2, 1.0>, <3, 0.9>, <4, 0.8>, <5, 0.6>, <6, 0.5>, <7, 0.4>, <8, 0.2>, <9, 0.1>} его концентрирование равно CON(A) = A^2 = {<1, 1.0>, <2, 1.0>, <3, 0.81>, <4, 0.64>, <5, 0.36>, <6, 0.25>, <7, 0.16>, <8, 0.04>, <9, 0.01>}.

Растяжение. Операция *растяжения*, обозначаемая через DIL (A), дает в результате нечеткое множество

$$D = \{x, \mu_D(x)\},$$

функция принадлежности которого:

$$\mu_D(x) = \mu_A(x)^{0.5} \qquad (\forall x \in \boldsymbol{X}). \qquad (5.13)$$

С помощью операций концентрирования и растяжения выполняется усиление и ослабление лингвистических понятий соответственно.

В частности, с помощью операции концентрирования можно задать модификатор «ОЧЕНЬ» для некоторого лингвистического понятия, а с помощью операции *растяжения* задается модификатор «СРАВНИТЕЛЬНО» или «БОЛЕЕ МЕНЕЕ».

Например, если некоторое понятие, скажем, «старый возраст», определяется как A = <x, $\mu_A(x)$>, тогда понятие «очень старый возраст» определяется так:

$$\text{CON (A)} = A^2 = \langle x, \mu_A(x)^2 \rangle.$$

5.5 Нечеткие и лингвистические переменные

Понятия нечеткой и лингвистической переменных используются при задании входных и выходных переменных в системах управления с использованием аппарата нечеткой логики.

Нечеткая переменная задается кортежем <α, X, A>, где α – имя переменной; X – область определения этой переменной (универсум); A = <x, μ_A(x)> – нечеткое множество на X, описывающее возможные значения нечеткой переменной.

Лингвистическая переменная представляет собой переменную, значение которой определяется набором вербальных (словесных) характеристик некоторого свойства. Например, лингвистическая переменная «давление»

определяется через набор {очень низкое, низкое, среднее, высокое, очень высокое}.

Значения лингвистической переменной (ЛП) определяются через нечеткие множества, которые, в свою очередь, определяются на некотором базовом наборе значений или базовой числовой шкале, имеющей размерность. Каждое значение ЛП определяется как нечеткое множество, например, нечеткое множество «высокое давление».

Лингвистической переменной называется набор $<\beta, T, X, G, M>$, где β – наименование лингвистической переменной; T – базовое множество (терм-множество) значений (термов) лингвистической переменной, представляющих собой наименования отдельных нечетких переменных α; G – синтаксическая процедура, позволяющая оперировать элементами терм-множества T, в частности, генерировать новые термы (значения) $G(T)$. При этом множество $T \cup G(T)$ называется *расширенным терм-множеством* лингвистической переменной; M – семантическая процедура, позволяющая преобразовать каждое новое значение лингвистической переменной, образуемое процедурой G, в нечеткую переменную, т. е. сформировать соответствующее нечеткое множество.

Пример. Пусть эксперт определяет температуру жидкости с помощью понятий «Малая температура», «Средняя температура» и «Большая температура», при этом минимальная температура равна 0^o, а максимальная – 100^o.

Формализация такого описания может быть проведена с помощью лингвистической переменной $<\beta, T, X, G, M>$, где β – температура жидкости; T – {«Малая температура», «Средняя температура», «Большая температура»}; X –[0, 100]; G – процедура образования новых термов с помощью связок «и», «или» и модификаторов типа «очень», «не», «слегка» и т. п. Например, «Малая или средняя температура», «Очень малая температура» и т. д. ; M – процедура задания на X = [0, 100] нечетких подмножеств A_1 = «Малая температура», A_2 = «Средняя температура», A_3 = «Большая температура», а также нечетких множеств для термов из $G(T)$ в соответствии с правилами трансляции нечетких связок и модификаторов «и», «или», «не», «очень», «слегка» и других операций над нечеткими множествами вида: $A \cap B$, $A \cup B$, $\bar{A}$, $CONA = A^2$, $DILA = A^{0,5}$ и т. п.

Замечание. Наряду с рассмотренными выше базовыми значениями лингвистической переменной «температура» (Т = {«Низкая температура», «Средняя температура», «Высокая температура»}) возможны значения, зависящие от области определения X. Так, значения лингвистической переменной «Температура жидкости» можно определить как «около 0^o», «около 50^o», «около 100^o», т. е. в виде нечетких чисел.

Терм-множество и расширенное терм-множество для нашего примера можно характеризовать функциями принадлежности, приведенными на рис. 5.6 и 5.7.

На рис. 5.6 приведены функции принадлежности нечетких множеств A_1 = *«Низкая температура»*, A_2 = *«Средняя температура»* и A_3 = *«Высокая температура»*, которые составляют терм-множество.

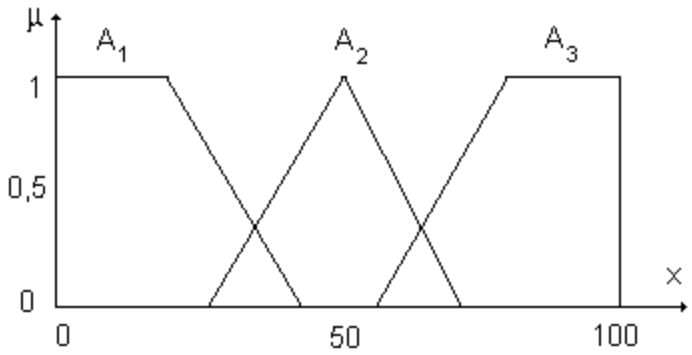


Рис. 5.6 Функции принадлежности терм-множества

На рис. 5.7 приведена функция принадлежности нечеткого множества $A_2 \cup A_3$ = *«Средняя или высокая температура»*, которое может в составе расширенного терм-множества.

Отметим, что функция принадлежности отличается от вероятности, которая имеет объективный характер и подчиняется другим математическим зависимостям.

Функция принадлежности обычно зависит от мнения экспертов, участвующих в определении нечеткого множества.

Например, для двух экспертов определение нечеткого множества «высокая» для лингвистической переменной «цена автомобиля» может заметно отличаться:

«высокая_цена_автомобиля_1» = {50 000/1+25 000/0,8+10 000/0,6+5000/0,4}.

«высокая_цена_автомобиля_2» = {25 000/1+10 000/0,8+5000/0,7+3000/0,4}.

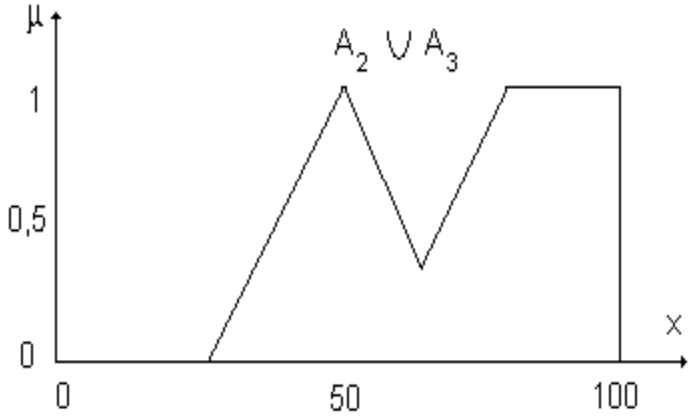


Рис. 5.7 Функция принадлежности нечеткого множества $A_2 \cup A_3$

Пример. Лингвистическая переменная «возраст».

Пусть требуется интерпретация значений лингвистической переменной «возраст», таких как «младенческий» и «детский». Базовый набор значений логической переменной «возраст» может быть определен следующим образом:

B = {«младенческий», «детский», «юный», «молодой», «зрелый», «преклонный»}.

Для логической переменной «возраст» базовую шкалу представляет числовая шкала прожитых лет от 0 до 120, а функция принадлежности определяет степень уверенности в том, что данное количество лет можно отнести к данной категории возраста.

На рис. 5.8 показана схема формирования нечетких множеств логической переменной «возраст».

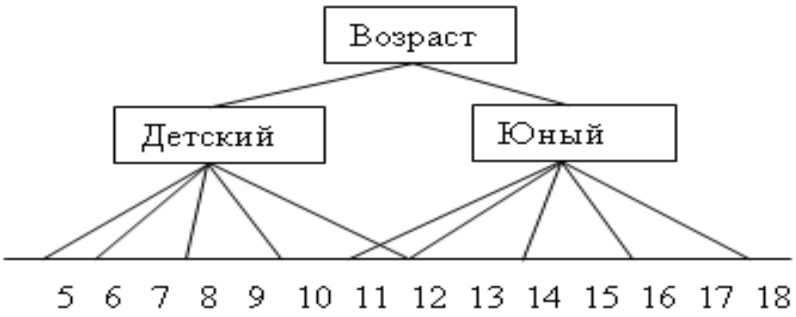


Рис. 5.8 Схема формирования нечетких множеств

Как видим, одни и те же значения базовой шкалы могут участвовать в определении различных нечетких множеств.

5.6 Нечеткие лингвистические высказывания

В системах нечеткого вывода условия и заключения отдельных правил формулируются в форме нечетких высказываний относительно значений лингвистических переменных.

Нечетким лингвистическим высказыванием (или просто нечетким высказыванием) называют высказывания следующих видов:

1 Высказывание «β *есть* α», где β – наименование лингвистической переменной, α – ее значение, которому соответствует отдельный лингвистический терм из базового терм-множества T лингвистической переменной β.

2 Высказывание «β *есть* ∇α», где ∇ – *модификатор*, соответствующий таким словам, как «ОЧЕНЬ», «БОЛЕЕ ИЛИ МЕНЕЕ», «МНОГО БОЛЬШЕ» и другим, которые могут быть получены с использованием процедур *G* и *M* данной лингвистической переменной.

3 Составные высказывания, образованные из высказываний видов 1 и 2 и нечетких логических операций в форме связок: «И», «ИЛИ», «ЕСЛИ … ТО», «НЕ».

Пример. Нечеткое высказывание второго вида «температура жидкости *очень высокая*» означает, что лингвистической переменной «температура жидкости» присваивается значение «*высокая*» с модификатором «ОЧЕНЬ», который изменяет значение соответствующего лингвистического терма «*высокая*» на основе использования расчетной формулы для операции концентрации CON (A) нечеткого множества A для терма «*высокая*».

Нечеткое высказывание второго вида «температура жидкости *сравнительно высокая*» означает, что лингвистической переменной «температура жидкости» присваивается значение «*высокая*» с модификатором «СРАВНИТЕЛЬНО», который изменяет значение соответствующего лингвистического терма «*высокая*» на основе использования расчетной формулы для операции растяжения DIL (A) нечеткого множества A для терма «*высокая*».

Ниже на рис. 5.9. приведен пример функции принадлежности A_c терм-множества «*средняя*» лингвистической переменной «температура жидкости» и определение значений функций принадлежности A_{dc} = DIL (A_c) этого же терм-множества для модификатора «СРАВНИТЕЛЬНО».

Нечеткое высказывание третьего вида «температура жидкости низкая и *скорость автомобиля высокая*» означает, что одной лингвистической переменной «температура жидкости» присваивается значение «низкая», а лингвистической переменной «*скорость автомобиля*» присваивается значение «*высокая*». Эти нечеткие высказывания первого вида соединены логической операцией нечеткая конъюнкция (операцией нечеткое «И»).

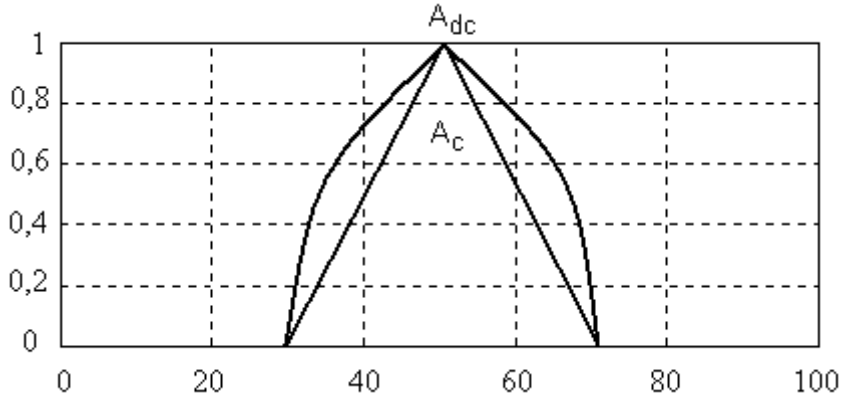


Рис. 5.9 Вид функций принадлежности
терм-множества «*средняя*»

5.7 Логические операции с нечеткими высказываниями

Для рассмотрения логических операций с нечеткими высказываниями обозначим: Y – некоторое множество элементарных нечетких высказываний, а T : Y→[0, 1]; T – отображение истинности высказываний.

Отрицанием (логическим) нечеткого высказывания A (обозначается ¬A, читается как «не A») называется унарная логическая операция, результат которой является нечетким высказыванием, истинность ее есть:

$$T(\neg A) = 1 - T(A). \qquad (5.14)$$

Конъюнкцией (логической) нечетких высказываний A и B (обозначается A∧B, читается как «A и B») называется бинарная логическая операция, результат которой является нечетким высказыванием, истинность которого:

$$T(A \wedge B) = \min\{T(A), T(B)\}. \qquad (5.15)$$

Конъюнкцию нечетких высказываний также называют *нечетким логическим «И»*, *нечеткой конъюнкцией* или min-*конъюнкцией* и записывают в форме A AND B. Формулу (5.15) принимают основной для определения степени истинности конъюнкции.

Для определения степени истинности *конъюнкции* нечетких высказываний могут быть использованы следующие *альтернативные формулы*.

Алгебраическое произведение степеней истинности нечетких высказываний (обозначается A•B):

$$T(A \wedge B) = T(A) \cdot T(B). \qquad (5.16)$$

Граничное произведение степеней истинности нечетких высказываний (обозначается A⊙B):

$$T(A \wedge B) = \max\{T(A)+T(B) - 1, 0\}. \qquad (5.17)$$

Драстическое произведение степеней истинности нечетких высказываний (обозначается AΔB):

$$\boldsymbol{T}(A \wedge B) = \begin{cases} \boldsymbol{T}(B), & \text{если } \boldsymbol{T}(A) = 1; \\ \boldsymbol{T}(A), & \text{если } \boldsymbol{T}(B) = 1; \\ 0, & \text{в остальных случаях} \end{cases} \qquad (5.18)$$

Дизъюнкцией (логической) нечетких высказываний A и B (обозначается A∨B − читается «A или B») называется бинарная логическая операция, результат которой является нечетким высказыванием, истинность которого есть:

$$T(A \vee B) = \max\{T(A), T(B)\}. \qquad (5.19)$$

Эту операцию также называют нечетким не исключающим логическим «ИЛИ», нечеткой дизъюнкцией или max-дизъюнкцией и иногда записывают также в форме A OR B. Формулу (5.19) считают основной для определения степени истинности дизъюнкции нечетких высказываний.

Для определения степени истинности дизъюнкции нечетких высказываний могут быть использованы следующие альтернативные формулы.

Алгебраическая сумма степеней истинности нечетких высказываний (обозначается A + B):

$$T(A \vee B) = T(A) + T(B) - T(A) \cdot T(B). \qquad (5.20)$$

Алгебраическую сумму часто называют также вероятностной суммой.

Граничная сумма степеней истинности нечетких высказываний (обозначается A⊕B):

$$T(A \vee B) = \min\{T(A) + T(B), 1\}.$$

Драстическая сумма степеней истинности нечетких высказываний (обозначается A∇B):

$$\boldsymbol{T}(A \vee B) = \begin{cases} \boldsymbol{T}(B), & \text{если } \boldsymbol{T}(A) = 0; \\ \boldsymbol{T}(A), & \text{если } \boldsymbol{T}(B) = 0; \\ 1, & \text{в остальных случаях.} \end{cases}$$

Замечание. В общем случае для определения истинности результатов нечеткой конъюнкции и нечеткой дизъюнкции могут использоваться и другие расчетные формулы, основанные на рассмотрении треугольных норм и конорм (произвольных функций двух переменных, удовлетворяющих ряду аксиом).

Нечеткой импликацией, или просто *импликацией* нечетких высказываний A и B (записывается как A⊃B и читается – «из A следует B», «ЕСЛИ A, ТО *B*»), называется бинарная логическая операция, результат которой является нечетким высказыванием, истинность которого определяется по одной из следующих формул:

1 Классическая нечеткая импликация Л. Заде

$$T(A \supset B) = \max\{\min\{T(A), T(B)\}, 1 - T(A)\}.$$

2 Классическая нечеткая импликация для случая $T(A) \geq T(B)$

$$T(A \supset B) = \max\{T(\neg A), T(B)\} = \max\{1 - T(A), T(B)\}.$$

Эту форму нечеткой импликации иногда называют нечеткой импликацией Геделя.

3 Нечеткая импликация, предложенная Э. Мамдани

$$T(A \supset B) = \min\{T(A), T(B)\}.$$

Эту форму нечеткой импликации также называют нечеткой импликацией *минимума корреляции*. Заметим, что в случае $T(A) \geq 0.5$ и $T(B) \geq 0.5$ классическая нечеткая импликация превращается в нечеткую импликацию Мамдани.

Кроме приведенных формул, существуют также варианты нечеткой импликации, предложенные Я. Лукасевичем, Дж. Гогеном, Н. Вади и др. Выбор того или иного варианта нечеткой импликации определяется с учетом характера решаемой прикладной задачи и простоты вычислений.

6 ПРОДУКЦИОННЫЕ СИСТЕМЫ С НЕЧЕТКИМИ ЗНАНИЯМИ

Отметим, что системы искусственного интеллекта, работающие с нечеткими знаниями, чаще всего строятся с использованием продукционной модели представления знаний на основе правил нечетких продукций. Рассмотрим, каким образом осуществляется вывод решений в продукционных системах с нечеткими знаниями.

6.1 Этапы нечеткого вывода

Чаще всего системы нечеткого вывода используются для управления техническими устройствами и процессами. Разработка и применение систем нечеткого вывода включают в себя ряд этапов.

Информацией, которая поступает на вход системы нечеткого вывода, являются измеренные некоторым образом входные переменные. Они соответствуют переменным процесса управления. Информация на выходе системы нечеткого вывода соответствует выходным переменным, являющимся управляющими переменными процесса управления.

Системы нечеткого вывода выполняют преобразование значений входных переменных процесса управления в выходные переменные на основе нечетких правил продукций. Для этого они должны содержать базу правил нечетких продукций и реализовывать нечеткий вывод заключений на основе посылок или условий, представленных в форме нечетких лингвистических высказываний.

В процессе нечеткого вывода основными являются следующие этапы:

1 Формирование базы правил систем нечеткого вывода.
2 Введение нечеткости (фаззификация).
3 Агрегирование подусловий в нечетких правилах продукций.
4 Активизация подзаключений в нечетких правилах продукций.

5 Аккумулирование заключений нечетких правил продукций.
6 Придание результатам четкости (дефаззификация).

6.2 Правила нечетких продукций

В системах нечеткого вывода условия и заключения отдельных нечетких правил формулируются в форме нечетких высказываний вида 1-3 относительно значений лингвистических переменных.

Под *правилом нечеткой продукции*, или просто нечеткой продукцией, в общем случае понимается выражение вида

$$(i) : Q; P; A \Rightarrow B; S, F, N, \qquad (6.1)$$

где (i) − имя нечеткой продукции; Q − сфера применения нечеткой продукции; P − условие применимости ядра нечеткой продукции; A⇒B − ядро нечеткой продукции, в котором A − условие ядра (антецедент); B − заключение ядра (консеквент); ⇒ − знак логической секвенции (или следования); S − способ определения степени истинности заключения ядра; F − коэффициент уверенности нечеткой продукции; N − постусловия продукции.

В качестве имени (i) нечеткой продукции может выступать та или иная совокупность букв или символов, позволяющая однозначным образом идентифицировать нечеткую продукцию в системе нечеткого вывода или базе нечетких правил. В качестве имени нечеткой продукции может использоваться ее номер в системе.

Ядро продукции записывается в форме «ЕСЛИ A, ТО B» или в распространенном виде «IF A, THEN B», где A и B часто представляются в форме нечетких высказываний. В качестве выражений A и B могут использоваться составные логические нечеткие высказывания.

Способ S в общем случае определяет так схему или алгоритм нечеткого вывода в продукционных нечетких системах и называется также методом композиции или методом активации согласно.

F − коэффициент уверенности выражает количественную оценку степени истинности или относительный вес нечеткой продукции. Принимает свое значение из интервала [0, 1] и часто называется *весовым коэффициентом нечеткого правила продукции*.

Простейший вариант правила нечеткой продукции, который наиболее часто используется в системах нечеткого вывода, может быть записан в форме

$$\text{ПРАВИЛО <\#>: ЕСЛИ «}\beta_1 \textit{ есть } \alpha_1\text{», ТО «}\beta_2 \textit{ есть } \alpha_2\text{».} \qquad (6.2)$$

Здесь нечеткое высказывание «β_1 *есть* α_1» представляет собой условие правила нечеткой продукции, а нечеткое высказывание «β_2 *есть* α_2» − нечеткое заключение правила. При этом считается, что $\beta_1 \neq \beta_2$.

Система нечетких правил продукций, или *продукционная нечеткая система,* представляет собой согласованное множество отдельных *нечетких продукций* или правил нечетких продукций в форме «ЕСЛИ A, ТО B», где A и B − нечеткие лингвистические высказывания вида 1, 2 или 3.

Рассмотрим вариант использования в качестве условия или заключения в правиле нечеткой продукции нечеткого высказывания вида 2, т. е. вида «β *есть* $\nabla\alpha$», где ∇ − модификатор, определяемый процедурами *G* и *M* лингвистической переменной β. Пусть терму α соответствует нечеткое множество A. В этом случае исходное нечеткое высказывание «β *есть* $\nabla\alpha$» можно преобразовать к виду 1 в форме нечеткого высказывание «β *есть* α_1», где терм α_1 получается на основе применения определенной процедурами *G* и *M* операции к нечеткому множеству A. Полученное в результате подобной операции нечеткое множество A_1 принимается за значение терм-множества α_1.

Если в качестве условия или заключения используются составные нечеткие высказывания, т. е. образованные из высказываний видов 1 и 2 и нечетких логических операций в форме связок «И», «ИЛИ», «ЕСЛИ … ТО», «НЕ», то ситуация несколько усложняется. Поскольку вариант использования нечетких высказываний вида 2 сводится к нечетким высказываниям вида 1, то достаточно рассмотреть сложные высказывания, в которых нечеткими логическими операциями соединены только нечеткие высказывания вида 1.

Во-первых, эта ситуация может соответствовать простейшему случаю, когда нечеткими логическими операциями соединены нечеткие высказывания, относящиеся к одной и той же лингвистической переменной, т. е. в форме «β *есть* α_1» ОП «β *есть* α_2», где ОП − некоторая из бинарных операций нечеткой конъюнкции «И» или нечеткой дизъюнкции «ИЛИ».

Замечание. Нечеткая импликация и нечеткая эквивалентность могут быть выражены через операции нечеткой конъюнкции и нечеткой дизъюнкции, а нечеткое отрицание в здесь является, по сути, модификатором.

В этом простейшем случае нечеткое высказывание «β *есть* α_1» И «β *есть* α_2» эквивалентно нечеткому высказыванию «β *есть* α^*», где терм-множеству α^* соответствует нечеткое множество A*, равное *пересечению* нечетких множеств A_1 и A_2, которые соответствуют термам α_1 и α_2. При этом операция пересечения определяется одним из ранее рассмотренных способов.

Соответственно, нечеткое высказывание «β *есть* α_1» ИЛИ «β *есть* α_2» эквивалентно нечеткому высказыванию «β *есть* α^*», где терм-множеству α^* соответствует нечеткое множество A*, равное *объединению* нечетких множеств

A_1 и A_2, которые соответствуют термам α_1 и α_2. При этом операция объединения определяется одним из ранее рассмотренных способов.

Пример. Рассмотрим составное нечеткое высказывание вида 3 «*скорость автомобиля средняя и скорость автомобиля высокая*». Ему соответствуют два нечетких высказывания первого вида, соединенные логической операцией нечеткой конъюнкции. Тогда исходное нечеткое высказывание эквивалентно нечеткому высказыванию первого вида «*скорость автомобиля средняя и высокая*».

Функция принадлежности терма «*средняя и высокая*» изображена на рис. 6.1а более темным фоном, при этом результат нечеткой конъюнкции определялся по формуле (5.7).

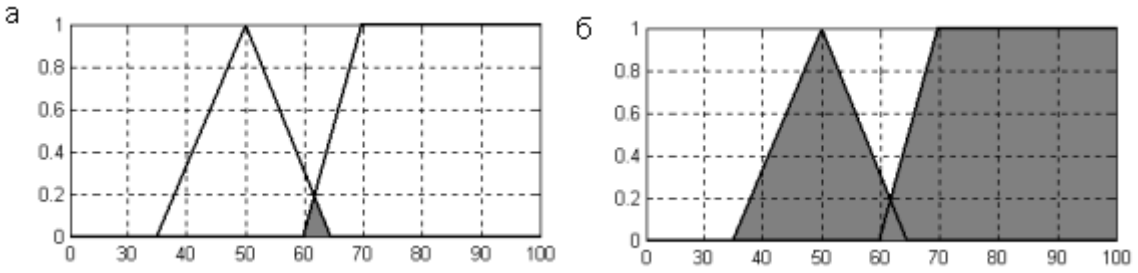


Рис. 6.1 Преобразование составных нечетких высказываний

Рассмотрим аналогичное составное нечеткое высказывание вида 3 «*скорость автомобиля средняя или скорость автомобиля высокая*». Ему также соответствуют два нечетких высказывания первого вида, соединенные логической операцией нечеткой дизъюнкции. Тогда исходное нечеткое высказывание эквивалентно нечеткому высказыванию первого вида: «*скорость автомобиля средняя или высокая*». Функция принадлежности терма «*средняя или высокая*» изображена на рис. 6.1б более темным фоном, при этом результат нечеткой дизъюнкции определялся по формуле (5.8).

Во-вторых, ситуация может соответствовать более сложному случаю, когда нечеткими логическими операциями соединены нечеткие высказывания, относящиеся к разным лингвистическим переменным в условии правила нечеткой продукции, т. е. в форме «β_1 *есть* α_1» ОП «β_2 *есть* α_2», где ОП − некоторая из бинарных операций нечеткой конъюнкции «И» или нечеткой дизъюнкции «ИЛИ», а β_1 и β_2 − различные лингвистические переменные.

Этот вариант правил нечетких продукций может быть записан в форме:

ПРАВИЛО <#>: ЕСЛИ «β_1 *есть* α_1» И «β_2 *есть* α_2» ТО «β_3 *есть* α_3» (6.3)

или

ПРАВИЛО <#>: ЕСЛИ «β_1 *есть* α_1» ИЛИ «β_2 *есть* α_2» ТО «β_3 *есть* α_3».

Здесь нечеткие высказывания: «β_1 *есть* α_1» И «β_2 *есть* α_2», «β_1 *есть* α_1» ИЛИ «β_2 *есть* α_2» представляют собой условия правил нечетких продукций, а нечеткое высказывание «β_3 *есть* α_3» – заключение правил. При этом считается, что $\beta_1 \neq \beta_2 \neq \beta_3$, а каждое из нечетких высказываний «β_1 *есть* α_1», «β_2 *есть* α_2» называют *подусловиями* правил нечетких продукций.

В случае правил нечетких продукций в форме (6.3) необходимо использовать один из методов агрегирования условий в левой части этих правил.

В-третьих, нечеткими логическими операциями могут быть соединены нечеткие высказывания, относящиеся к разным лингвистическим переменным в заключении правила нечеткой продукции. Этот вариант правил нечетких продукций может быть записан в следующей общей форме:

ПРАВИЛО <#>: ЕСЛИ «β_1 *есть* α_1» ТО «β_2 *есть* α_2» И «β_3 *есть* α_3» (6.4)

или

ПРАВИЛО <#>: ЕСЛИ «β_1 *есть* α_1» ТО «β_2 *есть* α_2» ИЛИ «β_3 *есть* α_3».

Здесь нечеткое высказывание «β_1 *есть* α_1» представляет собой условие правил нечетких продукций, а нечеткие высказывания: «β_2 *есть* α_2» И «β_3 *есть* α_3», «β_2 *есть* α_2» ИЛИ «β_3 *есть* α_3» – заключения данных правил. При этом считается, что $\beta_1 \neq \beta_2 \neq \beta_3$, а каждое из нечетких высказываний «β_2 *есть* α_2», «β_3 *есть* α_3» называют *подзаключениями* правила нечеткой продукции.

В случае правил нечетких продукций в форме (6.4) необходимо использовать один из методов аккумуляции подзаключений в правилах нечетких продукций.

6.3 Расчет количества и оптимизация базы правил

Составные нечеткие высказывания объединяются в правила с помощью нечеткой логической операции конъюнкции (И) или дизъюнкции (ИЛИ). В табл. 6.1 приводится база правил для алгоритмов нечеткого вывода Мамдани, Ларсена и Такаги-Сугено.

Рассмотрим описание ФП к табл. 6.1. Входная переменная X описывается тремя ФП X_i, а A двумя ФП A_i, выходная переменная F описывается тремя ФП F_i. В базе правил нельзя разместить больше вариантов правил, чем произведение числа функций принадлежности всех входных переменных. Их число равно 6 (2x3).

Таблица 6.1 **Вид правил нечеткого вывода для различных алгоритмов**

№ правила	Общие условия всех алгоритмов	Заключения для алгоритмов	
		Мамдани и Ларсена	Такаги-Сугено
1	**ЕСЛИ** z есть X_1 **И** q есть A_1	**ТО** y есть F_1 (низкая)	$y = a_1 z + b_1 q$
2	**ЕСЛИ** z есть X_1 **И** q есть A_2	**ТО** y есть F_2 (средняя)	$y = a_2 z + b_2 q$
3	**ЕСЛИ** z есть X_2 **И** q есть A_1	**ТО** y есть F_2 (средняя)	$y = a_2 z + b_2 q$
4	**ЕСЛИ** z есть X_2 **И** q есть A_2	**ТО** y есть F_3 (высокая)	$y = a_3 z + b_3 q$
5	**ЕСЛИ** z есть X_3	**ТО** y есть F_3 (высокая)	$y = a_3 z + b_3 q$

Поэтому максимальное количество правил в базе определяется соотношением [25]

$$l = l_1 \cdot l_2 \cdot \ldots \cdot l_m ,$$

где: l_i – количество функций принадлежности, используемых для задания входной переменной x_i $(i = \overline{1, m})$.

Поскольку сформированная база, состоящая из максимального количества правил, является избыточной, то экспертам необходимо ее оптимизировать для сокращения числа правил в базе.

Для этого может применяться ряд способов:

– сокращение количества предпосылок в правиле;

– объединение нескольких правил в одно с заменой в условии нечеткой логической операции И на ИЛИ, если у всех правил есть несколько одинаковых (важных) предпосылок, одинаковые заключения, а остальные предпосылки могут быть любыми (несущественны);

– объединение нескольких правил в одно, если у всех правил есть одна одинаковая предпосылка, одинаковые заключения, а остальные предпосылки могут быть любыми, и другие способы.

Для примера рассмотрим правило № 5 из табл. 6.1, в нем предпосылка «**ЕСЛИ** z есть X_3» является важной, поэтому входные данные q могут быть любыми, заключение все равно будет одинаковое – «**ТО** y есть F_3».

Таким образом, вместо двух правил:

ЕСЛИ z есть X_3 **И** q есть A_1 **ТО** y есть F_3 ,

ЕСЛИ z есть X_3 **И** q есть A_2 **ТО** y есть F_3 ,

мы получили одно правило, указанное в табл. 6.1 [4].

При проектировании базы знаний необходимо придерживаться следующих правил:

– должно существовать хотя бы одно правило для каждого терма выходной переменной;

– для любого терма входной переменной должно существовать хотя бы одно правило, в котором этот терм используется в качестве посылки;

– для произвольного вектора входных переменных должно существовать хотя бы одно правило, степень выполнения которого больше нуля.

Другими словами, правила базы знаний должны покрывать всю предметную область.

Таким образом, оптимизация базы правил приводит к существенному уменьшению количества и ликвидации противоречивости правил, оставляемых в базе.

6.4 Введение нечеткости

Введение нечеткости (или фаззификация) в продукционную систему нечеткого вывода представляет собой определение значений функции принадлежности нечетких множеств для всех значений входных переменных x_j, входящих в состав предпосылок (подусловий) всех нечетких продукционных правил. При этом $x_j \in X_j$, где X_j есть универсум лингвистической переменной из j-го подусловия.

Перед началом этапа введения нечеткости некоторым внешним образом, например, с помощью датчиков, задаются значения входных переменных x_j. Причем, эти значения в общем случае могут задаваться как четкие или как нечеткие (с помощью функции принадлежности).

Если значения входных переменных x_j являются *четкими*, то каждого из них находится количественное значение функции принадлежности $\mu_{Aj}(x_j)$ для каждого из подусловий «β_jесть α_j» в составе базы правил системы нечеткого вывода (j = 1,…k). Полученное таким образом значение $\mu_{Aj}(x_j)$ и является результатом введения нечеткости для подусловия «β_j есть α_j». Фактически функция принадлежности $\mu_{Aj}(x)$ задает нам степень истинности подусловия «β_j есть α_j» для j-го правила.

Пример. Рассмотрим пример фаззификации двух нечетких высказываний «температура жидкости небольшая» и «температура жидкости средняя» для входной лингвистической переменной β. Им соответствуют нечеткие высказывания вида: «β есть α_1» и «β есть α_2». Предположим, что температура жидкости составляет 45°, т. е. $x_j = 45°$. В этом случае фаззификация первого высказывания путем подстановки $x_1 = 45°$ в качестве аргумента функции принадлежности терма α_1 дает нам значение 0 (рис. 6.2а). Это значение указывает на степень истинности первого высказывания, аналогично получаем,

что степень истинности второго высказывания составляет примерно 0,72 (рис. 6.2б).

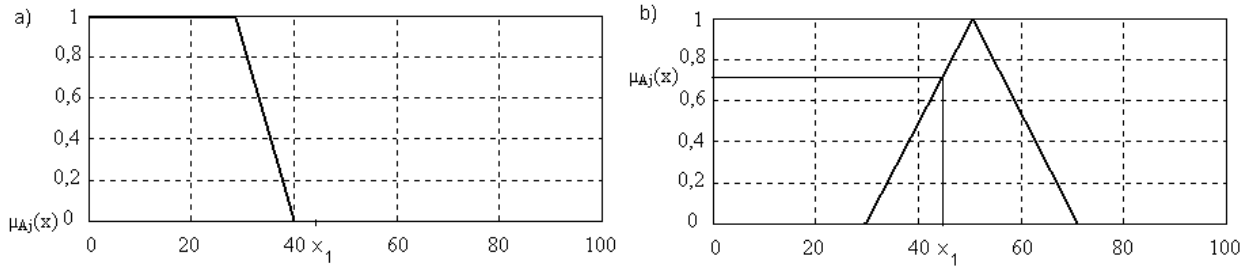


Рис. 6.2 Фаззификация нечетких высказываний по четкому значению

Если значения входных переменных χ_j являются *нечеткими* и задаются с помощью функции принадлежности, то для достижения цели фаззификации применяются операции нечеткой конъюнкции. Чаще всего при этом используются операции min-конъюнкции $\mu_{A'j}(x_j) = \min\{\mu_{\chi j}(x_j), \mu_{Aj}(x_j)\}$, $\forall x_j \in X_j$ или алгебраическое произведение $\mu_{A'j}(x_j) = \mu_{\chi j}(x_j)\mu_{Aj}(x_j)$, $\forall x_j \in X_j$.

Пример. Рассмотрим пример фаззификации нечеткого высказывания «температура жидкости небольшая» для входной лингвистической переменной β. Предположим, температура жидкости задается с помощью нечеткой входной переменной χ_j, задаваемой с помощью нечеткого множества с функцией принадлежности $\mu_{\chi j}(x_j)$. В этом случае операция min-конъюнкции дает нам степень истинности высказывания $\mu_{A'j}(x_j)$ равную примерно 0,63 (рис. 6.3а).

Операция алгебраического произведения дает нам степень истинности высказывания $\mu_{A'j}(x_j)$ равную примерно 0,34 (рис. 6.3б).

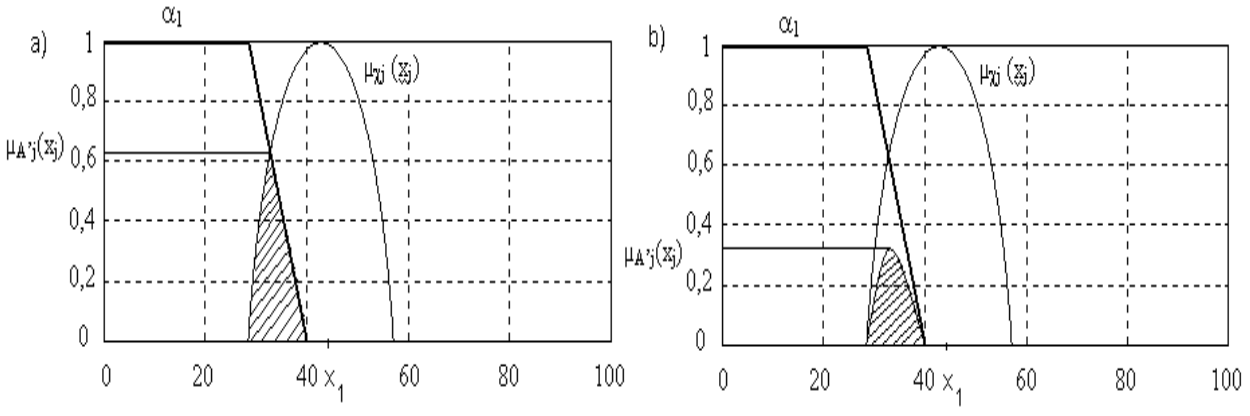


Рис. 6.3 Фаззификация нечеткого высказывания по нечеткому значению

6.5 Агрегирование подусловий в нечетких правилах

Агрегирование подусловий в нечетких правилах продукций представляет собой процедуру определения степени истинности условий по каждому из правил системы нечеткого вывода.

Формально процедура агрегирования выполняется следующим образом. Предварительно предполагаются известными значения истинности всех подусловий системы нечеткого вывода, т. е. множество значений $B = \{b_i'\}$. Далее рассматривается каждое из условий правил системы нечеткого вывода. Если условие правила представляет собой нечеткое высказывание вида 1 или 2, то степень его истинности равна соответствующему значению b_i'.

Если же условие состоит из нескольких подусловий вида (6.3), причем лингвистические переменные в подусловиях попарно не равны друг другу, то определяется степень истинности сложного высказывания на основе известных значений истинности подусловий. При этом для определения результата нечеткой конъюнкции или связки «И» может быть использована основная формула (5.15) или одна из альтернативных формул определения логической конъюнкции нечетких высказываний.

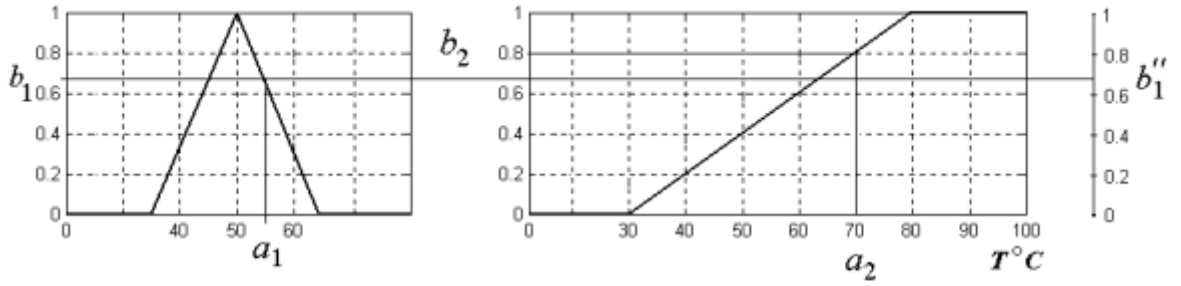
Для определения результата нечеткой дизъюнкции или связки «ИЛИ» может быть использована основная формула (5.19) или одна из альтернативных формул определения логической дизъюнкции нечетких высказываний. При этом значения b_i' используются в качестве аргументов соответствующих логических операций. Тем самым находятся значения степени истинности всех условий правил системы нечеткого вывода.

Этап агрегирования считается законченным, когда будут найдены все значения b_k'' для каждого из правил R_k, входящих в рассматриваемую базу правил P системы нечеткого вывода. Это множество значений обозначим через $B''=\{b_1'', b_2'',\ldots, b_n''\}$.

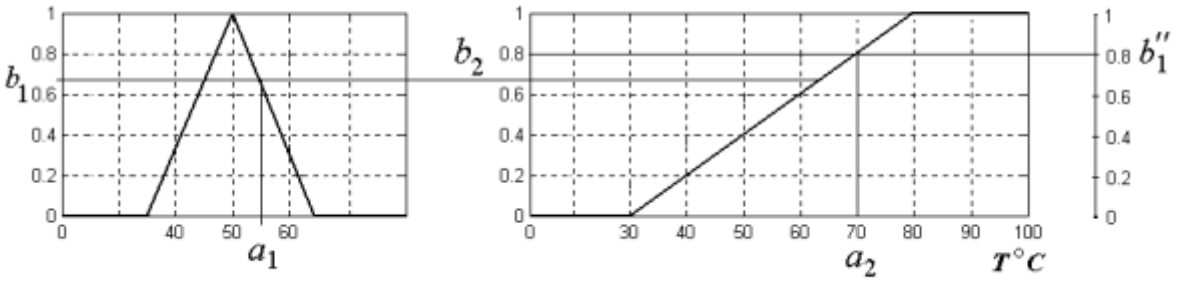
Пример. Рассмотрим агрегирование двух нечетких высказываний: «скорость автомобиля средняя» И «температура жидкости высокая» и «скорость автомобиля средняя» ИЛИ «температура жидкости высокая» для входной лингвистической переменной β_1 – скорость движения автомобиля и β_2 – температура жидкости. Пусть текущая скорость автомобиля равна 55 км/ч, т. е. $a_1 = 55$ км/ч, а температура жидкости равна $a_2 = 70$ °C.

Тогда агрегирование подусловий для первого нечеткого высказывания с использованием операции нечеткой конъюнкции (5.15) дает в результате число $b_1''= 0.67$ (приближенное значение), которое означает его степень истинности и получается как минимальное из значений 0.67 и 0.8 (рис. 6.4, а).

Агрегирование подусловий для второго нечеткого высказывания с использованием операции нечеткой дизъюнкции (5.19) дает в результате число $b_1'' = 0.8$, которое означает его степень истинности и получается как максимальное из значений 0.67 и 0.8 (рис. 6.4,б).



а



б

Рис. 6.4 Агрегирование подусловий

При использовании расчетных формул для определения результатов нечеткой конъюнкции и нечеткой дизъюнкции целесообразно применять попарно согласованные методы расчета для всех правил системы нечетких продукций. Так, например, если в некоторой системе нечеткого вывода результат нечеткой конъюнкции определяется по формуле алгебраического произведения (5.16), то для определения результата нечеткой дизъюнкции предпочтительно использовать алгебраическую сумму (5.20).

Агрегирование степеней истинности предпосылок правил, объединенных с помощью нечеткой логической операции конъюнкции (И), осуществляется по формуле min-конъюнкции [15]:

$$K_{\mu_{it} \cap \mu_{im}} = \min\{K_{\mu_{it}}(u_t), K_{\mu_{im}}(u_m)\}, \qquad (6.5)$$

где $K_{\mu_{it}}(u_t)$ и $K_{\mu_{im}}(u_m)$ – значения ФП предпосылок правил переменных *t* и *m*;

u_t , u_m – входные значения переменных *t* и *m*.

Агрегирование степеней истинности предпосылок правил, объединенных с помощью нечеткой логической операции дизъюнкции (ИЛИ), выполняется по формуле max-дизъюнкции:

$$K_{\mu_{it} \cup \mu_{im}} = \max\left\{K_{\mu_{ij}}(u_t), K_{\mu_{im}}(u_m)\right\}. \qquad (6.6)$$

Общий вид выражения (6.5) для СППР с шестью входами и одним выходом преобразуется к виду

$$K_n = \min\left(A_i(u_j), B_i(u_j), C_i(u_j), D_i(u_j), E_i(u_j), F_i(u_j)\right). \qquad (6.7)$$

Выполним агрегирование степени истинности предпосылок по каждому из нечетких продукционных правил СППР. Тогда агрегированная степень истинности предпосылок наличия ПВ в АСУ по правилу № 1 (5.3) рассчитывается с помощью формулы (6.7):

$$K_1 = \min\left(A_1(u_1), B_1(u_2), C_1(u_3), D_1(u_4), E_1(u_5)\right),$$

где K_1 – агрегированная степень истинности предпосылок правила № 1 (5.3); $A_1(u_1), B_1(u_2), C_1(u_3), D_1(u_4), E_1(u_5)$ – значения ФП предпосылок правила № 1 (полученные на этапе 1).

Тогда

$$K_1 = \min\ (1, 0.904, 0.087, 1, 1) = 0.087.$$

Соответственно, для правила № 2 (5.4) имеем

$$K_2 = \min\ (1, 0.904, 0.885, 1, 1\ , 0.585) = 0.585 \text{ и т. д.}$$

В работе [8] рассматривается система MYCIN. В ней для того чтобы не активизировались правила, которые являются малозначительными, вводится пороговое ограничение для K_n. Так, введение порогового ограничения для каждого правила $K_n \leq 0.05$, приводит к снижению сложности алгоритма и не оказывает существенного влияния на точность расчета степени уверенности СППР.

6.6 Активизация подзаключений в нечетких правилах

Активизация в системах нечеткого вывода представляет собой нахождение степени истинности каждого из подзаключений правил нечетких продукций. Активизация в общем случае во многом аналогична композиции нечетких отношений, но не тождественна ей. Поскольку в системах нечеткого вывода используются лингвистические переменные, то формулы для нечеткой композиции теряют свое значение. В действительности при формировании базы правил системы нечеткого вывода задаются весовые коэффициенты F_i для каждого правила (по умолчанию предполагается, если весовой коэффициент не задан явно, то его значение равно 1).

Формально процедура активизации выполняется следующим образом. Предварительно предполагаются известными значения истинности всех условий системы нечеткого вывода, т. е. множество значений $B''=\{b_1'', b_2'',\ldots, b_n''\}$ и значения весовых коэффициентов F_i для каждого правила. Далее рассматривается каждое из заключений правил системы нечеткого вывода. Если заключение правила представляет собой нечеткое высказывание вида 1 или 2, то степень его истинности равна алгебраическому произведению соответствующего значения b_i'' на весовой коэффициент F_i.

Если заключение состоит из нескольких подзаключений вида (6.2), причем лингвистические переменные в подзаключениях попарно не равны друг другу, то степень истинности каждого из подзаключений равна алгебраическому произведению соответствующего значения b_i'' на весовой коэффициент F_i. Таким образом, находятся все значения c_k степеней истинности подзаключений для каждого из правил R_k, входящих в рассматриваемую базу правил P системы нечеткого вывода. Это множество значений обозначим через $C = \{c_1, c_2,\ldots, c_q\}$, где q – общее количество подзаключений в базе правил.

Отметим, что весовой коэффициент F_i может быть задан индивидуально для отдельных подзаключений. При этом процедура активизации остается прежней.

После нахождения множества $C = \{c_1, c_2,\ldots, c_q\}$ определяются функции принадлежности каждого из подзаключений для рассматриваемых выходных лингвистических переменных. Для этой цели можно использовать один из методов, получивших наибольшее распространение и являющихся модификацией того или иного метода нечеткой композиции:

1) min-активизация:

$$\mu'(y) = \min\{c_i, \mu(y)\};$$

2) prod-активизация:

$$\mu'(y) = c_i \cdot \mu(y);$$

3) average-активизация:

$$\mu'(y) = 0.5\cdot(c_i+\mu(y)),$$

где $\mu(y)$ – функция принадлежности терма, который является значением некоторой выходной переменной ω_j, заданной на универсуме $\boldsymbol{Y}$.

Для выполнения активизации могут быть использованы и другие способы, основанные на модификации различных операций нечеткой композиции.

Этап активизации завершается, когда для каждой из выходных лингвистических переменных, входящих в отдельные подзаключения правил нечетких продукций, будут определены функции принадлежности нечетких множеств их значений, т. е. совокупность нечетких множеств: $C_1, C_2,\ldots,C_q$, где q – общее количество подзаключений в базе правил системы нечеткого вывода.

Пример. Рассмотрим пример процесса активизации заключения в следующем правиле нечеткой продукции:

ЕСЛИ «*скорость автомобиля средняя*» ТО «*температура охлаждающей жидкости высокая*».

Входной лингвистической переменной в этом правиле является β_1 – скорость автомобиля, а выходной переменной является β_2 – температура жидкости. Предположим, что текущая скорость автомобиля равна 55 км/ч, т. е. $a_1 = 55$ км/ч.

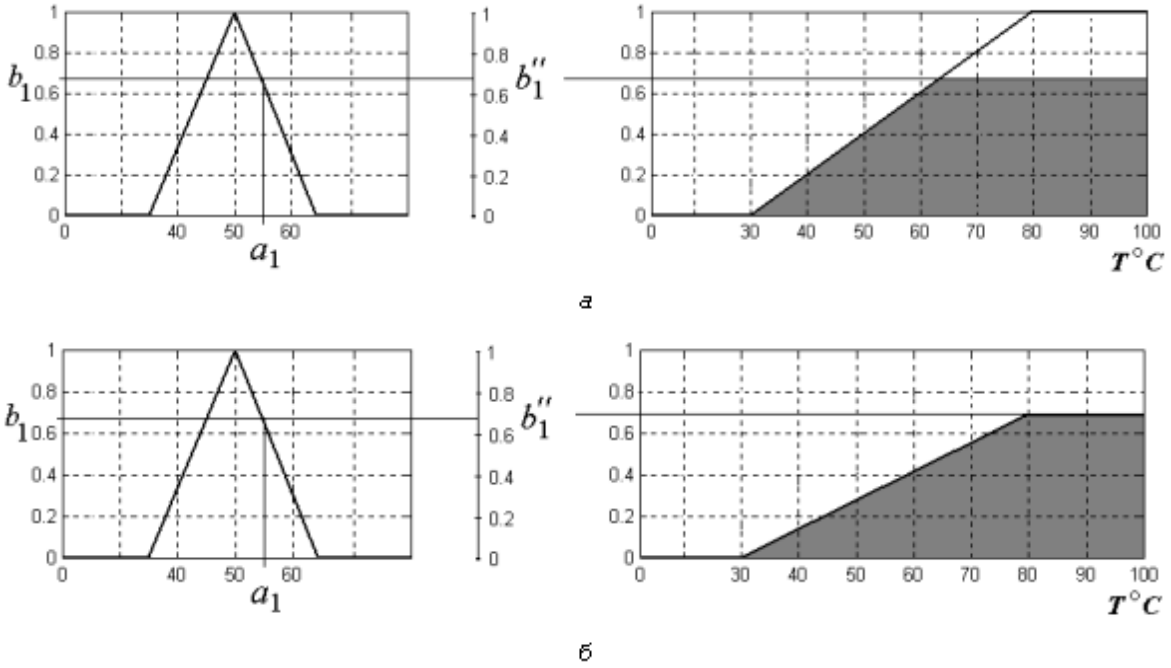


Рис. 6.5 Пример активизации заключения
для правила нечеткой продукции

Поскольку агрегирование условия этого правила дает в результате $b_1'' = 0.67$, а весовой коэффициент равен 1 (по умолчанию), то значение 0.67 будет использоваться в качестве c_1 для получения результата активизации. Результат, полученный методом min-активизации (2.5), изображен на рис. 6.5*а* более темным цветом, а результат, полученный методом prod-активизации (2.6), изображен на рис. 6.5*б* более темным цветом. В этом примере, в отличие от предыдущего, «*температура жидкости*» – выходная лингвистическая переменная.

Рассмотрим пример процедуры активизации (композиции или определение степеней истинности) заключений нечетких продукционных правил для правила № 1 (формула (5.3)).

Суть процедуры расчета состоит в определении модифицированных ФП этих заключений для каждого правила на основе выполнения композиционной операции (в качестве метода нечеткой композиции будем использовать min-

активизацию), модифицированной для нечеткой продукции, между определенным на предыдущем этапе агрегированным значением степеней истинности предпосылок правила, например K_n, и соответствующей ФП его заключения $G_i(y)$:

$$P_n(y) = \min(K_n, G_i(y)), \quad (6.8)$$

где u$\in 0, 0.01 \ldots 1$.

Тогда для правила № 1 (5.3) формула (6.8) примет вид

$$P_1(y) = \min(K_1, G_1(y)),$$

где K_1 – известное значение (0.087) агрегированной степени истинности предпосылок правила № 1, полученное на предыдущем этапе.

Активизированная ФП $P_1(y)$ для правила № 1 (5.3) приведена на рис. 6.6.

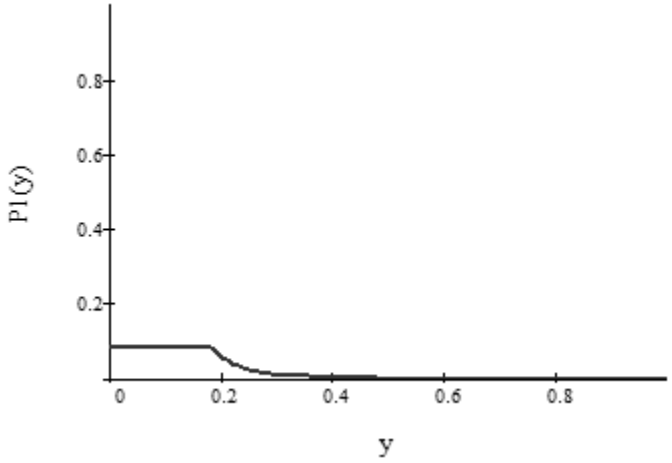


Рис. 6.6 Активизированная ФП $P_1(y)$

Аналогично рассчитываются степени истинности предпосылок по другим правилам.

6.7 Аккумулирование заключений нечетких правил

Аккумулирование заключений нечетких правил в системах нечеткого вывода представляет собой нахождение функции принадлежности для каждой из выходных лингвистических переменных множества $W=\{\omega_1, \omega_2, \ldots, \omega_s\}$.

Целью аккумулирования является объединение всех степеней истинности заключений для получения функции принадлежности каждой из выходных переменных. Необходимость выполнения этого этапа обусловлена тем, что подзаключения, относящиеся к одной и той же выходной лингвистической переменной, принадлежат различным правилам системы нечеткого вывода.

Аккумулирование выполняется следующим образом. Предполагаются известными значения истинности всех подзаключений для каждого из правил R_k, входящих в рассматриваемую базу правил P системы нечеткого вывода, в форме совокупности нечетких множеств: C_1, C_2, …, C_q, где q – общее количество подзаключений в базе правил. Далее последовательно рассматривается каждая из выходных лингвистических переменных $\omega_j \in W$ и относящиеся к ней нечеткие множества: C_{j1}, C_{j2}, …, C_{jq}. Результат аккумуляции для выходной лингвистической переменной ω_j определяется как объединение нечетких множеств C_{j1}, C_{j2}, …, C_{jq} по одной из формул:

max-объединение двух нечетких множеств A и *B* (см. формулу (5.8)):

$$\mu_D(x) = \max\{\mu_A(x), \mu_B(x)\} \quad (\forall x \in \boldsymbol{X});$$

алгебраическая сумма:

$$\mu_D(x) = \mu_A(x) + \mu_B(x) - \mu_A(x)\mu_B(x), \quad (\forall x \in \boldsymbol{X});$$

граничная сумма:

$$\mu_D(x) = \min\{\mu_A(x) + \mu_B(x), 1\} \quad (\forall x \in \boldsymbol{X});$$

драстическая сумма:

$$\boldsymbol{T}(\mathrm{A} \vee \mathrm{B}) = \begin{cases} \boldsymbol{T}(\mathrm{B}), & \text{если } \boldsymbol{T}(\mathrm{A}) = 0; \\ \boldsymbol{T}(\mathrm{A}), & \text{если } \boldsymbol{T}(\mathrm{B}) = 0; \\ 1, & \text{в остальных случаях.} \end{cases}$$

λ-сумма:

$$\mu_D(x) = \lambda\mu_A(x) + (1-\lambda)\,\mu_B(x) \quad (\forall x \in \boldsymbol{X}),\ \lambda \in [0, 1].$$

Этап аккумуляции завершается, когда для каждой из выходных лингвистических переменных будут определены итоговые функции принадлежности нечетких множеств их значений, т. е. совокупность нечетких множеств: C_1', C_2', …, C_s', где s – общее количество выходных лингвистических переменных в базе правил системы нечеткого вывода.

Пример. Рассмотрим аккумулирование заключений для двух нечетких множеств C_{11} и C_{12}, полученных в результате выполнения активизации для выходной лингвистической переменной «*скорость автомобиля*». Пусть функции принадлежности этих нечетких множеств имеют вид как показано на рис. 6.7*а, б* соответственно.

Аккумулирование этих функций принадлежности методом max-объединения нечетких множеств C_{11} и C_{12} по формуле (1.4) позволяет получить в результате функцию принадлежности выходной лингвистической переменной «*скорость автомобиля*», которая представлена на рис. 6.7*в*. Эта функция принадлежности соответствует нечеткому множеству C_1', если принять, что рассматриваемая выходная лингвистическая переменная есть ω_1.

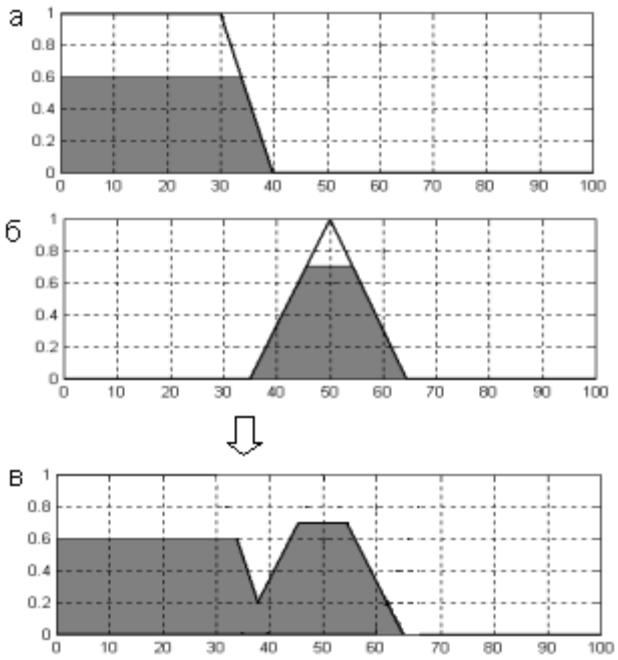


Рис. 6.7 Аккумулирование заключения
с использованием max-объединения

На рис. 6.8 на первых трех графиках слева показаны заключения трех нечетких продукционных правил, а на графике справа – результат их аккумулирования.

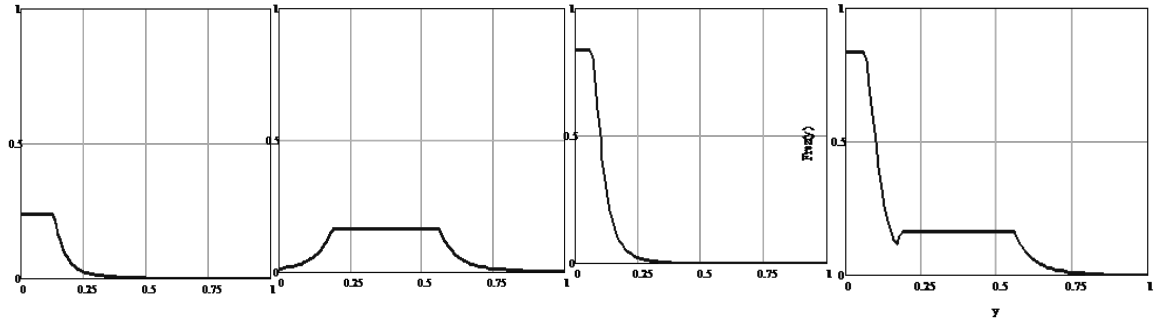


Рис. 6.8 Пример аккумулирования заключений трех правил
с использованием max-объединения

Кроме указанных методов для аккумулирования могут использоваться и другие способы, основанные на модификации различных операций объединения нечетких множеств.

6.8 Приведение результатов к четкости

***Приведение результатов к четкости**, или дефаззификация,* представляет собой процесс получения обычного (не нечеткого) значения для каждой из выходных лингвистических переменных множества $W = \{\omega_1, \omega_2, ..., \omega_s\}$.

При этом, используя результаты аккумуляции всех выходных лингвистических переменных, получают количественное значение каждой из выходных переменных, которое может быть использовано устройствами, внешними по отношению к системе нечеткого вывода.

Этап выполняется следующим образом. Предполагаются известными функции принадлежности всех выходных лингвистических переменных в форме нечетких множеств: X_1', X_2', …, X_s', где s – общее количество выходных лингвистических переменных в базе правил системы нечеткого вывода. Далее последовательно рассматривается каждая из выходных лингвистических переменных $\omega_j \in W$ и относящееся к ней нечеткое множество X_j'. Результат приведения к четкости для выходной лингвистической переменной ω_j определяется в виде количественного значения $y_j \in \nabla$, получаемого по одной из рассматриваемых ниже формул.

Этап завершается, когда для каждой из выходных лингвистических переменных будут определены итоговые количественные значения в форме некоторого действительного числа, т. е. в виде y_1, y_2, ..., y_s, где s – общее количество выходных лингвистических переменных в базе правил системы нечеткого вывода.

Для приведения конечных результатов к четкости чаще всего используют следующие *методы*:

Метод центра тяжести (CoG, Centre of Gravity), или центроид площади, реализуется с помощью формулы

$$y = \frac{\int_{Min}^{Max} x \cdot \mu(x)\, dx}{\int_{Min}^{Max} \mu(x)\, dx}, \qquad (6.9)$$

где y – четкое выходное значение; x – переменная, соответствующая выходной лингвистической переменной ω; $\mu(x)$ – функция принадлежности нечеткого множества, соответствующего выходной переменной ω после этапа аккумуляции; Min и Max – левая и правая точки интервала носителя нечеткого множества рассматриваемой выходной переменной ω.

При использовании этого метода обычное (четкое) значение выходной переменной равно абсциссе центра тяжести площади, ограниченной графиком кривой функции принадлежности соответствующей выходной переменной.

Пример дефаззификации методом центра тяжести функции принадлежности выходной лингвистической переменной «*скорость движения автомобиля*» приведен на рис. 6.9. В этом случае имеем y_1 = 40 км/ч (приближенное значение).

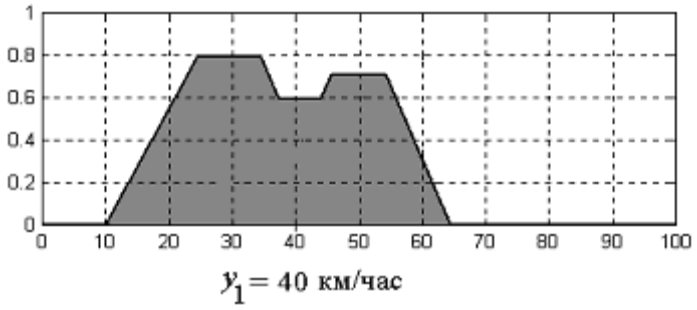


Рис. 6.9 Пример дефаззификации
методом центра тяжести

В случае дискретных функций принадлежности выходной лингвистической переменной *центр тяжести* рассчитывается по формуле

$$y = \frac{\sum_{i=1}^{n} x_i \cdot \mu(x_i)}{\sum_{i=1}^{n} \mu(x_i)},$$

где n – число одноточечных (одноэлементных) нечетких множеств, каждое из которых характеризует единственное значение рассматриваемой выходной лингвистической переменной.

Метод центра площади (CoA, Centre of Area) предполагает, что четкое результирующее значение y определяется из уравнения

$$\int_{Min}^{y} \mu(x)\, dx = \int_{y}^{Max} \mu(x)\, dx .$$

Центр площади равен абсциссе, которая делит площадь, ограниченную графиком кривой функции принадлежности соответствующей выходной переменной, на две равные части. Метод не может быть использован в случае дискретныхфункцийпринадлежности.

Пример приведения к четкости методом центра площади функции принадлежности выходной лингвистической переменной «*скорость движения автомобиля*» приведен на рис. 6.10. В этом случае y_1 = 35 км/ч (приближенное значение).

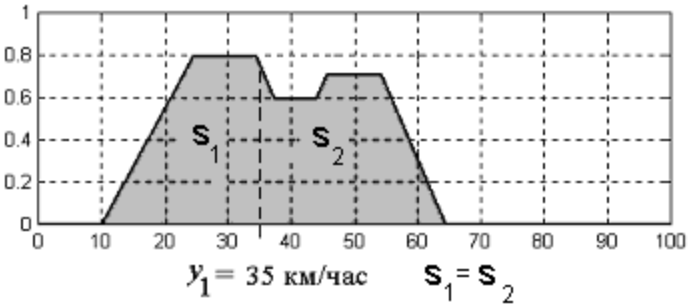


Рис. 6.10 Пример дефаззификации
методом центра площади

Метод левого модального значения (LM, Left Most Maximum), или самого левого максимума, предполагает, что расчет проводится по формуле:

$$y = \min\{x_m\} \quad .$$

Здесь x_m – модальное значение (мода) нечеткого множества, соответствующего выходной переменной ωпосле аккумуляции, рассчитываемое по формуле

$$x_m = \arg\max\{\mu(x)\},\ x \in [a, b]. \qquad (6.10)$$

Значение выходной переменной определяется как мода нечеткого множества для соответствующей выходной переменной или наименьшая из мод (самая левая), если нечеткое множество имеет несколько модальных значений.

Пример дефаззификации методом левого модального значения функции принадлежности выходной лингвистической переменной «*скорость движения автомобиля*» изображен на рис. 6.11. В этом случае y_1 = 24 км/ч (приближенное значение).

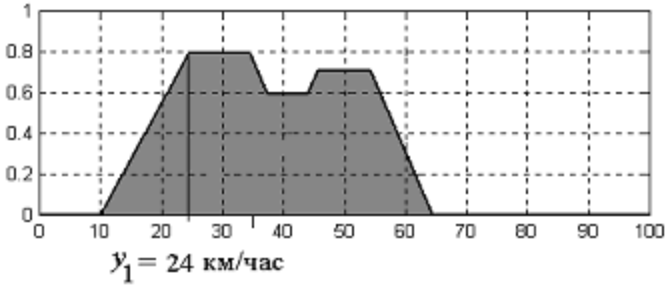


Рис. 6.11 Пример дефаззификации
методом левого модального значения

Метод правого модального значения (RM, Right Most Maximum) или самого правого максимума, предполагает, что расчет проводится по формуле

$$y = \max\{x_m\},$$

где x_m – модальное значение (мода) нечеткого множества для выходной переменной ω после аккумуляции, рассчитываемое по формуле (6.10).

В этом случае значение выходной переменной также определяется как мода нечеткого множества для соответствующей выходной переменной или наибольшая из мод (самая правая), если нечеткое множество имеет несколько модальных значений.

Для строго унимодального нечеткого множества левое и правое модальные значения совпадают.

Пример дефаззификации функции принадлежности выходной лингвистической переменной «*скорость автомобиля*» методом правого модального значения приведен на рис. 6.12. В этом случае y_1 = 54 км/ч (приближенное значение).

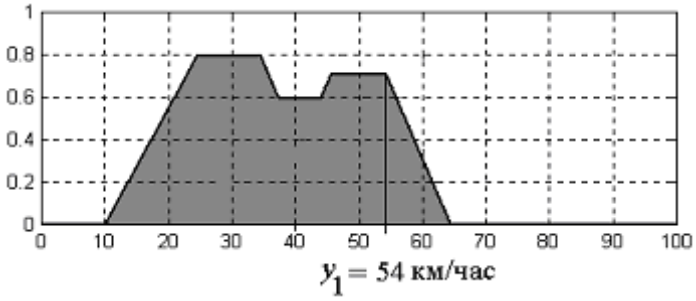


Рис. 6.12 Пример дефаззификации методом правого модального значения

Кроме приведенных методов для приведения результатов к четкости могут быть предложены и другие расчетные формулы. Здесь приводятся лишь те из них, которые нашли наибольшее практическое применение для систем нечеткого вывода.

6.9 Алгоритмы нечеткого вывода

Рассмотренные этапы нечеткого вывода могут быть реализованы по-разному – с использованием различных подходов на каждом из этапов. Выбор конкретных вариантов параметров каждого из этапов определяет алгоритм нечеткого вывода в системе правил нечетких продукций. В настоящее время наибольшее распространение получили следующие алгоритмы нечеткого вывода: Мамдани, Ларсена, Цукамото, Такаги-Сугено.

6.9.1 Алгоритм Мамдани

Алгоритм Мамдани предложен одним из первых для управления паровым двигателем и описывается следующим образом:

1 Формирование базы правил систем нечеткого вывода и введение нечеткости для входных переменных. Предположим, что база состоит из двух правил (рис. 6.13) с двумя входами и одним выходом:

Правило 1: **ЕСЛИ** x_1 есть A_{11} Их$_2$ есть A_{12} **ТО** у есть B_1,

Правило 2: **ЕСЛИ** x_1 есть A_{21} Их$_2$ есть A_{22} **ТО** у есть B_2.

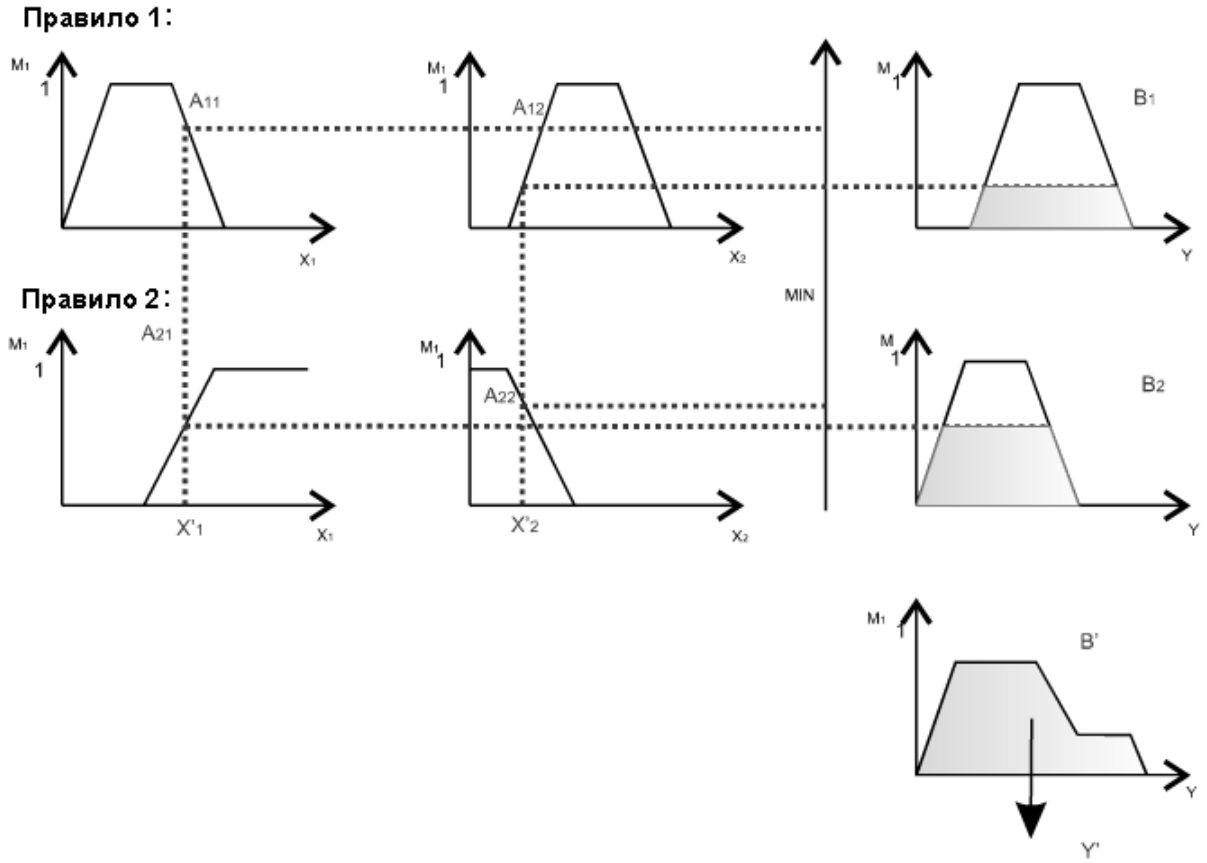


Рис. 6.13 Представление алгоритма Мамдани

2 Агрегирование подусловий в нечетких правилах продукций. Для нахождения степени истинности условий каждого из правил нечетких продукций используются парные нечеткие логические операции. Правила, степень истинности условий которых отлична от нуля, считаются *активными*.

3 Активизация подзаключений в нечетких правилах продукций выполняется с помощью min-активизации по формуле

$$\mu'(y) = \min\{c_i, \mu(y)\};$$

причем, для сокращения времени вывода учитываются только активные правила.

4 Аккумулирование заключений нечетких правил продукций выполняется с помощью max-объединения по формуле (5.8) для объединения нечетких множеств, соответствующих термам подзаключений, относящихся к одним и тем же выходным лингвистическим переменным.

5 Дефаззификация выходных переменных выполняется с использованием метода центра тяжести или метода центра площади.

6.9.2 Алгоритм Ларсена

Алгоритм Ларсена описывается следующим образом:

1 Формирование базы правил систем нечеткого вывода и задание нечеткости для входных переменных выполняется аналогично алгоритму Мамдани.

2 Агрегирование подусловий в нечетких правилах продукций. Используются парные нечеткие логические операции для нахождения степени истинности условий всех правил нечетких продукций (как правило, max-дизъюнкция и min-конъюнкция). Правила, степень истинности условий которых отлична от нуля, считаются *активными* и используются для дальнейших расчетов.

3 Активизация подзаключений в нечетких правилах продукций, в отличие от алгоритма Мамдани, осуществляется с использованием prod-активизации по формуле

$$\mu'(y) = c_i \cdot \mu(y);$$

4 Аккумулирование заключений нечетких правил продукций, как и в алгоритме Мамдани, выполняется с помощью max-объединения для объединения нечетких множеств, соответствующих термам подзаключений, относящихся к одним и тем же выходным лингвистическим переменным.

5 Дефаззификация выходных переменных. Может использоваться любой из рассмотренных выше методов дефаззификации.

6.9.3 Алгоритм Цукамото

Алгоритм Цукамото описывается следующим образом (рис. 6.14).

1 Формирование базы правил систем нечеткого вывода. Предполагается, что функции $\mu(w_j)$ ($\forall i \in \{1, 2, \ldots, q\}$) являются монотонными.

2 Введение нечеткости для входных переменных.

3 Агрегирование подусловий в нечетких правилах продукций. Правила, степень истинности условий которых отлична от нуля, считаются *активными* и используются для дальнейших расчетов.

4 Активизация подзаключений в нечетких правилах продукций, как и в алгоритме Мамдани, выполняется с помощью min-активизации, находятся уровни отсечений c_i. Затем находятся обычные (не нечеткие) значения выходных лингвистических переменных в каждом из подзаключений активных

правил нечетких продукций. Значение выходной лингвистической переменной w_j в каждом из подзаключений находится как решение уравнения:

$$c_i = \mu(w_j) \ \ (\forall i \in \{1, 2, \ldots, q\}),$$

где q – общее количество подзаключений в базе правил.

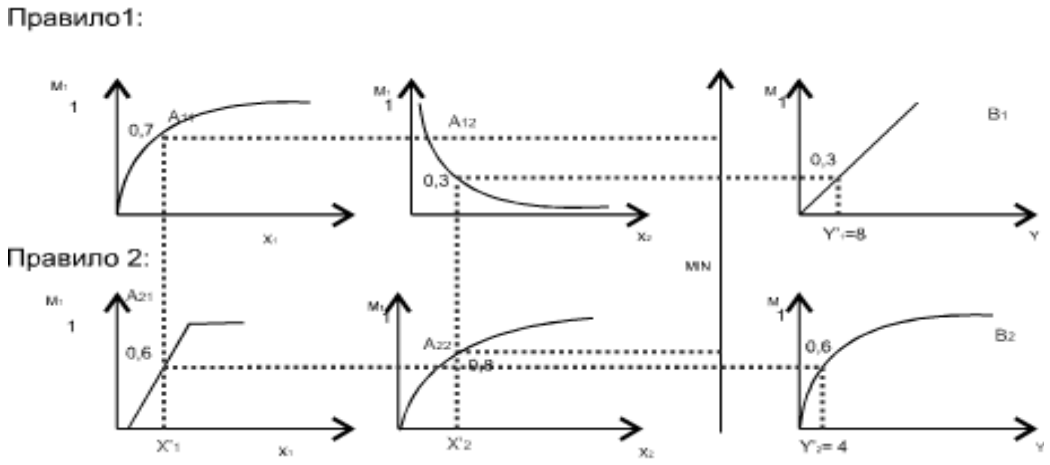


Рис. 6.14 Представление алгоритма Цукамото

5 Аккумуляция заключений нечетких правил продукций не требуется, поскольку расчеты осуществляются с обычными действительными числами w_j.

6 Дефаззификация выходных переменных выполняется с помощью модифицированного варианта метода центра тяжести для одноточечных множеств:

$$y = \frac{\sum_{i=1}^{n} c_i \cdot w_i}{\sum_{i=1}^{n} c_i},$$

где n – общее количество активных правил нечетких продукций, в подзаключениях которых присутствует выходная лингвистическая переменная ω_j.

6.9.4 Алгоритм Такаги-Сугено

Алгоритм Такаги-Сугено, предложенный Сугено и Такаги, описывается следующим образом.

Формирование базы правил систем нечеткого вывода и фаззификация входных переменных. В базе правил используются только правила нечетких продукций (предположим, что база состоит из двух правил с двумя входами и одним выходом) в форме:

Правило 1: **ЕСЛИ** x1 есть A11 И x2 есть A12 **ТО** y = c11 x1 + c12 x2 +c10,

Правило 2: **ЕСЛИ** x1 есть A21 И x2 есть A22 **ТО** y = c21 x1 + c22 x2 +c20.

Здесь cij – весовые коэффициенты компонентов вектора, ci0 – смещение. При этом значение выходной переменной y в заключении определяется как действительное число.

Агрегирование подусловий в нечетких правилах продукций. Для нахождения степени истинности условий всех правил нечетких продукций, как правило, используется логическая операция min-конъюнкции:

$$\alpha 1 = \min \{\mu A11(x1'), \mu A12(x2')\},$$
$$\alpha 2 = \min \{\mu A21(x1'), \mu A22(x2')\}.$$

Для выполнения агрегирования могут использоваться и другие логические операции. Правила, степень истинности условий которых отлична от нуля, считаются *активными* и используются для дальнейших расчетов.

Активизация подзаключений в нечетких правилах продукций. Во-первых, с использованием min-активизации, как и в алгоритме Мамдани, находятся значения степеней истинности всех заключений правил нечетких продукций. Во-вторых, осуществляется расчет обычных (не нечетких) значений выходных переменных каждого правила. Это выполняется с использованием формул для заключения:

$$y_1' = c_{11}x_1' + c_{12}x_2' + c_{10},$$
$$y_2' = c_{21}x_1' + c_{22}x_2' + c_{20}.$$

Здесь вместо x_1 и x_2 подставляются значения входных переменных до этапа фаззификации.

Аккумуляция заключений нечетких правил продукций. Фактически отсутствует, поскольку расчеты осуществляются с обычными действительными числами yj.

Дефаззификация выходных переменных. Используется модифицированный вариант в форме метода центра тяжести для одноточечных множеств:

$$y' = (\alpha_1 y_1' + \alpha_2 y_2')/(\alpha_1 + \alpha_2).$$

При этом не требуется проведения предварительного аккумулирования активизированных заключений отдельных правил.

6.9.5 Сравнение алгоритмов нечеткого вывода

Важное отличие алгоритма Ларсена от алгоритма Мамдани заключается в том, что на этапе 3 в качестве правила нечеткой композиции используется операция prod-активизации [2, 28-32]:

$$P_n(u) = K_n \cdot G_i(u),$$

где $P_n(u)$ – агрегированная функция принадлежности переменной вывода по правилу n;

K_n – агрегированная степень истинности предпосылок правила n;

i – порядковый номер терм-множества для функции принадлежности $G_i(u)$;

u – входная переменная;

$G_i(u)$ – функция принадлежности заключения правила n.

Таким образом, в алгоритме Ларсена используется алгебраическое умножение, а не логическое умножение, как в алгоритме Мамдани [29] (формула (2.8)).

Существенное отличие алгоритма нечеткого вывода Такаги-Сугено от алгоритма Мамдани касается заключений правил, применяемых на этапе 3. В алгоритме Мамдани оно задается с помощью ФП заключения правила (см. табл. 2.3), а в алгоритме Такаги-Сугено – в виде линейной функции от входов. Правила в базе знаний являются своего рода переключателями с одного линейного закона «входы-выход» на другой, тоже линейный [15, 32].

Активизация заключений по каждому i-му правилу в алгоритме Такаги-Сугено выполняется по формуле

$$P_n = a_i z + b_i q , \qquad (6.11)$$

где a_i, b_i – коэффициенты компонентов вектора, z и q – входные данные.

В алгоритме Такаги-Сугено этап 4 отсутствует вследствие четких значений выходных переменных. На этапе 5 в качестве метода дефаззификации используют разновидность метода центра тяжести для одноточечных множеств [47]:

$$\mathrm{F} = \frac{\sum_{i=1}^{l} P_n K_n}{\sum_{i=1}^{l} K_n},$$

где P_n – значение выходной переменной i-го правила, l – количество правил в базе.

Сравнительный анализ трех алгоритмов нечеткого вывода приведен в табл. 6.2.

Анализ показывает, что наименьшее отклонение от средних выходных значений показал алгоритм Мамдани (от 8.6 до 29.3%), наименьшую точность показал алгоритм Такаги-Сугено (отклонение от 24.2 до 59 %). В качестве недостатка алгоритма Такаги-Сугено нужно отметить сложность формирования экспертами заключений по каждому правилу.

Таблица 6.2 **Выходные значения нечетких продукционных моделей**

№	z_i	q_i	Выходные значения нечеткой продукционной модели						M
			Мамдани		Ларсена		Такаги-Сугено		
			$Y_М$	$\Delta_м$	$Y_Л$	$\Delta_л$	$Y_{ТС}$	$\Delta_{ТС}$	
1	0.22	4	0.216	29.3	0.234	40.1	0.076	54.5	0.167
2	0.71	1	0.71	14.8	0.704	15.5	0.631	24.2	0.833
3	0.34	2	0.457	8.6	0.379	24.2	0.205	59	0.5

Результаты, полученные с помощью алгоритма Ларсена, очень близки к результатам, показанным с помощью алгоритма Мамдани (отклонение между ними составляет от 0.7 до 15.6 %).

6.10 Лингвистическая модификация функции принадлежности

Сегодня не существует алгоритм, который позволял бы самой СППР автоматически обнаруживать свои ошибки. Системы с нечеткой логикой не могут автоматически приобретать знания для использования их в механизмах вывода [14]. Ошибки программ могут быть обнаружены только пользователями АС.

Автоматизированное корректирование правил СППР, которые привели к ошибке, позволяет таких ошибок избежать. В работе [5] для повышения точности вывода решения о наличии программных воздействий в СППР предлагается метод лингвистической модификации ФП.

6.10.1 Задача уточнения нечеткого вывода

Как отмечают во многих публикациях, например [2], в нечетких продукционных моделях имеет место экспоненциальное возрастание числа нечетких правил при стремлении к нулю ошибки аппроксимации.

Из ряда работ известно, что при реализации алгоритма нечеткого вывода следует обеспечить два важнейших условия: полноту покрытия возможных значений и приемлемую степень размытости (требуемый уровень четкости) функций принадлежности термов лингвистических переменных в правилах нечеткого вывода. Первое условие при формировании правил нечеткого вывода

выполняется достаточно просто соответствующим набором термов лингвистических переменных.

Чтобы выполнить второе условие, нужно, во-первых, оценить степень размытости функций принадлежности. При этом применительно к решаемой задаче, в нашем случае для нечеткого вывода по управлению системой обнаружения вторжений (СОВ), нужно предварительно задать допустимый уровень размытости функции принадлежности. В случае, если он оказывается больше допустимого, требуется выполнить модификацию всех функций принадлежности с недопустимо высоким уровнем, чтобы поднять четкость.

6.10.2 Оценивание степени размытости функции принадлежности

Для оценивания степени размытости функций принадлежности можно воспользоваться подходами, изложенными в работах [27, 33]. Различают аксиоматический и метрический подходы к определению размытости нечетких множеств [27]. При аксиоматическом подходе показатель размытости нечеткого множества определяют как меру внутренней неопределенности, двусмысленности объектов множества X по отношению к некоторому свойству A, характеризующему эти объекты и определяющему в X нечеткое множество объектов.

Внутренняя неопределенность объекта x по отношению к свойству A проявляется в том, что он, хотя и в разной степени, принадлежит сразу двум противоположным классам: классу объектов, «обладающих свойством A» и классу объектов, «не обладающих свойством A». Эта двусмысленность объекта x по отношению к свойству A максимальна, когда степени принадлежности объекта x к обоим классам равны, т. е. $\mu_A(x) = \mu_{\underline{A}}(x) = 0.5$. Двусмысленность объекта минимальна, когда объект принадлежит только к одному из этих классов, т. е. $\mu_A(x) = 1, \mu_{\underline{A}}(x) = 0$, или $\mu_A(x) = 0, \mu_{\underline{A}}(x) = 1$.

Глобальный показатель размытости нечеткого множества A можно определить в виде функционала d, удовлетворяющего следующим условиям:

1 $d(A) < d(B)$, если A является заострением B, т. е. $\mu_A(x) \le \mu_B(x)$ при $\mu_B(x) < 0.5$, $\mu_A(x) \ge \mu_B(x)$ при $\mu_B(x) > 0.5$ и $\mu_A(x)$ – любое при $\mu_B(x) = 0.5$.

2 $d(A) = d(\overline{A})$.

3 Если $A \cap B = \emptyset$, то $d(A \cup B) = d(A) + d(B)$.

При метрическом подходе показатель размытости нечетких множеств определяют: как меру отличия нечеткого множества от ближайшего к нему обычного множества или с помощью расстояния до максимального размытого

множества $A_{0.5}: \forall x \in \mu_{A_{0.5}}(x) = 0.5$ и расстояния между нечетким множеством и его дополнением.

Множеством, ближайшим к нечеткому множеству A, называют неразмытое множество $\underline{A}$ такое, что

$$\mu_{\underline{A}}(x) = \begin{cases} 1, \text{если } \mu_A(x) > 0.5; \\ 0, \text{если } \mu_A(x) \leq 0.5. \end{cases}$$

В качестве показателя размытости используют функционал вида

$$d(A) = \frac{2}{N}\sum_{j=1}^{N} |\mu_A(x_j) - \mu_{\underline{A}}(x_j)|, \quad (6.12)$$

Он может быть представлен также в виде

$$d(A) = \frac{2}{N}\sum_{j=1}^{N} \mu_{A\cap\overline{A}}(x_j).$$

Если вместо расстояния Хэмминга использовать евклидово расстояние, то имеем:

$$d(A) = \frac{2}{\sqrt{N}}\sqrt{\sum_{j=1}^{N}(\mu_A(x_j) - \mu_{\underline{A}}(x_j))^2}.$$

Показатель размытости можно задать с помощью расстояния между нечетким множеством и его дополнением:

$$d(A) = k\left[\rho(\emptyset, U) - \rho\left(A, \overline{A}\right)\right].$$

В случае метрики Хэмминга $\rho\left(A, \overline{A}\right)$ имеет следующий вид:

$$\rho\left(A, \overline{A}\right) = \sum_{j=1}^{N} |\mu_A(x_j) - \mu_{\overline{A}}(x_j)| = \sum_{j=1}^{N} |2\mu_A(x_j) - 1|.$$

Такой показатель размытости удовлетворяет указанным выше свойствам 1 и 2. Ими мы и воспользуемся. Формула расчета показателя размытости (6.12) для двух ФП может быть преобразована к виду:

$$d(A) = \frac{2}{g-f}\int_f^g \left|\mu_A(x_j) - \mu_{\underline{A}}(x_j)\right| dx_j, \quad (6.13)$$

где $\mu_A(x_j)$ и $\mu_{\underline{A}}(x_j)$ – функции принадлежности элементов нечетким множествам A и $\underline{A}$ соответственно; g, f – границы области определения ФП.

6.10.3 Лингвистические модификаторы

Чтобы повысить степень четкости ФП лингвистических переменных, следуя [33], воспользуемся лингвистическим модификатором. Суть лингвистических модификаторов наглядно представлена на рис. 6.15.

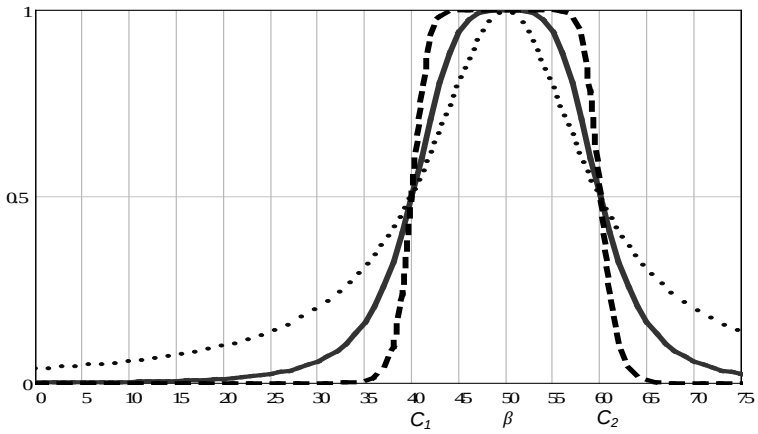


Рис. 6.15 ФП с различными значениями
лингвистического модификатора

Точечно показана исходная колокообразная ФП, непрерывной линией и пунктиром показана эта же ФП для различных значений параметра s – коэффициента лингвистического модификатора [5].

Для заданного нечеткого множества $A^0(x)$ лингвистический модификатор может задаваться с помощью соотношения вида [33]:

$$A(x) = \begin{cases} \dfrac{1}{(\mu c_1)^{s-1}}\left(A^0(x)\right)^s, & x < c_1; \\ 1 - \dfrac{1}{(1-\mu c_1)^{s-1}}\left(1 - A^0(x)\right)^s, & c_1 \le x \le \beta; \\ 1 - \dfrac{1}{(1-\mu c_2)^{s-1}}\left(1 - A^0(x)\right)^s, & \beta \le x \le c_2; \\ \dfrac{1}{(\mu c_2)^{s-1}}\left(A^0(x)\right)^s, & c_2 \le x\,. \end{cases}$$

Здесь: $s \ge 1$ является лингвистическим параметром модификатора для управления нечеткостью, c_1 и c_2 называют левыми и правыми точками пересечения функции принадлежности A, соответственно, и их степени принадлежности определяются следующими уравнениями:

$$\mu c_1 = A^0(c_1) \quad \text{и} \quad \mu c_2 = A^0(c_2).$$

Величина β есть основной центр A, определяемый по формуле

$$\beta = \frac{1}{2}(\beta_1^0 + \beta_2^0),$$

где β_1^0 и β_2^0 суть нижняя и верхняя границы ядра множества $\operatorname{norm} A^0$, соответственно, и $\operatorname{norm} A^0$ есть норма множества A^0, определенная как

$$\operatorname{norm} A^0(x) = \frac{A^0(x)}{\sup\limits_x (A^0(x))}\ ,$$

где $\sup_x(A^0(x))$ – означает верхнюю границу множества элементов $A^0(x)$.

Влияя на степень размытости входной ФП с помощью изменения коэффициента лингвистического модификатора, можно оказывать влияние и на результат выходной ФП.

6.10.4 Влияние коэффициента лингвистической модификации функции принадлежности на степень уверенности системы поддержки принятия решения в наличии программных воздействий в автоматизированной системе управления

Рассмотрим пример модификации входных функций принадлежности, задаваемых при нечетком выводе в системе поддержки принятия решения (СППР) по обнаружению программных воздействий (ПВ). Для оценки влияния значения коэффициента лингвистического модификатора (КЛМ) ФП на степень уверенности о наличии ПВ (Y) построена модель СППР в среде MathCad 14. В табл. 6.3 представлены исходные данные для шести входных переменных трех различных примеров [5].

В табл. 6.4 представлены результаты оценки степени уверенности в наличии ПВ в АСУ для разных исходных данных (разные примеры, разные значения КЛМ s, отсутствие и наличие ограничений > 5 % для расчета степени истинности предпосылок правил). Расчет относительных погрешностей выполнен по формуле: $\Delta Y_s = (|\min Y_s - Y_{s=2}|) \times 100/Y_{s=2}$.

Таблица 6.3 **Исходные данные**

№ примера	Входные переменные					
	$\mathbf{u_1}$	$\mathbf{u_2}$	$\mathbf{u_3}$	$\mathbf{u_4}$	$\mathbf{u_5}$	$\mathbf{u_6}$
1	1	4	35	0	0	12.5
2	3	6	15	0	0	11.5
3	3	7	30	0	0	16.5

Как следует из табл. 6.4, значения относительных погрешностей ΔY_s, при которых имеется снижение размытости, находятся в диапазоне от 31 (29.6) до 43.1 (46) %. В скобках указаны значения относительных погрешностей ΔY_s с пороговым ограничением > 5 %.

Изменение значения КЛМ позволяет снизить степень размытости выходной ФП, что может привести к уменьшению степени уверенности в наличии ПВ, тем самым обеспечивается уменьшение числа возможных ошибок первого рода.

Таблица 6.4 **Значения степени уверенности**

s	Степени уверенности Y_s для примеров					
	1		2		3	
	ограничения					
	нет	есть	нет	есть	нет	есть
2	0.261	0.253	0.273	0.273	0.385	0.385
3	0.227	0.227	0.179	0.178	0.284	0.268
4	0.207	0.178	0.232	0.232	0.358	0.352
5	0.195	0.178	0.178	0.178	0.219	0.208
6	0.187	0.178	0.206	0.203	0.384	0.5
7	0.183	0.178	0.178	0.178	0.233	0.5
8	0.18	0.178	0.191	0.189	0.427	0.5
	Относительные погрешности ΔY_s (%)					
	31	29.6	34.8	34.8	43.1	46

При переборе КЛМ от 3 до 8 и сравнении полученных модифицированных результатов с оценкой степени уверенности в наличии ПВ (при использовании КЛМ по умолчанию, s = 2) приходим к следующему выводу. Всегда имеется такой КЛМ, применение которого приводит к снижению размытости выходной ФП и может оказывать влияние на нечеткий вывод (например, лингвистический терм «ниже среднего» будет заменен на «низкая»). Осуществлять перебор КЛМ более 8 не целесообразно, так как это не приводит к снижению степени уверенности СППР в наличии ПВ в АСУ.

На рис. 6.16 приведены графики выходной ФП (без пороговых ограничений), которая определяет степень уверенности в наличии ПВ в АСУ где пунктирной линией показано значение Y_s. Подписи под графиками имеют следующий смысл: строка 1s3 означает, что график относится к входным данным примера № 1 и s=3 [5].

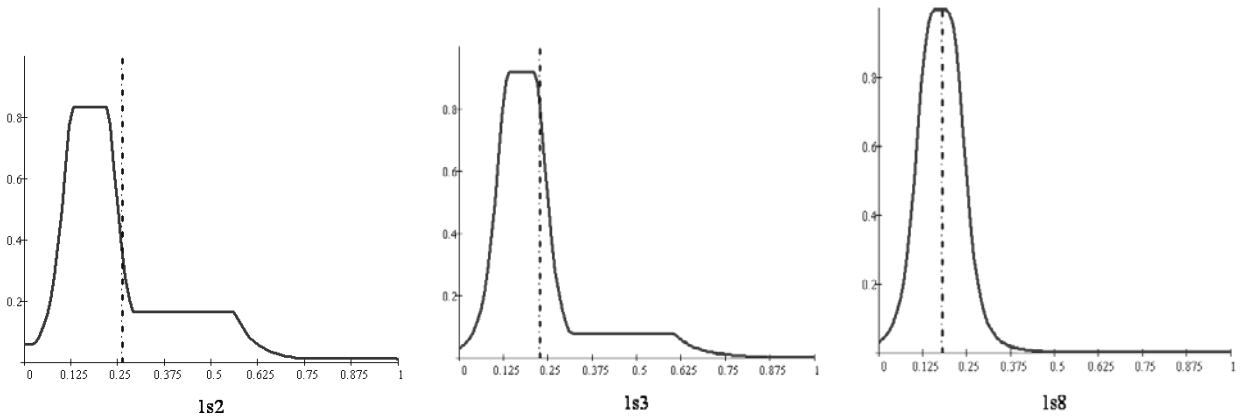


Рис. 6.16 Графики выходной функции принадлежности для примера № 1

При формировании алгоритма работы СППР при расчете степени истинности предпосылок каждого правила целесообразно использовать пороговое ограничение >5%.

Это дает следующие преимущества:

- снижается сложность алгоритма нечеткого вывода, так как не рассчитываются степени уверенности правил в наличии ПВ в АСУ которые являются малозначительными;
- снижается сложность алгоритма нечеткого вывода, так как $\min Y_s$ находится за меньшее число переборов s;
- итоговое значение ΔY_s с ограничением > 5 % отличается от стандартного несущественно (от 1 до 2.9 %).

Предложенный алгоритм уточнения базы правил нечеткого вывода на основе применения лингвистического модификатора позволяет улучшить точность решения по определению ПВ в АСУ. Ключевым местом алгоритма уточнения базы правил нечеткого вывода на основе лингвистической модификации функции принадлежности является необходимость автоматического поиска значения КЛМ, дающего наилучший результат по точности расчета степени уверенности в наличии ПВ в АСУ [5].

7 НЕЙРОННЫЕ СЕТИ

7.1 Характеристика искусственных нейронных сетей

В настоящее время искусственные нейронные сети находят все более широкое исследование и практическое прменение как важнейшая разновидность систем, основанных на знаниях. ***Искусственные нейронные сети*** (ИНС) представляют собой математические модели и их аппаратно-программные реализации, которые основаны на принципе организации функционирования биологических нейронных сетей нервных клеток живыв организмов. С точки зрения структурного построения искусственная нейроная сеть представляет собой систему взаимодействующих искусственных нейронов.

К числу возможных вариантов применения нейронных сетей относятся следующие: распознавание образов и речи, диагностика, обработка сигналов, адаптивное управление, аппроксимация функционалов, прогнозирование, создание экспертных систем, финансовые задачи и др.

Ввиду заметного различия решаемых задач в настоящее время не создаются универсальные нейронные сети, а разрабатываются специализированные НС, ориентированные на решение отдельных практических задач. К общим чертам искусственных нейронных сетей можно отнести следующее:

1 Основу искусственной нейронной сети составляют простые, зачастую однотипные, элементы (ячейки), имитирующие работу нейронов головного мозга.

2 В нейронной сети реализуется принцип параллельной обработки сигналов, который достигается путем объединения большого числа нейронов в так называемые слои и соединения определенным образом нейронов различных слоев, а также, в некоторых конфигурациях, и нейронов одного слоя между собой, причем обработка взаимодействия всех нейронов ведется послойно.

7.2 Искусственный нейрон

Под ***искусственным нейроном*** или просто нейроном подразумевается ячейка нейронной сети. По аналогии с нервными клетками головного мозга, которые могут быть возбуждены или заторможены, нейрон характеризуется текущим состоянием.

Искусственный нейрон (рис. 7.1), как и биологический, состоит из синапсов, связывающих входы нейрона с ядром; ядра нейрона, которые осуществляют обработку входных сигналов, и аксона, который связывает нейрон с нейронами следующего слоя.

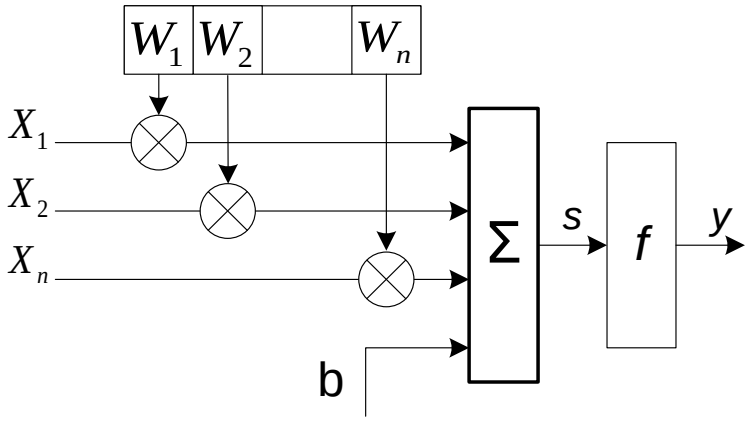


Рис. 7.1 Структура искусственного нейрона

Каждый синапс имеет вес, который определяет, насколько соответствующий вход нейрона влияет на его состояние. Состояние нейрона определяется по формуле

$$S = \sum_{i=1}^{n} x_i \cdot W_i + b \qquad (7.1)$$

,

где W_i – вес (weight) нейрона, $i=1…n$; b – значение смещения, S – результат суммирования, x_i – элемент входного вектора (входной сигнал), $i=1…n$; y – выходной сигнал нейрона, n – число синапсов (связей) нейрона, f – нелинейное преобразование (функция активации нейрона).

Значения входных сигналов, весовых коэффициентов и смещений могут принимать любые значения из области значений функции активации нейрона.

Состояние нейрона является основным параметром для функции активации нейрона, которая отвечает за формирование итогового значения нейрона, которое будет передано далее:

$$Y = f(S), \qquad (7.2)$$

где $f(S)$ – некоторая функция, которая называется активационной. Нелинейная функция f(s) может иметь различный вид, как показано на рис. 7.2.

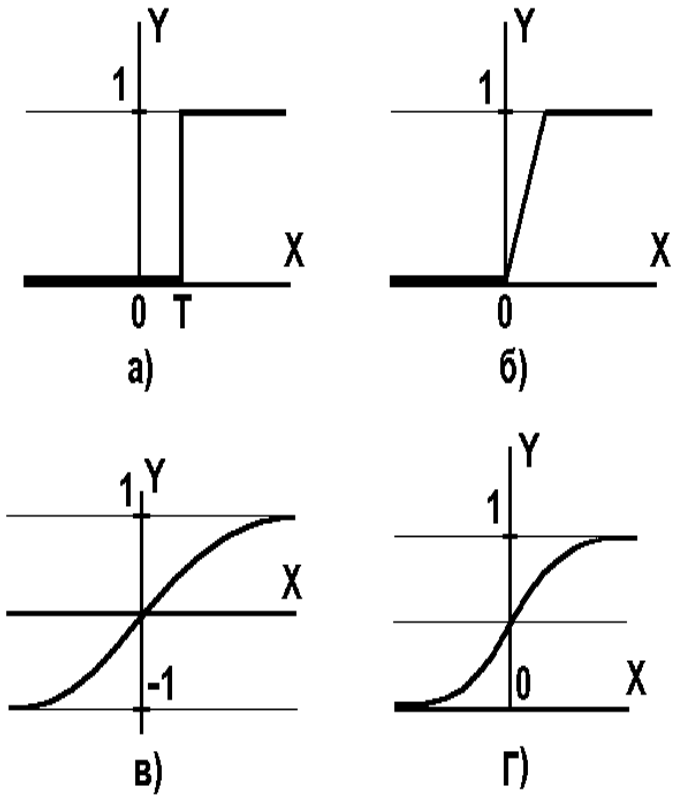


Рис. 7.2 Виды активационной функции:
а) функция единичного скачка; б) линейный порог (гистерезис);
в) сигмоид – гиперболический тангенс; г) сигмоид

В табл. 7.1 приведены наиболее распространенные функции активации.

Таблица 7.1 **Функции активации нейронов**

Название	Формула	Область значений
Линейная	$f(s) = k \cdot s$	(–∞,+∞)
Полулинейная	$f(s) = \begin{cases} k \cdot s, & s > 0 \\ 0, & s \le 0 \end{cases}$	(0,+∞)
Логистическая (сигмоидальная)	$f(s) = \dfrac{1}{1 + e^{-\alpha \cdot s}}$	**(0,1)**
Гиперболический тангенс (сигмоидальная)	$f(s) = \dfrac{e^{a \cdot s} - e^{-a \cdot s}}{e^{a \cdot s} + e^{-a \cdot s}}$	(–1,1)
Экспоненциальная	$f(s) = e^{-a \cdot s}$	(0,+∞)
Синусоидальная	$f(s) = Sin(s)$	(–1,1)
Сигмоидальная (рациональная)	$f(s) = \dfrac{s}{a + \lvert s \rvert}$	(–1,1)
Шаговая (линейная с насыщением)	$f(s) = \begin{cases} -1, & s \le -1 \\ s, & -1 < s < 1 \\ 1, & s \ge 1 \end{cases}$	(–1,1)
Пороговая	$f(s) = \begin{cases} 0, & s < 0 \\ 1, & s \ge 0 \end{cases}$	(0,1)
Модульная	$f(s) = \lvert s \rvert$	(0,+∞)
Знаковая (сигнатурная)	$f(s) = \begin{cases} 1, & s > 0 \\ -1, & s \le 0 \end{cases}$	(–1,1)
Квадратичная	$f(s) = s^2$	(0,+∞)

Одной из наиболее распространенных является нелинейная функция с насыщением, так называемая логистическая функция или сигмоид (функция *S*-образного вида)

$$f(x) = \frac{1}{1 + e^{-\alpha x}}. \qquad (7.3)$$

При уменьшении α сигмоид становится более пологим, в пределе при $\alpha = 0$, вырождаясь в горизонтальную линию на уровне 0.5; при увеличении α сигмоид приближается по виду к функции единичного скачка с порогом T в точке x = 0.

Из выражения для сигмоида очевидно, что выходное значение нейрона лежит в диапазоне [0,1]. Одно из ценных свойств сигмоидной функции – простое выражение для ее производной.

$$f'(x) = \alpha \cdot f(x) \cdot (1 - f(x)). \qquad (7.4)$$

Отметим, что сигмоидная функция дифференцируема на всей оси абсцисс, что используется в некоторых алгоритмах обучения. Кроме того, она обладает свойством усиливать слабые сигналы лучше, чем большие, и предотвращает насыщение от больших сигналов, так как они соответствуют областям аргументов, где сигмоид имеет пологий наклон.

7.3 Обобщенная модель искусственной нейронной сети

В общепринятом представлении ИНС – набор нейронов, соединенных между собой. Некоторые входы нейронов помечены как внешние входы сети, а некоторые выходы – как внешние выходы сети. Подавая любые числа на входы нейронной сети, мы получаем какой-то набор чисел на выходах сети. В обобщенном виде ИНС состоит из 3 групп нейронов (рис. 7.3).

Работа ИНС состоит в преобразовании входного вектора в выходной вектор, причем это преобразование задается весами сети. Под понятием ИНС также подразумеваются вычислительные структуры, которые моделируют простые биологические процессы, обычно ассоциируемые с процессами, свойственными человеческому мозгу. Адаптируемые и обучаемые, они представляют собой распараллеленные системы, способные к обучению путем анализа положительных и отрицательных воздействий, генерируемых учителем на основе успешного или неуспешного решения задачи нейронной сетью. Подобно биологической нейронной системе ИНС является вычислительной системой

с огромным числом параллельно функционирующих простых процессоров с множеством связей.

По данным исследований, проводившихся в рамках проекта SyNAPSE – Systems of Neuromorhic Adaptive Plastic Scalable Electronics (Системы нейроморфической адаптивной пластически масштабируемой электроники), участие в котором принимает ряд ведущих университетов США и корпорация IBM, показывают, что в среднем мозг человека содержит порядка $2*10^{12}$ нейронов и $2*10^{14}$ синапсов.

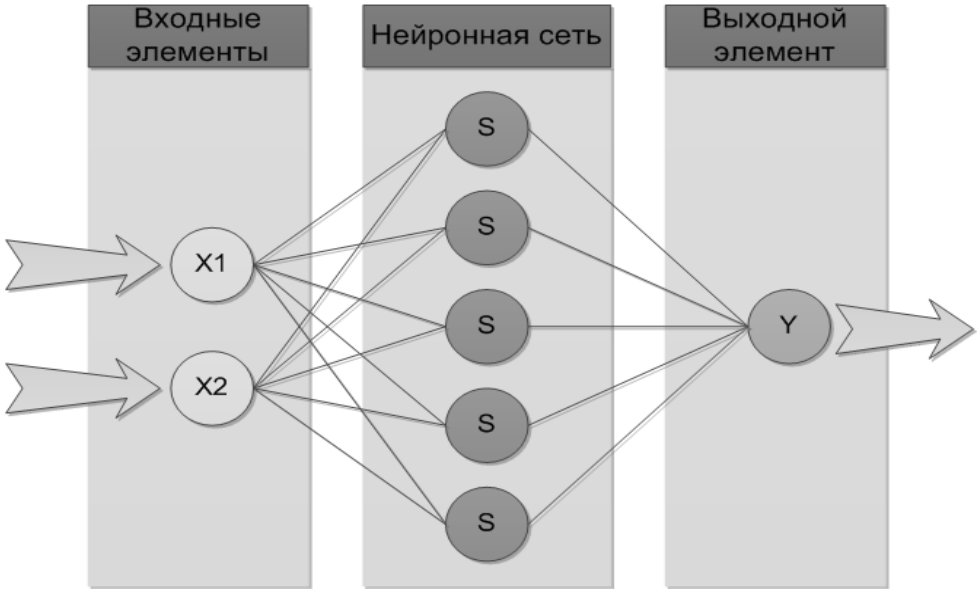


Рис.7.3 Обобщенная модель ИНС

К основным результатам, достигнутым в рамках проекта, следует отнести создание полноценной модели мозга кошки, содержащего порядка $7{,}63 \cdot 10^8$ нейронов и $6{,}1 \cdot 10^{12}$ синапсов. Работоспособность модели обеспечивалась суперкомпьютером Blue Gene/P, построенном на базе 147 456 процессоров и 144 Тб оперативной памяти.

7.4 Классификация нейронных сетей

ИНС может рассматриваться как направленный граф со взвешенными связями, в котором искусственные нейроны являются узлами. По архитектуре связей ИНС могут быть сгруппированы в два класса (рис. 7.4):

1. Нейронные сети прямого распространения.
2. Нейронные сети обратного распространения (рекуррентные).

ИНС сети прямого распространения характеризуются четкой направленностью движения импульса от слоя к слою, начиная с входного слоя и заканчивая результирующим. Сети прямого распространения являются статическими в том смысле, что на заданный вход они вырабатывают одну совокупность выходных значений, не зависящих от предыдущего состояния сети. В сетях обратного распространения результат зависит как от входных данных, так и от значений, получаемых на выходах нейронов. Рекуррентные сети являются динамическими, так как в силу обратных связей в них модифицируются входы нейронов, что приводит к изменению состояния сети.

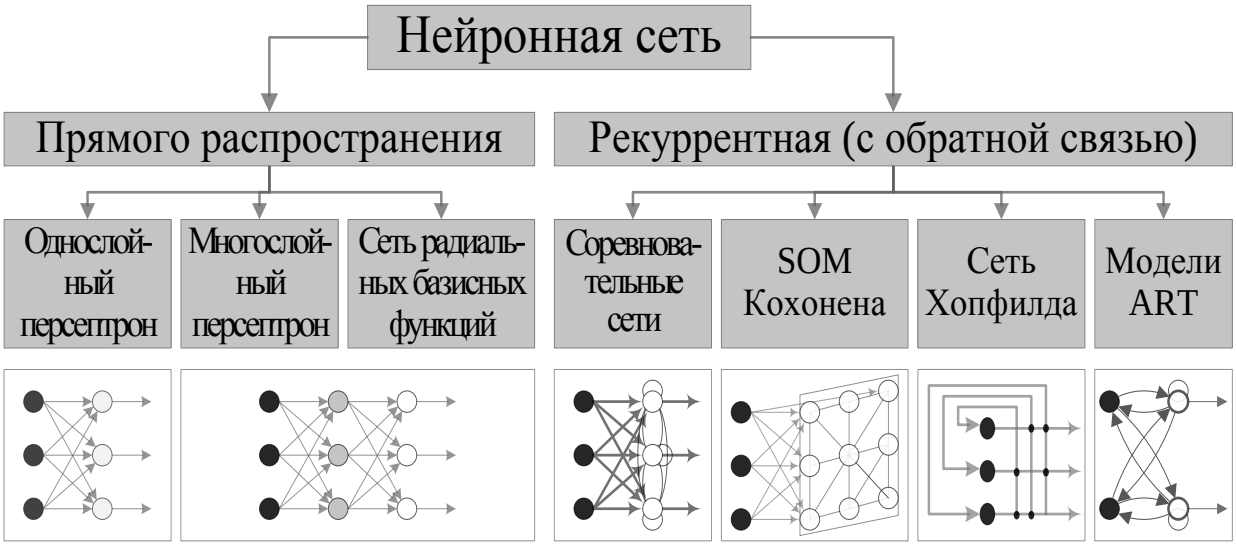


Рис. 7.4 Классификация нейронных сетей

Наиболее подходящими ИНС для использования в СППР являются нейронные сети прямого распространения, из которых многослойный персептрон представляет более широкие возможности в решении задач и при этом достаточно прост в реализации. ИНС прямого распространения являются универсальным средством аппроксимации функций, что позволяет их использовать в решении задач классификации. Эффективность применения ИНС обусловлена тем, что они генерируют большое число регрессионных моделей.

Сравнение сетей RBF и многослойных персептронов. Сети на основе радиальных базисных функций (RBF) и многослойный персептрон (MLP) являются примерами нелинейных многослойных сетей прямого распространения. И те, и другие являются универсальными аппроксиматорами. Таким образом, неудивительно, что всегда существует сеть RBF, способная имитировать многослойный персептрон (и наоборот). Однако эти два типа сетей отличаются по некоторым важным аспектам:

1 Сети RBF (в своей основной форме) имеют один скрытый слой, в то

время как многослойный персептрон может иметь большее количество скрытых слоев.

2 Обычно вычислительные (computational) узлы многослойного персептрона, расположенные в скрытых и выходном слоях, используют одну и ту же модель нейрона. С другой стороны, вычислительные узлы скрытого слоя сети RBF могут в корне отличаться от узлов выходного слоя и служить разным целям.

3 Скрытый слой в сетях RBF является нелинейным, в то время как выходной линейным. В то же время скрытые и выходной слои многослойного персептрона, используемого в качестве классификатора, являются нелинейными. Если многослойный персептрон используется для решения задач нелинейной регрессии, в качестве узлов выходного слоя обычно выбираются линейные нейроны.

4 Аргумент функции активации каждого скрытого узла сети RBF представляет собой Евклидову норму (расстояние) между входным вектором и центром радиальной функции. В то же время аргумент функции активации каждого скрытого узла многослойного персептрона – это скалярное произведение входного вектора и вектора синаптических весов данного нейрона.

5 Многослойный персептрон обеспечивает глобальную аппроксимацию нелинейного отображения. С другой стороны, сеть RBF с помощью экспоненциально уменьшающихся локализованных нелинейностей (т. е. функций Гаусса) создает локальную аппроксимацию нелинейного отображения. Отметим, что для аппроксимации нелинейного отображения с помощью многослойного персептрона может потребоваться меньшее число параметров, чем для сети RBF при одинаковой точности вычислений.

7.5 Характеристика процесса обучения искусственной нейронной сети

Важным свойством нейронных сетей является их способность обучаться на основе данных окружающей среды и в результате обучения повышать свою производительность. Повышение производительности происходит со временем в соответствии с определенными правилами. Обучение нейронной сети происходит посредством интерактивного процесса корректировки синаптических весов и порогов. В идеальном случае нейронная сеть получает знания об окружающей среде на каждой итерации процесса обучения.

С понятием обучения ассоциируется довольно много видов деятельности, поэтому сложно дать этому процессу однозначное определение. Более того, процесс обучения зависит от точки зрения на него. С позиции нейронной сети, обучение – это процесс, в котором свободные параметры нейронной сети настраиваются посредством моделирования среды, в которую эта сеть встроена. Тип обучения определяется способом подстройки этих параметров.

Это определение процесса обучения предполагает следующую последовательность событий:

1 В нейронную сеть поступают стимулы из внешней среды.

2 В результате этого изменяются свободные параметры нейронной сети.

3 После изменения внутренней структуры нейронная сеть отвечает на возбуждения уже иным образом.

В процессе обучения ИНС просматривает обучающую выборку. Порядок просмотра может быть последовательным, случайным и т. д. Некоторые сети, обучающиеся *без учителя*, например, сети Хопфилда просматривают выборку только один раз. Другие, например, сети Кохонена, а также сети, обучающиеся с учителем, просматривают выборку множество раз, при этом один полный проход по выборке называется эпохой обучения (рис. 7.5).

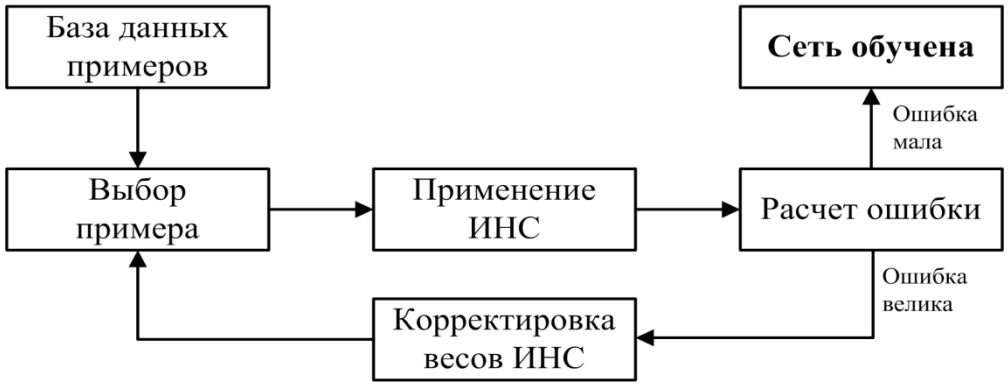


Рис. 7.5 Схема обучения ИНС

При обучении с учителем набор исходных данных делят на две части: обучающую выборку и тестовые данные. Принцип разделения может быть произвольным. Обучающие данные подаются сети для обучения, а проверочные используются для расчета ошибки сети. Таким образом, если на проверочных данных ошибка уменьшается, то сеть действительно выполняет обобщение. Если ошибка на обучающих данных продолжает уменьшаться, а ошибка на тестовых данных увеличивается, значит, сеть перестала выполнять обобщение и просто «запоминает» обучающие данные. Это явление называется переобучением сети или оверфиттингом. В таких случаях обучение обычно прекращают. В процессе обучения могут проявиться другие проблемы, такие как паралич или попадание сети в локальный минимум поверхности ошибок.

В 1970-х годах Вербос разработал подходящий алгоритм корректировки весов – алгоритм обратного распространения ошибки. Дальнейшее развитие этого алгоритма корректировки весов связано с работами Румелхарта и др.

Алгоритм обратного распространения определяет два потока в сети:

1 Прямой поток – от входного слоя к выходному. Продвигает входные векторы через сеть, в результате чего в выходном слое генерируются результирующие значения.

2 Обратный поток – от выходного слоя к входному. Обратный поток продвигает значение ошибки в обратном направлении, начиная с выходного слоя нейронов, в результате чего определяется величина корректирующего воздействия на весовые коэффициенты нейронных связей.

Простейший и самый распространенный пример оценки – сумма квадратов расстояний от выходных сигналов сети до их требуемых значений:

$$E(w) = \frac{1}{2} \cdot \sum_{k=1}^{M} (y_k - d_k)^2 \quad (7.5)$$

.

Как правило, обучение ИНС производится методом *градиентного спуска*, т. е. на каждой итерации изменение веса производится по формуле

$$\Delta w_{i,j} = -\eta \cdot \frac{\partial E}{\partial w_{i,j}}, \quad (7.6)$$

где η – параметр, определяющий скорость обучения.

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial S_j} \cdot \frac{\partial S_j}{\partial w_{i,j}}, \quad (7.7)$$

где y_j – значение выхода j-го нейрона, S_j – взвешенная сумма входных сигналов. При этом множитель определяется по формуле

$$\frac{\partial S_j}{\partial w_{i,j}} \equiv x_i, \quad (7.8)$$

где x_i – значение i-го входа нейрона. Далее рассмотрим определение первого множителя формулы:

$$\frac{\partial E}{\partial y_j} = \sum_k \frac{\partial E}{\partial y_k} \cdot \frac{\partial y_k}{\partial S_k} \cdot \frac{\partial S_k}{\partial w_j} = \sum_k \frac{\partial E}{\partial y_k} \cdot \frac{\partial y_k}{\partial S_k} \cdot w_{j,k}^{(n+1)}, \quad (7.9)$$

где k – число нейронов в слое n+1. Введем вспомогательную переменную

$$\delta_j^{(n)} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial S_j}. \quad (7.10)$$

Тогда мы сможем определить рекурсивную формулу для определения *n*-го слоя, если нам известно следующего *(n+1)*-го слоя.

$$\delta_j^{(n)} = \left[\sum_k \delta_k^{(n+1)} \cdot w_{jk}^{(n+1)} \right] \cdot \frac{\partial y_j}{\partial S_j}. \quad (7.11)$$

Нахождение для последнего слоя НС не представляет трудности, так как нам известен целевой вектор, т. Е. вектор тех значений, которые должна выдавать НС при данном наборе входных значений.

$$\delta_j^{(n)} = \left(y_i^{(n)} - d_i \right) \cdot \frac{\partial y_i}{\partial S_i}. \quad (7.12)$$

И наконец запишем формулу (7.6) в раскрытом виде:

$$\Delta w_{i,j}^{(n)} = -\eta \cdot \delta_j^{(n)} \cdot x_i^{(n)}. \quad (7.13)$$

Полный алгоритм обучения нейронной сети состоит в следущем:

1 Подать на вход ИНС один из требуемых образов и определить значения выходов нейронов.

2 Рассчитать для выходного слоя по формуле (7.12) и рассчитать изменения весов выходного слоя N по формуле (7.13).

3 Рассчитать по формулам (7.11) и (7.13) соответственно и $\Delta w_{i,j}^{(n)}$ для остальных слоев ИНС, n = N-1..1.

4 Скорректировать все веса НС:

$$w_{i,j}^{(n)}(t) = w_{i,j}^{(n)}(t-1) + \Delta w_{i,j}^{(n)}(t).$$

5 Если ошибка существенна, то осуществить корректировку весов повторно.

В процессе обучения сети, поочередно предъявляются вектора из обучающей последовательности, т. е. сети предъявляется образец и вычисляется вектор ошибок, в результате чего выясняется, насколько следует изменить значения весов. Процесс повторяется для каждого образца. Все образцы подаются на рассмотрение сети снова и снова, до тех пор, пока на протяжении одной эпохи все значения реального вывода для каждого образца не попадут в допустимые рамки.

Отметим, что процесс обучения является уникальным для каждой новой задачи. Например, при решении задачи верификации соответствия ПО требованиям безопасности, важнейшим требованием к обученной ИНС является минимальное количество ошибок при принятии решения.

Часть II ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ ЭКСПЕРТНЫХ СИСТЕМ

Экспертные системы (ЭС) – это системы искусственного интеллекта (интеллектуальные системы), предназначенные для решения плохоформализованных и слабоструктурированных задач в определенных проблемных областях, на основе заложенных в них знаний специалистов-экспертов. В настоящее время ЭС внедряются в различные виды человеческой деятельности, где использование точных математических методов и моделей затруднительно или вообще невозможно. К ним относятся: медицина, обучение, поддержка принятия решений и управление в сложных ситуациях, деловые различные приложения и т. д.

Основными компонентами ЭС являются базы данных (БД) и знаний (БЗ), блоки поиска решения, объяснения, извлечения и накопления знаний, обучения и организации взаимодействия с пользователем. БД, БЗ и блок поиска решений образуют ядро ЭС.

Инструментальное средство разработки экспертных систем – это язык программирования, используемый инженером знаний или (и) программистом для построения экспертной системы. Этот инструмент отличается от обычных языков программирования тем, что обеспечивает удобные способы представления сложных высокоуровневых понятий.

По своему назначению и функциональным возможностям инструментальные программы, применяемые при проектировании экспертных систем, можно разделить на четыре достаточно большие категории [9]:

1 Оболочки экспертных систем – программный продукт, обладающий средствами представления знаний для определенных предметных областей. Задача пользователя заключается не в непосредственном программировании, а в формализации и вводе знаний с использованием предоставленных оболочкой возможностей. Недостатком этих систем можно считать невозможность охвата одной системой всех существующих предметных областей. Примером могут служить Exsys Developer 8.0, VP-Expert, система EMYCIN, созданная на основе прошедшей длительную «обкатку» системы MYCIN.

2 Языки программирования высокого уровня. Экспертные системы, выполненные в виде отдельных программ, на некотором алгоритмическом языке, база знаний которых является непосредственно частью этой программы. Как правило, такие системы предназначены для решения задач в одной фиксированной предметной области. При построении таких систем

применяются как традиционные процедурные языки PASCAL, C и др., так и специализированные языки искусственного интеллекта LISP, PROLOG.

3 Генераторы экспертных систем или среда программирования, поддерживающая несколько парадигм (multiple-paradigm programming environment) – мощные программные продукты, предназначенные для получения оболочек, ориентированных на то или иное представление знаний в зависимости от рассматриваемой предметной области. Примеры этой разновидности – системы KEE, ART и др. Эти программы предоставляют в распоряжение квалифицированного пользователя множество опций и для последующих разработок, таких как KAPPA и CLIPS, и стали своего рода стандартом.

4 Дополнительные модули. Средства этой категории представляют собой автономные программные модули, предназначенные для выполнения специфических задач в рамках выбранной архитектуры системы решения проблем. Так математические расчеты нечетких продукционных систем можно осуществлять с помощью компьютерных средств автоматизации математических расчетов: Eureka, Gauss, TK Solver, Derive, Matlab, MathCad, Mathematica, Maple и др.

8 ПОСТРОЕНИЕ НЕЧЕТКИХ ПРОДУКЦИОННЫХ СИСТЕМ В СИСТЕМЕ VISUAL PROLOG

Пролог (Prolog) – язык логического программирования, основанный на логике дизъюнктов Хорна, представляющей собой подмножество логики предикатов первого порядка.

Разработка языка Prolog началась в 1970 г. Аланом Кулмероэ и Филиппом Русселом. Будучи декларативным языком программирования, Пролог воспринимает в качестве программы некоторое описание задачи, и сам производит поиск решения, пользуясь механизмом бэктрекинга и унификацией. Целью разработки языка Prolog было предоставить возможность задания спецификаций решения и позволить компьютеру вывести из них последовательность выполнения для этого решения, а не задание алгоритма решения задачи, как в большинстве языков.

Visual Prolog – объектно-ориентированное расширение языка программирования PDC Prolog, развивавшегося из Turbo Prolog (Borland), семейства Prolog, а также система визуального программирования датской

фирмы Prolog Development Center. Prolog Development Center затратил более трех лет на разработку системы Visual Prolog с поэтапным бета-тестированием, поставки коммерческой версии которой начались с февраля 1996 года.

Visual Prolog автоматизирует построение сложных процедур и освобождает программиста от выполнения тривиальных операций. С помощью Visual Prolog проектирование пользовательского интерфейса и связанных с ним окон, диалогов, меню, строки уведомлений о состояниях и т. д. производится в графической среде. С созданными объектами могут работать различные «Кодовые Эксперты» (Code Experts), которые используются для генерации базового и расширенного кодов на языке Prolog, необходимых для обеспечения их функционирования.

Последняя версия программы Visual Prolog 7.4 вышла в ноябре 2012 года.

8.1 Запуск среды визуальной разработки Visual Prolog

Программа установки Visual Prolog инсталлирует группу программ с пиктограммой, которая обычно используется для запуска среды визуальной разработки Visual Prolog. Впрочем, существует множество других вариантов запуска приложения в Windows, например, запуск исполняемого файла среды визуальной разработки VIP.EXE из папки BIN\WIN\32 (32-битная версия Windows), находящейся в основном каталоге Visual Prolog.

Если во время закрытия среды визуальной разработки был открыт проект (PRJ или VPR файл), то в следующий раз при запуске среды визуальной разработки этот проект откроется автоматически.

Если в процессе инсталляции Visual Prolog вы выбрали флажок **Associate 32-bit VDE with Project File Extensions PRJ & VPR**, то для открытия проекта достаточно дважды щелкнуть на файле с расширением PRJ или VPR. Среда визуальной разработки запускается и загружает выбранный проект.

Для запуска большинства примеров из данного руководства необходимо использовать Test Goal утилиту среды визуальной разработки. Эта утилита может быть активизирована при помощи команды **Project | Test Goal** или комбинации горячих клавиш <Ctrl>+<G>.

Для корректного выполнения примеров с утилитой Test Goal среда визуальной разработки использует специальные настройки загружаемых проектов. Мы рекомендуем вам создать и всегда использовать следующий специальный TestGoal проект.

Создание TestGoal-проекта для выполнения примеров

При использовании утилиты Test Goal для выполнения примеров требуется определить некоторые непредопределенные опции компилятора Visual Prolog. Для этого нужно выполнить следующие действия:

1 Запустить среду визуальной разработки Visual Prolog.

При первом запуске VDE (среды визуальной разработки) проект не будет загружен, и вы увидите окно, показанное на рис. 8.1. Также вас проинформируют, что по умолчанию создан инициализационный файл для Visual Prolog VDE.

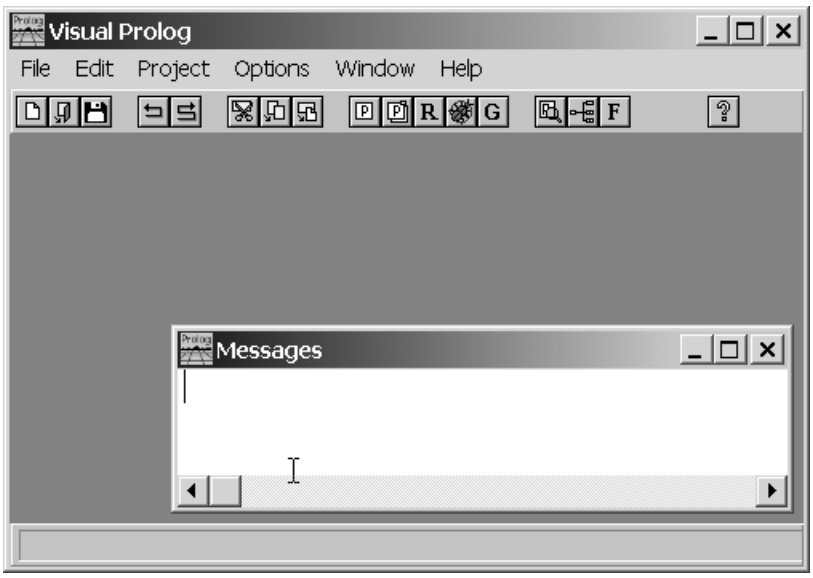


Рис. 8.1 Запуск среды визуальной разработки

2 Создайте новый проект.

Выберите команду **Project | New Project**, активизируется диалоговое окно **Application Expert**.

3 Определите базовый каталог и имя проекта.

Примеры из данного руководства следует устанавливать в подкаталог \DOC\Examples корневого каталога Visual Prolog. Допустим, что при установке Visual Prolog вы выбрали C:\VIP в качестве корневого каталогаVisual Prolog**.** В этом случае мы рекомендуем вам задать имя в поле **Base Directory** следующим образом:

C:\VIP\DOC\Examples\TestGoal.

Это определение очень удобно для будущей загрузки исходных текстов примеров, представленных в данном руководстве.

Имя в поле **ProjectName** следует определить как «TestGoal».

Также установите флажок **Multiprogrammer Mode**и щелкните мышью внутри поля **Name of .PRJ File**. Вы увидите, что появится имя файла проекта TestGoal.PRJ (рис. 8.2).

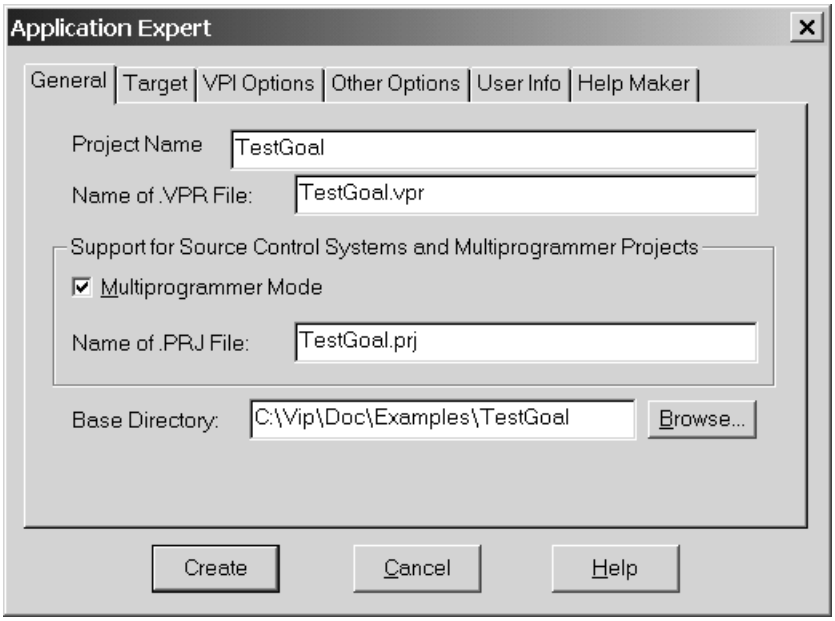


Рис. 8.2 Общие установки диалогового окна **Application Expert**

Определите цель проекта.

На вкладке **Target** (цель) мы рекомендуем выбрать параметры, отмеченные на рис. 8.3.

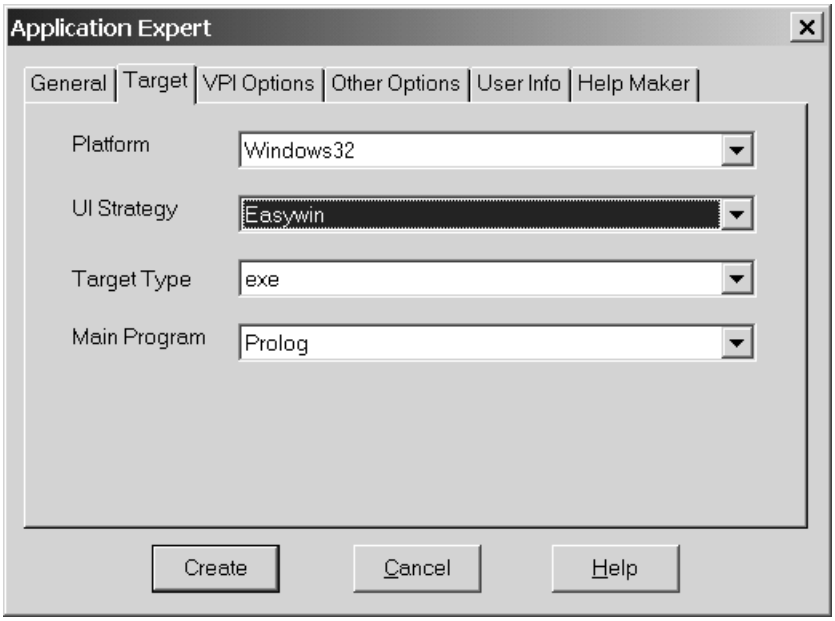


Рис. 8.3 Установки вкладки **Target** диалогового окна **ApplicationExpert**

Теперь нажмите кнопку **Create** для того, чтобы создать файлы проекта по умолчанию.

4 Установите требуемые опции компилятора для созданного TestGoalпроекта.

Для активизации диалогового окна **Compiler Options** выберите команду **Options | Project | Compiler Options**. Откройте вкладку **Warnings**.

Выполните следующие действия:

a) установите переключатель **Nondeterm**. Это нужно для того, чтобы компилятор Visual Prolog принимал по умолчанию, что все определенные пользователем предикаты недетерминированные (могут породить более одного решения);

b) снимите флажки **Non Quoted Symbols**, **Strong Type Conversion Check** и **Check Type of Predicates**. Это будет подавлять некоторые возможные предупреждения компилятора, которые неважны для понимания выполнения примеров данного руководства;

c) в результате этих действий диалоговое окно **Compiler Options** будет выглядеть, как на рис. 8.4.

Нажмите кнопку **OK,** чтобы сохранить установки опций компилятора.

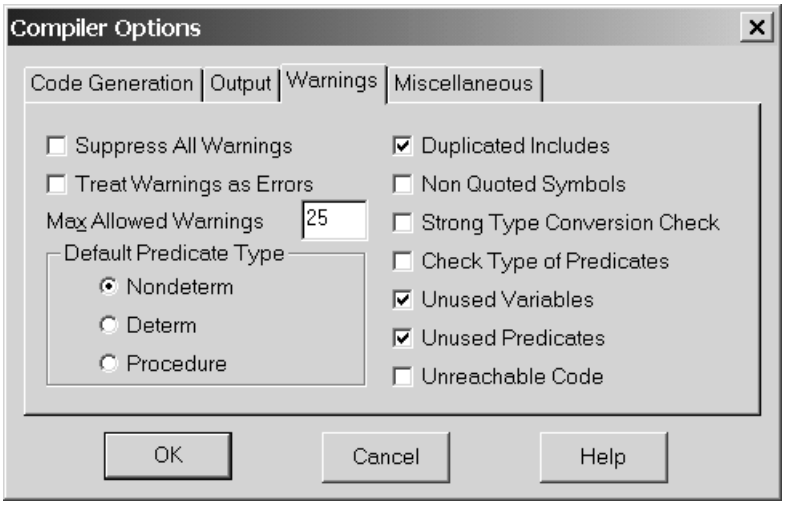


Рис. 8.4 Установки опций компилятора

Открытие окна редактора

Для создания нового окна редактирования вы можете использовать команду меню **File | New**. В результате появится новое окно редактирования с именем **Noname**.

Редактор среды визуальной разработки – стандартный текстовый редактор. Можно использовать клавиши управления курсором и мышь так же, как и в других редакторах. Редактор среды визуальной разработки поддерживает команды **Cut**, **Copy** и **Paste,Undo** и **Redo**, которые находятся в меню **Edit.** В меню **Edit** также показаны комбинации горячих клавиш для этих действий. Подробное описание редактора вы можете найти в системе помощи среды визуальной разработки (нажмите клавишу <F1> находясь в окне редактора).

Запуск и тестирование программы

Для проверки того, что ваша система настроена должным образом, вам следует напечатать следующий текст в окне:

GOAL write («Hello world»),nl.

В терминологии языка Пролог это называется *GOAL* (цель), и этого достаточно для программы, чтобы она могла быть выполнена. Для того, чтобы выполнить GOAL, вам следует активировать команду **Project | Test Goal** или нажать комбинацию клавиш <Ctrl>+<G>. Если ваша система установлена правильно, то экран монитора будет выглядеть, как на рис. 8.5.

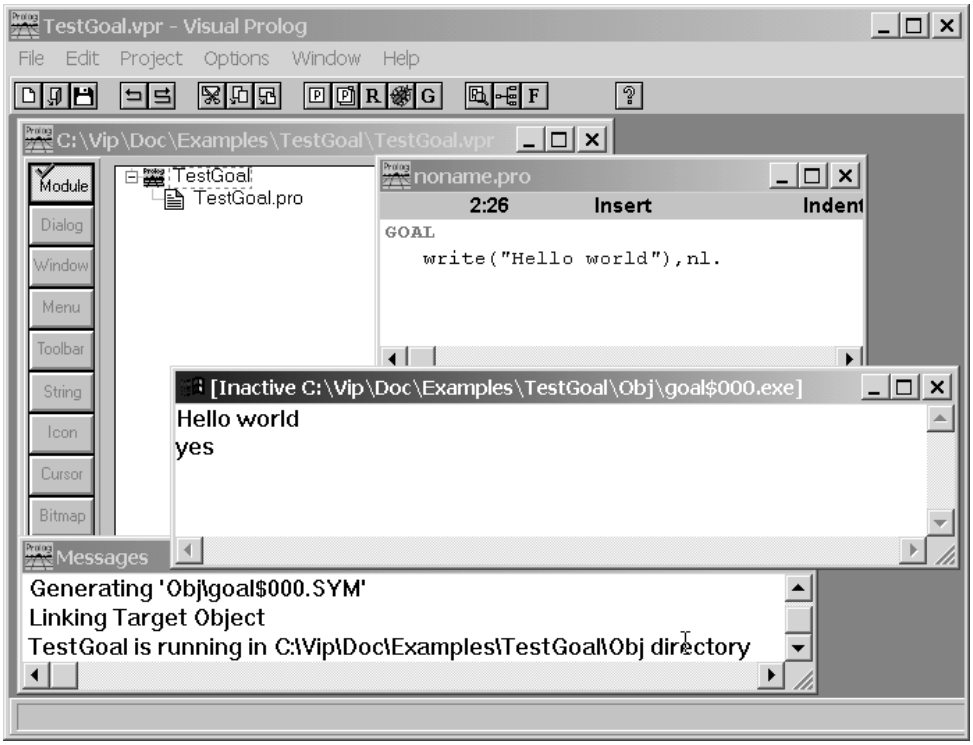


Рис. 8.5 Тестовая программа «Hello world»

Результат выполнения программы будет вверху в отдельном окне (на рисунке оно называется **[Inactive C:\Vip\Doc\Examples\TestGoal\Obj\ goal$000.exe]**), которое вы должны закрыть перед тем, как тестировать другую GOAL.

Тестирование примеров в Test Goal

Мы советуем вам попробовать сейчас открыть один из примеров в среде визуальной разработки и протестировать его, используя утилиту Test Goal. Для этого выполните следующие шаги:

1 Запустите среду визуальной разработки Visual Prolog.

2 Используйте команду меню **Project | Open Project** для открытия специального TestGoal проекта (см. ранее).

3 Используйте команду меню **File | Open** для открытия одного из файлов chCCeNN.pro.

4 Используйте команду меню **Project | Test Goal** (или нажмите комбинацию клавиш <Ctrl>+<G>**)** для тестирования загруженного примера.

Test Goalнайдет ВСЕ возможные решения GOAL и покажет значения ВСЕХ переменных, используемых в GOAL.

Комментарии к свойствам утилиты Test Goal

Утилита среды визуальной разработки Test Goal интерпретирует GOAL как специальную программу, которая компилируется, компонуется, генерируется исполняемый файл и Test Goal запускает его на выполнение. Test Goal внутренне расширяет заданный код GOAL, чтобы сгенерированная программа находила ВСЕ возможные решения и показывала значения ВСЕХ используемых переменных.

Утилита Test Goal компилирует этот код с использованием опций компилятора, заданных для открытого проекта (мы определили рекомендуемые опции компилятора для Test Goal проекта ранее). Заметим, что утилита Test Goal компилирует только тот код, который определен в активном окне редактора (код в других открытых редакторах или модулях проектов, если они есть, игнорируется). При компоновке исполняемого файла Test Goal использует стратегию EASYWIN (она описана в разделе книги, посвященном среде визуальной разработки).

Обратите внимание на то, что вы не можете определить какие-либо опции компоновки для Test Goal, т. к. игнорируются любые **Make Options,** заданные для открытого проекта. Поэтому Test Goal не может использовать никакие глобальные предикаты, определенные в других модулях. Заметим, что Test Goal имеет ограничение на количество переменных, которые могут быть использованы в GOAL. На данный момент их 12 для 32-битовой среды визуальной разработки, но это число может быть изменено без дополнительных уведомлений.

Тестирование примеров как автономных исполняемые программы

Большинство примеров в данном руководстве предназначено для тестирования с утилитой Test Goal, но некоторые примеры, такие как ch11e05.pro, предназначены для тестирования как *автономные исполняемые программы* (standaloneexecutables). Мы рекомендуем следующую процедуру для тестирования подобных примеров:

1 Запустите среду визуальной разработки Visual Prolog и откройте ранее созданный TestGoal проект (см. выше).

2 Откройте файл TestGoal.PRO для редактирования (двойной щелчок по пиктограмме TestGoalв окне проектов).

3 Код языка Пролог в файле TestGoal.PRO начинается (после начального комментария) с директивы include:

Include “TestGoal.INC”.

4 Закомментируйте эту директиву include и удалите остальной код Пролога. Заметим, что директива include может быть закомментирована для простых примеров из данного руководства, но она необходима в более сложных программах.

5 Включите файл с исходным кодом примера, пусть это будет ch11e05.pro. Заметим, что имя файла в директиве includeдолжно содержать правильный путь относительно корневого каталога проекта, поэтому мы рекомендует использовать команду редактора **Insert | FileName**:

Напечатайте:

include.

Выполните команду меню **Edit | Insert | FileName**; откроется диалоговое окно **Get & Insert FileName**. Найдите нужное имя файла (ch11e05.pro), выделите его и нажмите кнопку **Open**. После этого файл TestGoal.PRO должен содержать следующие строки:

```
% include "TestGoal.INC"   % Can be commented in simple examples
include "C:\\VIP_52\\DOC\\EXAMPLES\\ch04e05.pro"
```

6 Теперь можно компилировать исходный код примера ch11e05.pro, создавать исполняемый файл (в этом случае он будет называться TestGoal.EXE) и запускать его как автономную исполняемую программу. Все эти действия можно выполнить одной командой **Project | Run** или простым нажатием горячей клавиши <F9>.

Обработка ошибок

Если вы допустили некоторые ошибки в программе и пытаетесь скомпилировать ее, то среда визуальной разработки отобразит окно **Errors (Warnings)**, которое будет содержать список обнаруженных ошибок. Дважды щелкнув на одной из этих ошибок, вы попадете на место ошибки в исходном тексте. Вы можете нажать клавишу <F1> для вывода на экран интерактивной справочной системы Visual Prolog. Когда окно помощи откроется, щелкните по кнопке **Search** и наберите номер ошибки; на экране появится соответствующее окно помощи с более полной информацией об ошибке (рис. 8.6).

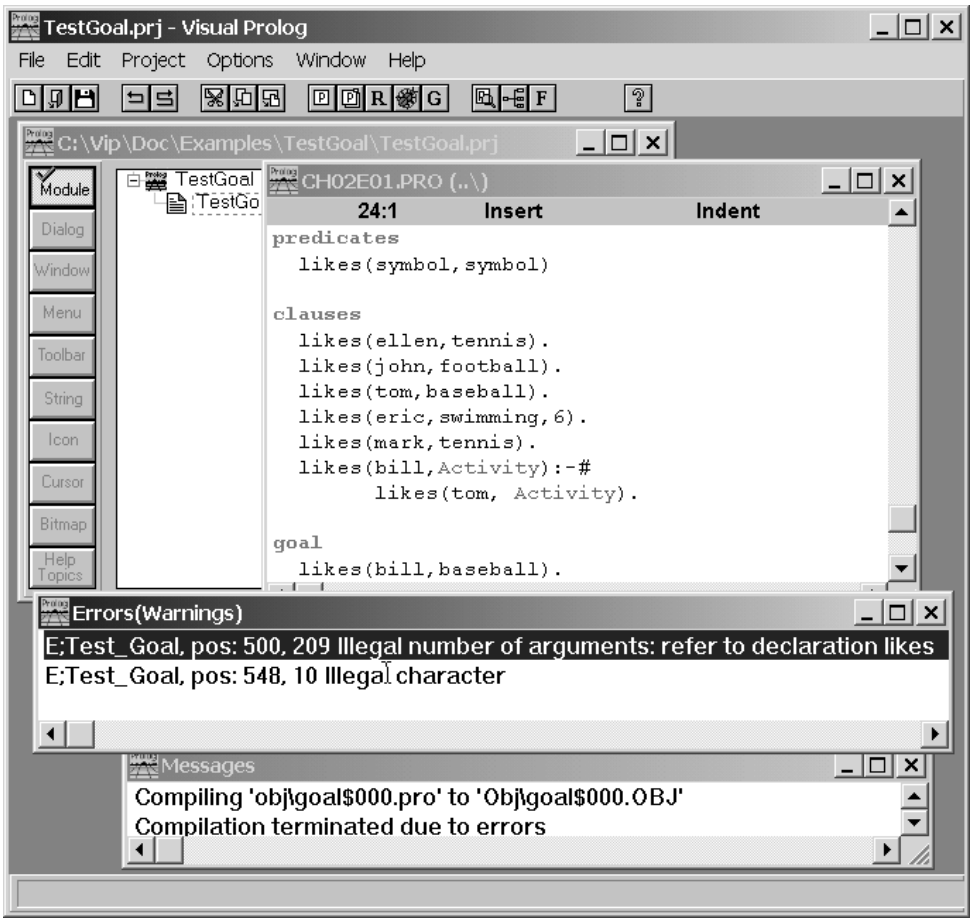


Рис. 8.6 Обработка ошибок

8.2 Программирование в логике

В Прологе (Prolog – PROgramming LOGic) вы достигаете решения задачи логическим выводом его из ранее известных положений. Обычно, программа на Прологе не является последовательностью действий – она представляет собой набор фактов с правилами, обеспечивающими получение заключений на основе этих фактов. Поэтому Пролог известен как декларативный язык.

Пролог базируется на предложениях Хорна, являющихся подмножеством формальной системы, называемой логикой предикатов. Не стоит пугаться этого термина. Логика предикатов – это простейший способ объяснить, как «работает» мышление. И она проще, чем арифметика, которой вы давно пользуетесь.

Пролог использует упрощенную версию синтаксиса логики предикатов, так как он является простым для понимания и очень близок к естественному языку. Пролог включает механизм вывода.

Механизм вывода включает сопоставление образцов. С помощью подбора ответов на запросы он извлекает хранящуюся (известную) информацию. Пролог пытается проверить истинность гипотезы (другими словами – ответить на вопрос), запрашивая для этого информацию, о которой уже известно, что она истинна. Прологовское знание о мире – это ограниченный набор фактов (и правил), заданных в программе.

Одной из важнейших особенностей Пролога является то, что в дополнение к логическому поиску ответов на поставленные вами вопросы он может иметь дело с альтернативами и находить все возможные решения, а не одно. Вместо обычной работы от начала программы до ее конца, Пролог может возвращаться назад и просматривать более одного «пути» при решении всех составляющих задачу частей.

Логика предикатов была разработана для наиболее простого преобразования базирующихся на логике мыслей в записываемую форму. Пролог использует преимущества ее синтаксиса для разработки программного языка, основанного на логике. В логике предикатов вы прежде всего исключаете из своих предложений все несущественные слова. Затем вы преобразуете эти предложения, ставя в них на первое место отношение, а после него – сгруппированные объекты. Затем объекты становятся аргументами , между которыми устанавливается это отношение.

В качестве примера в табл. 8.1 представлены предложения, преобразованные в соответствии с синтаксисом логики предикатов.

Таблица 8.1. **Синтаксис логики предикатов**

Предложения на естественном языке	Синтаксис логики предикатов
Машина красивая Роза красная	Fun (car) Red (rose)
Билл любит машину, если машина красивая	Likes (bill, Car) if fun (Car)

Предложения: факты и правила

Программист на Прологе описывает *объекты* (objects *отношения* (relations), а затем описывает *правила* (rules), при которых эти отношения являются истинными. Например, предложение

Билл любит собак. (Bill likes dogs.)

устанавливает отношение между объектами Bill и dogs (Билл и собаки); этим отношением является likes (любит). Ниже представлено предложение, описывающее, когда предложение «Билл любит собак» является истинным:

Билл любит собак если собаки хорошие. (Bill likes dogs ***if*** *the dogs are nice.).*

Факты: то что известно

В Прологе отношение между объектами называется фактом (fact). В естественном языке отношение устанавливается в предложении. В логике предикатов, используемой Прологом, отношение соответствует простой фразе (факту), состоящей из имени отношения и объекта или объектов, заключенных в круглые скобки. Как и предложение, факт завершается точкой (.).

Ниже представлено несколько предложений на естественном языке с отношением «любит» (likes):

Билл любит Синди. (Bill likes Cindy).
Синди любит Билла. (Cindy likes Bill).
Билл любит собак. (Bill likes dogs).

А теперь перепишем эти же факты, используя синтаксис Пролога:

likes (bill, cindy)
likes (cindy, bill)
likes (bill, dogs).

Факты, помимо отношений, могут выражать и свойства. Так, например, предложения естественного языка «Kermit is green» (Кермит зеленый) и «Caitlin is girl» (Кейтлин – девочка) на Прологе, выражая те же свойства, выглядят следующим образом:

green (kermit)
girl (caitlin).

Правила: то, что вы можете получить из заданных фактов

Правила позволяют вам вывести один факт из других фактов.

Другими словами, можно сказать, что правило– это заключение, для которого известно, что оно истинно, если одно или несколько других найденных заключений или фактов являются истинными. Ниже представлены правила, соответствующие связи «любить» (likes):

Синди любит все, что любит Билл. (Cindy likes everything that Bill likes).
Кейтлин любит все зеленое. (Caitlin likes everything that is green).

Используя эти правила, вы можете из предыдущих фактов найти некоторые вещи, которые любят Синди и Кейтлин:

Синди любит Синди. (Cindy likes Cindy).
Кейтлин любит Кермит. (Caitlin likes Kermit).

Чтобы перевести эти правила на Пролог, вам нужно немного изменить синтаксис, подобно этому:

likes (cindy, Something):- likes (bill, Something)
likes (caitlin, Something):- green (Something).

Символ :- имеет смысл «если», и служит для разделения двух частей правила: заголовка и тела.

Вы можете рассматривать правило и как процедуру. Другими словами, эти правила

likes (cindy, Something): - likes (bill, Something)

likes (caitlin, Something): - green (Something)

также означают «Чтобы доказать, что Синди любит что-то, докажите, что Билл любит это» и «Чтобы доказать, что Кэтлин любит что-то, докажите, что это что-то зеленое». С такой «процедурной» точки зрения правила могут «попросить» Пролог выполнить другие действия, отличные от доказательств фактов – такие как напечать что-нибудь или создать файл.

Запросы

Однократно дав языку Пролог несколько фактов, мы можем продолжать задавать вопросы, касающиеся этих фактов. Это называется *запросом* (query)системы языка Пролог. Мы можем задавать Прологу такие же вопросы, которые мы могли бы задать вам об этих отношениях. Основываясь на известных, заданных ранее фактах и правилах, вы можете ответить на вопросы об этих отношениях, в точности как это может сделать Пролог.

На естественном языке мы спрашиваем вас:

Does Bill like Cindy?(Билл любит Синди?)

По правилам Пролога мы спрашиваем Пролог:

likes (bill, cindy).

Получив такой запрос, Prolog мог бы ответить

yes,

потому что Prolog имеет факт, подтверждающий, что это так.

Немного усложнив вопрос, мы могли бы спросить вас на естественном языке:

What does Bill like?(Что любит Билл?)

По правилам Пролога мы спрашиваем Пролог:

likes (bill, What).

Заметим, что синтаксис Пролога не изменяется, когда вы задаете вопрос: этот запрос очень похож на факт. Впрочем, важно отметить, что второй объект – *What* – начинается с большой буквы, тогда как первый объект – bill – нет. Это происходит потому, что bill – фиксированный, постоянный объект – известная величина, а What – переменная. Переменные всегда начинаются с заглавной буквы или символа подчеркивания.

Пролог всегда ищет ответ на запрос, начиная с первого факта, и перебирает все факты, пока они не закончатся.

Получив запрос о том, что Билл любит, Пролог ответит:

What=cindy

What=dogs

2 Solutions.

Так как ему известно, что

likes (bill, cindy).

и

likes (bill, dogs).

Надеемся, что вы пришли к такому же выводу.

Если бы мы спросили вас (и Пролог):

What does Cindy like? (Что любит Синди?)

likes (cindy, What).

Prolog ответил бы

What = bill

What = cindy

What = dogs

3 solutions.

Это происходит потому, что Пролог знает, что Синди любит Билла, и что Синди любит то же, что и Билл, и что Билл любит Синди и собак.

Мы могли бы задать Прологу и другие вопросы, которые можно задать человеку, однако вопросы типа «Какую девушку любит Билл?» не дадут решения, так как Прологу в данном случае не известны факты о девушке, и он не может вывести заключение, основанное на неизвестных данных. В этом примере мы не дали Прологу какого-нибудь отношения или свойства, чтобы определить, являются ли какие-либо объекты девушками.

Размещение вместе фактов, правил и запросов

Полагаем, у вас есть следующие факты и правила:

Быстрая машина – забавная. (A fast car is fun).

Большая машина – красивая. (A big car is nice).

Маленькая машина – практичная. (A little car is practical).

Биллу нравится машина, если она забавная. (Bill likes a car if the car is fun).

Читая эти факты, вы можете сделать вывод, что Биллу нравится быстрый автомобиль. В большинстве случаев Пролог придет к подобному решению. Если бы не было фактов о быстрых автомобилях, вы не смогли бы логически вывести, какие автомобили нравятся Биллу. Вы можете делать предположения о том, какой тип машин может быть крепким, но Пролог знает только то, что вы ему скажете. Пролог не строит предположений.

Вот пример, демонстрирующий, как Пролог использует правила для ответа на запросы. Посмотрите на факты и правила в этой части программы ch02e01.pro:

likes (ellen, tennis).

likes (john, football).

likes (tom, baseball). likes(eric, swimming).

likes (mark, tennis).

likes (bill, Activity):- likes (tom, Activity).

Последняя строка в программе ch02e01.proявляется правилом:

likes (bill, Activity):- likes (tom, Activity).

Это правило соответствует предложению естественного языка

Биллу нравится занятие, если Тому нравится это занятие. (Bill likes an activity if Tom likes that activity).

В этом правиле заголовок – это likes (bill, Activity), а тело – likes (tom, Activity). Заметим, что в этом примере нет фактов о том, что Билл любит бейсбол. Чтобы выяснить, любит ли Билл бейсбол, можно дать Прологу такой запрос:

likes (bill, baseball).

Пытаясь отыскать решение по этому запросу, Пролог будет использовать правило:

likes (bill, Activity):- likes(tom, Activity).

Загрузите программу ch02e01.pro в среду визуальной разработки Visual Prolog и запустите ее утилитой Test Goal (см. подраздел «Тестирование примеров в Test Goal» данного руководства в главе 8):

```
predicates
likes (symbol,symbol)
clauses
likes (ellen, tennis).
likes (john, football).
likes (tom, baseball).
likes (eric, swimming).
likes (mark, tennis).
likes (bill, Activity):-
likes (tom, Activity).
goal
likes (bill, baseball).
```

Test Goal ответит в окне приложения

yes.

Система использовала комбинированное правило:

likes (bill, Activity):- likes (tom, Activity)

с фактом

likes (tom, baseball)

для решения, что

likes (bill, baseball).

Попробуйте также этот запрос в GOAL разделе:

likes (bill, tennis).

Test Goal ответит

no.

Visual Prolog ответит no на последний запрос «Does Bill like tennis?» (любит ли Билл теннис), так как:

- нет фактов, которые говорят, что Билл любит теннис;
- отношение Билла к теннису не может быть логически выведено с использованием данного правила и имеющихся в распоряжении фактов.

Конечно, вполне возможно, что Билл обожает теннис в реальной жизни, но ответ Visual Prolog основан только на фактах и правилах, которые вы дали ему в тексте программы.

Переменные: общее представление

В Прологе *переменные* позволяют вам записывать общие факты и правила и задавать общие вопросы. В естественном языке вы пользуетесь переменными в предложениях постоянно. Обычное предложение на английском языке могло бы быть таким

Bill likes the same thing as Kim.(Билл любит то же, что и Ким).

Как мы говорили ранее в этой главе, при задании переменной в Прологе первый символ имени должен быть заглавной буквой или символом подчеркивания.

Например, в следующей строке Thing – это переменная

likes (bill, Thing):- likes (kim, Thing).

В предшествующем обсуждении правил вы видели эту строку:

likes (cindy, Something):- likes (bill, Something).

Объект Something начинается с заглавной буквы, так как это переменная; он определяет что-то, что Билл любит. С таким же успехом этот объект мог бы называться X или Zorro.

Объекты bill и cindy начинаются со строчной буквы, так как они не являются переменными – это *идентификаторы*, имеющие постоянное значение. Visual Prolog может обрабатывать произвольные текстовые строки подобно тому, как мы оперировали символами, упомянутыми выше, если текст заключен в двойные кавычки. Следовательно, вместо bill вы могли бы также успешно написать «Bill».

Краткий обзор

1 Программы на языке Пролог состоят из двух типов фраз, также называемых предложениями: фактов и правил.

Факты – это отношения или свойства, о которых Вы, программист, знаете, что они имеют значение «истина».

Правила – это связанные отношения; они позволяют Прологу логически выводить одну порцию инфомации из другой. Правило принимает значение «истина», если доказано, что заданный набор условий является истинным.

2 В Прологе все правила имеют две части: заголовок и тело, разделенные специальным знаком :-.

Заголовок – это факт, который был бы истинным, если бы были истинными несколько условий. Это называется выводом или зависимым отношением.

Тело – это ряд условий, которые должны быть истинными, чтобы Пролог мог доказать, что заголовок правила истинен.

3 Как вы уже, наверное, заметили, факты и правила – практически одно и то же, кроме того, что факты не имеют явного тела. Факты ведут себя так, как если бы они имели тело, которое всегда истинно.

Однажды дав Прологу ряд фактов и/или правил, вы можете задавать по ним вопросы; это называется запросом системы Пролог. Пролог всегда ищет решение, начиная с первого факта и/или правила, и просматривает весь список фактов и/или правил до конца.

Механизм логического вывода Пролога берет условия из правила (тело правила) и просматривает список известных фактов и правил, пытаясь удовлетворить условиям. Если все условия истинны, то зависимое отношение (заголовок правила) считается истинным. Если все условия не могут быть согласованы с известными фактами, то правило ничего не выводит.

Упражнения

Переведите следующие предложения на естественный язык так, чтобы они были понятны читателю (помните, что для компьютера эти факты являются просто порциями информации, которые могут быть использованы для согласования ответов с вопросами):

1 likes (jeff, painting).
2 male (john).
3 building ("Empire State Building", new_york).
4 person (roslin, jeanie, "1429 East Sutter St.", "Scotts Valley", "CA", 95066).

От естественного языка к программам на Прологе

В первой части этого раздела мы говорили о фактах и правилах, отношениях, основных конструкциях и запросах. Все эти термины являются частью логики и естественного языка. Сейчас мы будем обсуждать теже понятия, но используя больше терминов Пролога, таких как предложения, предикаты, переменные и цели.

Предложения (факты и правила)

По сути, есть только два типа фраз, составляющих язык Пролог: фраза может быть либо фактом, либо правилом. Эти фразы в Прологе известны под термином *предложения* (clause). Сердце программ на Прологе состоит из предложений.

Подробнее о фактах

Факт представляет либо свойство объекта, либо отношение между объектами. Факт самодостаточен. Прологу не требуется дополнительных сведений для подтверждения факта, и факт может быть использован как основа для логического вывода.

Подробнее о правилах

В Прологе, как и в обычной жизни, можно судить о достоверности чего-либо, логически выведя это из других фактов. Правило – это конструкция Пролога, которая описывает, что можно логически вывести из других данных. Правило – это свойство или отношение, которое достоверно, когда известено, что ряд других отношений достоверен. Синтаксически эти отношения разделены запятыми, как показано ниже в примере 1.

Примеры правил

1 *Первый пример* показывает правило, которое может быть использовано для того, чтобы сделать вывод, подходит ли некоторое блюдо из меню для Дианы.

Диана – вегитарианка и ест только то, что говорит ей ее доктор. (Diane is a vegetarian and eats only what her doctor tells her to eat).

Зная меню и предыдущее правило, вы можете сделать вывод о том, выберет ли Диана данное блюдо. Чтобы выполнить это, вы должны проверить, соответствует ли блюдо заданному ограничению:

является ли Food_on_menu овощем?

находится ли Food_on_menu в списке доктора?

Заключение: если оба ответа положительны, Диана может заказать Food_on_menu.

В Прологе подобное отношение должно быть выражено правилом, так как вывод основан на фактах.

Вот один вариант записи правила:

diane_can_eat (Food_on_menu):-
vegetable (Food_on_menu),
on_doctor_list (Food_on_menu).

Обратите внимание, что после vegetable (Food_on_menu) стоит запятая. Запятая обозначает конъюнкцию нескольких целей и читается как «и»; оба vegetable (Food_on_menu) **и** on_doctor_list (Food_on_menu) – должны быть истинны для истинности diane_can_eat (Food_on_menu).

2 *Второй пример*. Предположим что вам хотелось бы записать факт, который истинен, если Person 1 является родителем Person 2. Это достаточно просто, нужный факт выглядит так:

parent (paul, samantha).

Этот факт обозначает, что Пол – родитель Саманты. Но, предположим, что в базе данных фактов Visual Prolog есть факты, формулирующие отношения отцовства. Например, «Пол – отец Саманты»:

father (paul, samantha).

Пусть у вас также есть факты, формулирующие отношение материнства, например, «Джулия – мать Саманты»:

mother (julie, samantha).

Если у вас уже есть несколько фактов, формулирующих отношения отцовства / материнства, то не стоит тратить время на запись факта родства в базу данных фактов для каждого отношения родства.

Так как вы знаете, что Person 1 – родитель Person 2, если Person 1 – отец Person 2 или Person 1 – мать Person 2, то почему бы не записать правило, объединяющее эти ограничения? После формулирования этих условий на естественном языке, будет достаточно просто записать их в правило Пролога:

parent (Person1, Person2):- father (Person1, Person2).
parent (Person1, Person2):- mother (Person1, Person2).

Эти правила Пролога говорят, что

Person 1 является родителем Person 2, если Person 1 является отцом Person 2.

Person 1 является родителем Person 2, если Person 1 является матерью Person 2.

3 Вот *третий пример*.

Человек может купить машину, если машина ему нравится (likes), и если машина продается (for sale).

Это отношение может быть переведено на язык Пролог следующим правилом:

```
can_buy (Name, Model):-
person (Name),
car (Model),
likes (Name, Model),
for_sale (Model).
```

Это правило выражает следующие отношения:

Name может купить (can_buy) Model если
Name является человеком (person) и
Model является машиной (car) и
Name нравится (likes) Model и
Model продается (for_sale).

Это правило будет истинным, если истинны все четыре условия в теле правила.

4 Далее в листинге 1 представлена программа, которая ищет решение проблемы покупки автомобиля (протестируйте ее).

Листинг 1. Программа ch02e02.pro

```
predicates
can_buy (symbol, symbol)
person (symbol)
car (symbol)
likes (symbol, symbol)
for_sale (symbol)
clauses
can_buy (X,Y):-
person (X),
car (Y),
likes (X,Y),
for_sale (Y).
person (kelly).
person (judy).
person (ellen).
person (mark).
car (lemon).
car (hot_rod).
likes (kelly, hot_rod).
likes (judy, pizza).
likes (ellen, tennis).
likes (mark, tennis).
for_sale (pizza).
for_sale (lemon).
for_sale (hot_rod).
```

Что могут купить Джуди и Келли? Кто может купить hot_rod? Испытайте следующие цели в разделе GOAL:

```
can_buy (Who, What).
can_buy (judy, What).
can_buy (kelly, What).
can_buy (Who, hot_rod).
```

Эксперимент!

Добавьте другие факты и, может, даже 1-2 правила к этой программе. Протестируйте новую программу теми запросами, что вы составили. Оправдали результаты ваши ожидания или нет?

Упражнения

1 Напишите предложения на естественном языке, соответствующие следующим правилам Visual Prolog:

eats (Who, What):- food (What), likes (Who, What).
pass_class (Who):- did_homework (Who), good_attendance (Who).
owns (Who, What):- bought (Who, What).

2 По данным предложениям составьте правила Visual Prolog:

человек голоден, если его желудок пуст;
каждому нравится работа, если она веселая и хорошо оплачиваемая;
обладает автомобилем тот, кто купил его, платит за него и управляет им.

Предикаты (отношения)

Символическое имя отношения называется именем *предиката*. Аргументы – это объекты, которые связываются этим отношением; в факте likes (bill, cindy) отношение likes – это предикат, а объекты bill и cindy – аргументы.

Вот несколько примеров предикатов с различным числом аргументов:

pred (integer, symbol)
person (last, first, gender)
run ()
insert_mode ()
birthday (first Name, last Name, date).

В вышеприведенном примере показано, что предикаты вовсе могут не иметь аргументов, но использование таких предикатов ограничено. Чтобы выяснить имя Mr. Rosemont, можно применить запрос person (rosemont, Name, male). Но что делать с запросом без аргументов run? Вы можете выяснить, есть ли в программе предложение run, или, если run – это заголовок правила, то вы можете вычислить данное правило. В некоторых случаях это может быть полезно – например, если бы вы захотели создать программу, работающую по-разному в зависимости от того, имеется ли предложение insert_mode.

Переменные (общие предложения)

В простом запросе, чтобы найти, кто любит теннис, можно использовать переменные. Например:

likes (X, tennis).

В этом запросе буква X используется как переменная для нахождения неизвестного человека. Имена переменных в Visual Prolog должны начинаться с заглавной буквы (или символа подчеркивания), после которой может стоять любое количество букв (заглавных или строчных), цифр или символов подчеркивания.

Например, здесь приведены правильные имена переменных:

My_first_correct_variable_name
Sales_10_11_86

в то время как следующие три – неправильные:

1stattempt
second_attempt
"disaster".

Осмысленный выбор имен переменных делает программу более удобной для чтения. Например:

likes (Person, tennis).

лучше, чем

likes (X, tennis).

потому что Person имеет больше смысла, чем X. Теперь испытайте цель

GOAL likes (Person, tennis).

Visual Prolog ответит:

Person=ellen
Person=mark
2 Solutions.

Так как цель может получить два решения, а именно, сопоставляя переменную Person последовательно со значениями ellen и mark.

В имени переменной, исключая первый символ, Visual Prolog позволяет использовать заглавные и строчные буквы. Один из способов сделать программу более удобной для чтения – это использовать в названии переменной буквы разного регистра, как здесь

IncomeAndExpenditureAccount.

Инициализация переменных

Вы уже могли заметить, что Пролог не имеет оператора присваивания. Это важное отличие Пролога от других языков программирования.

Замечание. Переменные в Прологе инициализируются при сопоставлении с константами в фактах или правилах.

До инициализации переменная свободна; после присвоения ей значения она становится связанной. Переменная остается связанной только то время, которое необходимо для получения решения по запросу; затем Пролог освобождает ее и ищет другое решение.

Замечание. Нельзя сохранить информацию, присвоив значение переменной.Переменные используются как часть процесса поиска решения, а не как хранилище информации.

Посмотрите на следующий пример, в котором программа ch02e03.pro используется для иллюстрации того, как и когда переменные получают свои значения.

Листинг 2. Программа ch02e03.pro

```
predicates
likes (symbol, symbol)
clauses
likes (ellen, reading).
likes (john, computers).
likes (john, badminton).
likes (leonard, badminton).
likes (eric, swimming).
likes (eric, reading).
```

Рассмотрите запрос: есть ли человек, который любит и чтение и плавание?

likes (Person, reading), likes (Person, swimming).

Пролог будет решать обе части запроса посредством поиска предложений программы с начала и до конца. В первой части запроса

likes (Person, reading)

переменная Person свободна; ее значение неизвестно перед тем, как Пролог пытается найти решение. С другой стороны, второй аргумент, reading, известен. Пролог ищет факт, который соответствует первой части запроса. Первый факт в программе

likes (ellen, reading)

удовлетворяет первой части запроса (reading в факте соответствует reading в запросе), значит Пролог связывает свободную переменную Person со значением ellen, соответствующим значению в факте. В то же время Пролог помещает указатель в список фактов, показывающий, как далеко продвинулась процедура поиска.

Далее, для полного разрешения запроса (поиска человека, которому нравится и чтение, и плавание) должна быть решена вторая часть запроса. Так как Person сейчас связана со значением ellen, Пролог должен искать факт

likes (ellen, swimming).

Пролог ищет этот факт от начала программы, но совпадений нет (потому что в программе нет такого факта). Вторая часть запроса ложна, если Person имеет значение ellen.

Теперь Пролог освобождает переменную Person и пытается найти другое решение первой части запроса. Поиск другого факта, удовлетворяющего первой части запроса, начинается с указателя в списке фактов (такое возвращение к отмеченной позиции называется *поиск с возвратом*).

Пролог ищет следующего человека, кто любит чтение и находит факт likes (eric, reading). Переменная Person сейчас связана со значением eric, и Пролог пытается вновь найти соответствие со второй частью запроса посредством поиска в программе факта

likes (eric, swimming).

Пролог находит совпадение (последнее предложение в программе), и запрос полностью удовлетворяется. Пролог (Test Goal) возвращает

Person = eric
1 Solution.

Анонимные переменные

Анонимные переменные позволяют вам привести в порядок ваши программы. Если вам нужна только определенная информация запроса, вы можете использовать анонимные переменные для игнорирования ненужных вам значений. В Прологе анонимные переменные обозначаются символом подчеркивания "_".

Следующий «семейный» пример демонстрирует использование анонимных переменных. Загрузите программу ch02e04.pro в TestGoal проект.

Листинг 3. Программа ch02e04.pro

```
predicates
male (symbol)
female (symbol)
parent (symbol, symbol)
clauses
male (bill).
male (joe).
female (sue).
female (tammy).
parent (bill, joe).
parent (sue, joe).
parent (joe, tammy).
```

Замечание. Анонимная переменная может быть использована на месте любой другой переменной. Разница в том, что анонимной переменной никогда не будет присвоено значение.

Например, в следующем запросе вам понадобится узнать, какие люди являются родителями, но вам неинтересно, кто их дети. Пролог знает, что каждый раз, когда вы используете символ подчеркивания в запросе, вам не нужна информация о значении, представленном на месте переменной.

goal
parent (Parent, _).

Получив такойзапрос, Пролог (Test Goal**)** ответчает

Parent=bill
Parent=sue
Parent=joe
3 Solutions.

В этом случае Пролог находит и выдает трех родителей, но он не выдает значения, связанные со вторым аргументом в предложении parent**.**

Анонимные переменные также можно использовать в фактах. Следующие факты Пролога

owns (_, shoes).
eats (_).

могли быть использованы для выражения утверждений на естественном языке:

У каждого есть туфли. (Every one owns shoes).
Каждый ест. (Every one eats).

Замечание. Анонимные переменные сопоставляются с чем угодно.

Цели (запросы)

До сих пор мы, говоря о вопросах, задаваемых Прологу, употребляли слово *запрос*. Далее мы будем использовать более общее слово *цель*. Трактование запросов как целей осмысленно: когда вы даете Прологу запрос, вы в действтельности даете ему цель для выполнения. («Найди ответ на вопрос, если он существует: …»).

Цели могут быть или простыми, как эти две:

likes (ellen, swimming).
likes (bill, What).

или более сложными. В подразделе «Переменные» этого раздела вы видели цель из двух частей:

likes (Person, reading), likes (Person, swimming).

Цель, состоящая из двух и более частей, называется *сложной целью*, а каждая часть сложной цели называется ***подцелью***.

Часто вам нужно узнать общее решение двух целей. Например, в предыдущем примере о родителях вы могли бы поинтересоваться, которые из родителей являются родителями мужского пола. Вы можете заставить

Пролог искать решение такому запросу, задав сложную цель. Загрузите программу ch02e04.pro и задайте следующую сложную цель:

Goalparent(Person, _), male(Person).

Сначала Пролог попытается решить подцель

parent(Person, _)

путем поиска соответствующего предложения и последующего связывания переменной Personco значанием, возвращенным parent(Person – это родитель). Значение, возвращаемое parent**,** далее предоставляется второй подцели в качестве значения, с которым будет продолжен поиск (Является ли Person – теперь связанная переменная – мужского пола?):

male (Person).

Если вы задали цель корректно, Пролог (Test Goal) ответит

Person=bill
Person=joe
2 Solutions .

Составные цели: конъюнкция и дизъюнкция

Как вы уже видели, вы можете использовать составные цели для поиска решения, в котороми обе подцели A и B истинны (конъюнкция), разделяя подцели запятой. Но это не все. Вы также можете искать решения в том случае, если истинна либо подцель A, либо подцель B (дизъюнкция), разделяя подцели точкой с запятой. Ниже представлен пример программы ch02e05.pro, иллюстрирующей эту идею:

Листинг 4. Программа ch02e05.pro

```
predicates
car (symbol,long,integer,symbol,long)
truck (symbol,long,integer,symbol,long)
vehicle (symbol,long,integer,symbol,long)
clauses
car (chrysler,130000,3,red,12000).
car (ford,90000,4,gray,25000).
car (datsun,8000,1,red,30000).
truck (ford,80000,6,blue,8000).
truck (datsun,50000,5,orange,20000).
truck (toyota,25000,2,black,25000).
vehicle (Make,Odometer,Age,Color,Price):-
car (Make,Odometer,Age,Color,Price);
truck (Make,Odometer,Age,Color,Price).
```

Загрузите эту программу в проект TestGoal, а затем добавьте цель:

```
goal
car (Make, Odometer, Years_on_road, Body, 25 000).
```

Данная цель попытается найти описанную в предложениях машину (car), которая стоит ровно $25.000. Пролог (Test Goal) ответит:

Make=ford, Odometer=90000, Years_on_road=4, Body=gray
1 Solution.

Однако, данная цель несколько неестественна, т. к. скорее всего вы захотите задать вопрос типа:

Есть ли в списке машина, стоящая меньше, чем $25.000? (Is there a car listed that costs less than $25,000?)

Для поиска такого решения вы можете задать Visual Prolog следующую составную цель:

car (Make, Odometer, Years_on_road, Body, Cost), % подцель A ***и***
Cost < 25000. % подцель B

Это и является конъюнкцией. Для разрешения этой составной цели Пролог будет пытаться по очереди решать подцели. Вначале он попытается решить

car (Make, Odometer, Years_on_road, Body, Cost).

а затем

Cost< 25000.

с переменной Cost, имеющей идентичное значение в обеих подцелях. Теперь попытайтесь проделать все это с TestGoal.

Замечание. Подцель Cost < 25000 соответствует отношению «меньше чем», которое встроено в систему Visual Prolog. Отношение «меньше чем» ничем не отличается от любого другого отношения, использующего два числовых объекта, но гораздо естественнее использовать для его обозначения символ «<», помещая его между двумя объектами.

Сейчас мы попытаемся посмотреть, является ли истинным следующее выражение на естественном языке:

Есть ли в списке автомобиль, стоимостью меньше $25.000, или грузовик стоимостью меньше $ 20.000?

При задании следующей составной цели Пролог выполнит поиск требуемого решения:

car (Make,Odometer,Years_on_road,Body,Cost),
Cost < 25000 % подцель A ***или;***
truck(Make,Odometer,Years_on_road,Body,Cost),
Cost < 20000. % подцель B.

Этот тип составной цели является дизъюнкцией. Данная цель установила две альтернативные подцели также, как если бы это были два предложения одного правила. Пролог будет искать все решения, удовлетворяющие обеим подцелям.

При разрешении такой составной цели Пролог вначале попытается решить первую подцель («найти автомобиль...»), состоящую из следующих подцелей:

car (Make, Odometer, Years_on_road, Body, Cost)

и

Cost< 25000.

Если автомобиль найдется – цель истинна; если нет – Пролог попытается разрешить вторую составную цель («найти грузовик ...»), состоящую из подцелей:

truck(Make, Odometer, Years_on_road, Body, Cost),

и

Cost< 20000.

Комментарии

Хорошим стилем программирования является включение в вашу программу комментариев, объясняющих все то, что может быть непонятно кому-то другому (или вам самому спустя полгода). Это делает вашу программу проще для понимания и вами, и всеми другими. Если вы подбираете подходящие имена для переменных, предикатов и доменов, то вам понадобится меньше комментариев, т. к. ваша программа будет объяснять сама себя.

Многострочные комментарии должны начинаться с символов «/*» (косая черта, звездочка) и завершаться символами «*/» (звездочка, косая черта). Для установки однострочных комментариев вы можете либо использовать эти же самые символы, либо начинать комментарий символом процента (%).

```
/* Это пример комментария */
% Это тоже комментарий
/*****************************************/
/* и эти три строчки — тоже       */
/*****************************************/
/*Вы также можете поместить комментарий Visual Prolog /*внутри
комментария */ как здесь */
```

В Visual Prolog вы также можете использовать комментарий после каждого субдомена в объявлении доменов:

```
domains
articles = book (string Title, string Author); horse (string Name)
```

и в объявлениях предикатов:

```
predicates
conv (string Uppercase, string Lowercase).
```

Слова Title, Author, Name, Uppercase и Lowercase будут проигнорированы компилятором, но сделают программу гораздо более читабельной.

Что такое сопоставление

В предыдущих подразделах данного раздела вы познакомились с тем, как Пролог «сопоставляет вопросы и ответы», «ищет сопоставление», «сопоставляет условия с фактами», «сопоставляет переменные с константами» и так далее. В данном разделе объясняется, что же понимается под термином *сопоставление* (matching).

В Прологе имеется несколько примеров сопоставления одной вещи с другой. Ясно, что *идентичные структуры сопоставимы (сравнимы) друг с другом*: parent (joe, tammy) сопоставимо с parent (joe, tammy).

Однако, сопоставление (сравнение) обычно использует одну или несколько свободных переменных. Например, если X свободна, то parent (joe, X) сопоставимо с parent (joe, tammy) и X принимает значение (связывается с) «tammy».

Если же X уже связана, то она действует также, как обычная константа. Таким образом, если X связана со значением «tammy», то parent (joe, X) сопоставимо с parent (joe, tammy), но parent (joe, X) не сопоставимо с parent (joe, millie).

Во втором примере сопоставление не выполняется, так как если переменная становится связанной, то ее значение не может изменяться.

Как может переменная оказаться уже связанной при попытке Пролога сопоставления ее с чем-либо? Вспомните, что переменные не могут хранить значения, так как они становятся связанными только на промежуток времени, необходимый для отыскания (или попытки отыскания) одного решения одной цели. Поэтому, имеется только одна возможность того, что переменная может оказаться связанной перед попыткой сопоставления – если цель требует больше одного шага, и переменная стала связанной на предыдущем шаге.

Например,

parent (joe, X), parent (X, jenny)

является корректной целью. Она означает: «Найти кого-либо, являющегося ребенком Joe и родителем Jenny». Здесь при достижении подцели parent (X, jenny) переменная X уже будет связана. Если для подцели parent (X, jenny) нет решений, Пролог «развяжет» переменную X и вернется назад, пытаясь найти новое решение для parent (joe, X), а затем проверит, будет ли работать parent (X, jenny) с новым значением X.

Две свободные переменные могут сопоставляться друг с другом. Например, parent (joe, X) сопоставляется с parent (joe, Y), связывая при этом переменные X и Y между собой. С момента «связывания» X и Y трактуются как одна переменная и любое изменение значения одной из них приводит к немедленному соответствующему изменению другой. В случае подобного «связывания» между собой нескольких свободных переменных все они

называются совмещенными *свободными переменными*. Некоторые действительно мощные методы программирования специально используют «взаимосвязывание» свободных переменных, являющихся, на самом деле, различными.

В Прологе связывание переменных (со значениями) производится двумя способами: на входе и выходе. Направление, в котором передаются значения, указывается в *шаблоне потока параметров* (flowpattern). В дальнейшем мы для краткости будем опускать слово «шаблон» и говорить просто «поток параметров». Когда переменная передается в предложение, она считается *входным аргументом* и обозначается (i); когда же переменная возвращается из предложения, она является *выходным аргументом* и обозначается (o).

Выводы

Ниже сведены основные идеи, введенные в данной главе:

1 Пролог-программа состоит из предложений, которые могут быть фразами двух типов: фактами или правилами.

Факты – это связи или свойства, о которых вы (программист) твердо знаете, что они истинны; правила – это зависимые связи (отношения); они позволяют Прологу выводить один фрагмент информации из другого.

2 Факты имеют общий вид:

property (object1, object2, ..., objectN)

или

relation (object1, object2, ..., objectN),

где property – это свойство объектов, а relation – отношение между объектами. Различия между этими понятиями несущественны, и, поэтому, в дальнейшем мы будем использовать термин *отношение*.

3 Каждый факт программы задает либо отношение, влияющее на один или более объектов, либо свойство одного или более объектов.

Например, в факте Пролога

likes (tom, baseball).

Отношение – это likes (нравится), а объекты – tom и baseball (бейсбол); Тому нравится бейсбол.

В другом факте

left_handed (benjamin)

left_handed (левый крайний) является свойством объекта benjamin;

другими словами, Бенджамин – левый крайний.

4 *Правила* имеют общую форму «Заголовок: – Тело», которые выглядят в соответствии со следующей программой:

relation (object, object,..., object):
relation (object, ..., object),

...

relation (object, ..., object).

5 Сообразуясь со следующими ограничениями, вы можете устанавливать любые имена для связей и объектов в своей программе: имя объекта должно начинаться со строчной буквы, за которой может быть любое число символов. Этими символами могут быть: буквы верхнего и нижнего регистров, цифры и символы подчеркивания; имена свойств и связей должны начинаться со строчной буквы, за которой может следовать любая комбинация букв, цифр и символов подчеркивания.

6 Предикат – это символическое имя (идентификатор) связи с последовательностью аргументов. Программа на Прологе – это последовательность предложений и директив, а процедура – это последовательность предложений, описывающих предикат. Предложения, принадлежащие одному предикату, должны следовать друг за другом.

7 Переменные позволяют вам записывать общие факты и правила и задавать общие вопросы. Имя переменной в Visual Prolog должно начинаться с заглавной буквы или символа подчеркивания (_), после которой вы можете использовать любое число букв (верхнего и нижнего регистра), цифр и символов подчеркивания; переменные в Прологе получают свои значения в результате сопоставления с константами в фактах или правилах. До получения значения переменная является свободной, после – становится связанной; вы не можете длительно хранить информацию с помощью «связывания» переменной со значением, так как переменная является связанной только в пределах предложения.

8 Если в запросе вас интересует только определенная информация, то для игнорирования не нужных вам значений вы можете использовать анонимные переменные. В Прологе анонимные переменные обозначаются одиночным символом подчеркивания (_). Анонимная переменная может быть использована вместо любой другой переменной; она сопоставляется с чем угодно. Анонимная переменная никогда не принимает какого-либо значения.

9 Задание Прологу вопросов о фактах в вашей программе называется запросами к системе Пролога; более общим термином для запроса является цель(goal). Пролог пытается разрешить цель (ответить на вопрос), начиная с первого факта и просматривая все факты до достижения последнего из них.

10 Составная цель – это цель, включающая две или более частей; каждая часть составной цели называется подцелью. Составная цель может быть конъюнктивной (подцель А и подцель В) или дизъюнктивной (подцель А или подцель В).

11 Комментарии делают вашу программу более удобной для чтения. Вы можете заключать комментарии в разделители /*как здесь */ или предварять их одним символом процента, % как здесь.

12 В Прологе имеется несколько способов сопоставления одного обьекта с другим: идентичные структуры сопоставляются друг с другом; свободная переменная сопоставляется с константой или с ранее связанной переменной (и становится связанной с соответствующим значением); две свободные переменные могут сопоставляться (и связываться) друг с другом. С момента связывания они трактуются как одна переменная: если одна из них принимает какое-либо значение, то вторая немедленно принимает то же значение.

8.3 Основные разделы Visual Prolog программ

Обычно программа на Visual Prolog состоит из четырех основных программных разделов. К ним относятся: раздел clauses (предложений), раздел predicates (предикатов), раздел domains (доменов) и раздел goal (целей):

1 Раздел clauses – это сердце Visual Prolog программы; именно в этот раздел вы записываете факты и правила, которыми будет оперировать Visual Prolog, пытаясь разрешить цель программы.
2 Раздел predicates – это тот, в котором вы объявляете свои предикаты и домены (типы) их аргументов (вам не нужно объявлять предикаты, встроенные в Visual Prolog).
3 Раздел domains служит для объявления всех используемых вами доменов, не являющихся стандартными доменами Visual Prolog (вам не нужно объявлять стандартные домены).
4 Раздел goal – это тот, в который вы помещаете цель Visual Prolog программы.

Раздел предложений

В раздел clauses (предложений) вы помещаете все факты и правила, составляющие вашу программу. Основное внимание в разделе 8 было сосредоточено вокруг предложений (фактов и правил) вашей программы: что они означают, как их писать и т. д.

Если вы поняли, что собой представляют факты и правила и как их записывать в Прологе, то вы знаете, что должно быть в разделе clauses. Все предложения для каждого конкретного предиката в разделе clauses должны

располагаться вместе; последовательность предложений, описывающих один предикат, называется процедурой.

Пытаясь разрешить цель, Visual Prolog, начиная с первого предложения раздела clauses, будет просматривать каждый факт и правило, стремясь найти сопоставление. По мере продвижения Visual Prolog вниз по разделу clauses, он устанавливает внутренний указатель на каждое предложение, следующее по отношению к сопоставляемому с текущей подцелью. Если текущее предложение не является частью логического пути, ведущего к решению, то Visual Prologвозвращается к установленному указателю и ищет следующее сопоставление – этот процесс называется *поиском с возвратом*.

Раздел предикатов

Если в разделе clauses программы на Visual Prolog вы описали собственный предикат, то вы обязаны объявить этот предикат в разделе predicates; в противном случае Visual Prolog не поймет, о чем вы ему говорите. В результате объявления предиката вы сообщаете Visual Prolog, к каким доменам (типам) принадлежат аргументы этого предиката.

Visual Prolog поставляется с богатым набором встроенных предикатов. Вам не нужно объявлять какие-либо встроенные предикаты Visual Prolog, используемые в вашей программе. Интерактивное справочное руководство по Visual Prolog предоставляет полное описание всех встроенных предикатов.

Предикаты задают факты и правила. В разделе же predicates все предикаты просто перечисляются с указанием типов (доменов) их аргументов. Хотя сердцем программы является раздел clauses, эффективность работы Visual Prolog значительно возрастает именно из-за того, что вы объявляете типы объектов (аргументов), с которыми работают ваши факты и правила.

Как объявить пользовательский предикат

Объявление предиката начинается с имени этого предиката, за которым следует открывающая (левая) круглая скобка, после чего следует ноль или больше доменов (типов) аргументов предиката:

predicate Name(argument_type 1 Optional Name1,
argument_type2 Optional Name2, ...,
argument_typeN Optional Name3)

После каждого домена (типа) аргумента следует запятая, а после последнего типа аргумента – закрывающая (правая) скобка. Отметим, что, в отличие от предложений в разделе вашей программы clauses, декларация предиката не завершается точкой. Доменами (типами) аргументов предиката могут быть либо стандартные домены, либо домены, объявленные вами в разделе domains.

Также вы можете указывать имена аргументов OptionalNameK, это улучшает читаемость программы, а компилятор просто игнорирует их.

Имена предикатов

Имя предиката должно начинаться с буквы, за которой может следовать последовательность букв, цифр и символов подчеркивания. Регистр букв не имеет значения, однако мы не советуем вам использовать заглавные буквы в качестве первой буквы имени предиката (другие версии Пролога не разрешают, чтобы имя предиката начиналось с заглавной буквы, и будущая версия Visual Prolog версия 6, тоже будет запрещать это). Имя предиката может иметь длину до 250 символов.

В именах предикатов вы не можете использовать пробел, символ минус, звездочку и другие алфавитно-цифровые символы. Корректные имена Visual Prolog могут включать символы, перечисленные в табл. 8.2.

Имена предикатов и аргументов могут состоять из любых комбинаций этих символов, при условии что вы подчиняетесь правилам построения соответствующих имен.

Таблица 8.2 **Имена в Visual Prolog**

Название символов	**Примеры символов**
Заглавные буквы	: A, B, ... , Z
Строчные буквы	: a, b, ... , z
Цифры	: 0, 1, ... , 9
Символ подчеркивания	: _

В табл. 8.3 приведены корректные и некорректные имена предикатов.

Таблица 8.3 **Имена предикатов в Visual Prolog**

Корректные имена предикатов	Некорректные имена предикатов
fact	[fact]
is_a	*is_a*
has_a	has/a
patternCheckList	pattern-Check-List
choose_Menu_Item	choose Menu Item
predicateName	predicate<Name>
first_in_10	>first_in_10

Аргументы предикатов

Аргументы предикатов должны принадлежать доменам, известным Visual Prolog. Эти домены могут быть либо стандартными доменами, либо некоторыми из тех, что вы объявили в раздели доменов.

Примеры

1 Если вы объявляете предикат my_predicate(symbol, integer) в разделе predicates следующим образом:

```
predicates
my_predicate (symbol, integer)
```

то вам не нужно в разделе domains декларировать домены его аргументов, так как symbol и integer – стандартные домены.

Однако, если этот же предикат вы объявляете следующим образом:

```
predicates
my_predicate (name, number)
```

то вам необходимо объявить, что name (символический тип) и number (целый тип) принадлежат к стандартным доменам symbol и integer, т. е.

```
domains
name = symbol
number = integer
predicates
my_predicate (name, number)
```

2 Следующий фрагмент программы показывает несколько различных объявлений доменов и предикатов:

```
domains
    person, activity = symbol
    car, make, color = symbol
    mileage, years_on_road, cost = integer
predicates
    likes (person, activity)
    parent (person, person)
    can_buy (person, car)
    car (make, mileage, years_on_road, color, cost)
    green (symbol)
    ranking (symbol, integer)
```

Этот фрагмент сообщает следующую информацию об этих предикатах и их аргументах:

- предикат likes имеет два аргумента (person и activity), причем они оба принадлежат домену symbol (что означает, что их значениями являются идентификаторы, а не числа);

- предикат parent имеет два аргумента (person), причем каждый из person относится к отдельному домену symbol;
- предикат can_buy имеет два аргумента (person и car), которые относятся к типу symbol;
- предикат car имеет 5 аргументов: make и color относятся к домену symbol, а mileage, years_on_road и cost – к домену integer**;**
- предикат green имеет один аргумент типа symbol: нет необходимости декларировать этот тип аргумента, так как он относится к стандартному домену symbol;
- предикат ranking имеет два аргумента, каждый из которых принадлежит к стандартному домену (symbol и integer), поэтому декларировать типы этих аргументов не требуется.

Раздел доменов

В традиционном Прологе есть только один тип – *терм*. В Visual Prolog мы обьявляем домены всех аргументов предикатов.

Домены позволяют вам задавать различные имена различным видам данных, которые, в противном случае, будут выглядеть абсолютно одинаково. В Visual Prolog объекты в отношениях (аргументы предикатов) принадлежат доменам, причем это могут быть как стандартные, так и описанные вами специальные домены.

Раздел domains служит двум очень полезным целям. Во-первых, вы можете задать доменам осмысленные имена, даже если внутренне эти домены аналогичны уже имеющимся стандартным. Во-вторых, объявление специальных доменов используется для описания структур данных, не имеющихся в стандартных доменах.

Иногда очень полезно описать новый домен – особенно, когда вы хотите прояснить отдельные части раздела predicates. Объявление собственных доменов, благодаря присваиванию осмысленных имен типам аргументов, помогает документировать описываемые вами предикаты.

Примеры

1 Данный пример показывает, как объявление доменов помогает документировать ваши предикаты:

Франк – мужчина, которому 45 лет.

Используя стандартные домены, вы можете так объявить соответствующий предикат

person (symbol, symbol, integer).

В большинстве случаев такое объявление будет работать очень хорошо. Однако предположим, что несколько месяцев спустя после написания

программы, вы решили скорректировать ее. Но уже через шесть месяцев подобное объявление этого предиката абсолютно ничего вам не скажет. И напротив, декларация этого же предиката, представленная ниже, поможет вам разобраться в том, что же представляют собой аргументы данного предиката:

```
domains
name, sex = symbol
age       = integer
predicates
person (name, sex, age)
```

Одним из главных преимуществ объявления собственных доменов является то, что Visual Prolog может отслеживать ошибки типов, как, например, при следующей очевидной ошибке:

```
same_sex(X, Y) :-
person (X, Sex, _),
person (Sex, Y, _).
```

Несмотря на то, что и name и sex описываются как symbol, они не эквивалентны друг другу. Это и позволяет Visual Prolog определить ошибку в случае, если вы перепутаете их. Данная особенность очень полезна, особенно в случаях, когда ваши программы очень велики и сложны.

Вы можете пожелать узнать, почему же мы не можем использовать специальные домены для объявления всех аргументов, ведь такие домены привносят намного больше смысла в обозначение аргументов. Ответ заключается в том, что *аргументы с типами из специальных доменов не могут смешиваться между собой, даже если эти домены одинаковы!* Именно поэтому, несмотря на то, что name и sex принадлежат одному домену symbol, они не могут смешиваться. Однако, все собственные домены пользователя могут быть сопоставлены стандартным доменам.

2 Следующий пример программы приведет (при его загрузке) к *ошибке*:

Листинг 5. Программа ch03e01.pro

```
domains
product,sum = integer
predicates
add_em_up (sum,sum,sum)
multiply_em (product,product,product)
clauses
add_em_up (X,Y,Sum):-
Sum=X+Y.
```

multiply_em (X,Y,Product):-
*Product=X*Y.*

Эта программа выполняет две операции: складывает и умножает.

Зададим ей следующую цель:

add_em_up (32, 54, Sum).
Visual Prolog (Test Goal) ответит:
Sum=86
1 Solution,

что является суммой двух целых чисел, которые вы передали в программу.

С другой стороны эта же программа с помощью предиката multiply_em умножает два аргумента. Поэкспериментируем. Если вам нужно узнать произведение 13 и 31, то введите цель:

multiply_em (31, 13, Product).

Visual Prolog вернет вам корректный результат:

Product=403
1 Solution.

А теперь предположим, что вам понадобилась сумма 42 и 17; цель для этого выглядит так:

add_em_up (42, 17, Sum).

Теперь же вы хотите удвоить произведение 31 на 17 и, поэтому, задаете следующую цель:

multiply_em (31, 17, Sum), add_em_up (Sum, Sum, Answer).

Выждете, чтоVisual Prolog (TestGoal) ответит:

Sum=527, Answer=1054
1 Solution.

Однако вместо этого вы получите ошибку типа. Это случится из-за того, что вы попытались передать результирующее значение предиката multiply_em (которое относится к домену product) в качестве первого и второго аргументов в предикат add_em_up, которые относятся к домену sum. Это и привело к ошибке, так как домен product отличается от домена sum. И хотя оба эти домена, в действительности, соответствуют типу integer – это различные домены. Поэтому, если переменная в предложении используется более чем в одном предикате, она должна быть одинаково объявлена в каждом из этих предикатов. Очень важно, чтобы вы полностью поняли концепцию описанной здесь ошибки типа; знание этой концепции позволит вам избегать сообщений об ошибках компиляции. Далее в этой главе мы опишем различные автоматические и явные преобразования типов, предлагаемые Visual Prolog.

З В свете еще более полного понимания того, как вы можете использовать объявления доменов для отслеживания ошибок типа, рассмотрим следующий пример программы.

Листинг 6. Программа ch03e02.pro

```
domains
brand,color = symbol
  age = byte
price, mileage = ulong
predicates
car (brand,mileage,age,color,price)
clauses
car (chrysler,130000,3,red,12000).
car (ford,90000,4,gray,25000).
car (datsun,8000,1,black,30000).
```

Здесь предикат car, объявленный в разделе predicates, имеет пять аргументов. Один из них относится к домену age типа byte. В семействе процессоров x86 byte – это 8-битное беззнаковое целое, которое может принимать значения от 0 до 255, включая границы. Аналогично, домены mileage и price типа ulong, который представляет собой 32-битные беззнаковые целые, а домены brand и color – символьного типа (symbol).

Мы обсудим стандартные домены более детально далее. А сейчас загрузите данную программу в проект TestGoal и попытайтесь вычислить по очереди следующие цели:

```
car (renault, 13, 40000, red, 12000).
car (ford, 90000, gray, 4, 25000).
car (1, red, 30000, 80000, datsun).
```

Каждая из этих целей приведет к ошибке типа. В первом случае, например, это произойдет из-за того, что age должен быть типа byte. Следовательно, Visual Prolog сможет легко определить, что при вводе этой цели объекты mileage и age в предикате car были перепутаны местами. Во втором случае были перепутаны age и color, а в третьем – попытайтесь найти ошибку самостоятельно.

Раздел цели

По существу, раздел goal(цели) аналогичен телу правила: это просто список подцелей. Цель отличается от правила лишь следующим:

1) за ключевым словом goal не следует:- ;
2) при запуске программы Visual Prolog автоматически выполняет цель.

Это происходит так, как будто Visual Prolog вызывает goal, запуская тем самым программу, которая пытается разрешить тело правила goal. Если все подцели в раздели goal истинны – программа завершается успешно. Если же какая-то подцель из раздела goal ложна, то считается, что программа

завершается неуспешно (хотя чисто внешне никакой разницы в этих случаях нет – программа просто завершит свою работу).

Подробнее о декларациях и правилах

В Visual Prolog есть несколько встроенных стандартных доменов. Использовать эти стандартные домены вы можете при декларации типов аргументов предикатов. Так как стандартные домены уже известны Visual Prolog, их описание в разделе domains не требуется. Основные стандартные домены показаны в табл. 8.4.

Таблица 8.4 **Основные стандартные домены**

Домен	Описание	Имплементация		
short	Короткое, знаковое, количественное	Все платформы	16 бит	-32 768 – 32 767
ushort	Короткое, беззнаковое, количественное	Все платформы	16 бит	0 – 65 535
long	Длинное, знаковое, количественное	Все платформы	32 бит	-2 147 483 648 – 2 147 483 647
Ulong	Длинное, беззнаковое, количественное	Все платформы	32 бит	0 – 4 294 967 295
integer	Знаковое, количественное, имеет платформенно зависимый размер	16-битные платформы	16 бит	-32 768 – 32 767
		32-битные платформы	32 бит	-2 147 483 648 – 2 147 483 647
unsigned	Беззнаковое, количественное, имеет платформенно зависимый размер	16-битные платформы	16 бит	0 – 65 535
		32-битные платформы	32 бит	0 – 4 294 967 295
byte		Все платформы	8 бит	0 – 255
word		Все платформы	16 бит	0 – 65 535
dword		Все платформы	32 бит	0 – 4 294 967 295

Синтаксически, значение, принадлежащее одному из целочисленных доменов, записывается как последовательность цифр, которой в случае знакового домена может предшествовать не отделенный от нее пробелом знак минус. Имеются также восьмиричные и шестнадцатиричные синтаксисы для основных доменов. Домены типов byte, word и dword наиболее полезны при работе с машинными числами. В основном используются integer и unsigned

типы, а также short and long (и их беззнаковые аналоги) для более специализированных приложений. В объявлениях доменов, ключевые слова signed и unsigned могут использоваться вместе со стандартными доменами типов byte, word и dword для построения новых базовых доменов, как то

domains

i8 = signedbyte

создает новый базовый домен диапазона от -128 до +127.

Другие базовые домены показаны в табл. 8.5.

Таблица 8.5 **Основные стандартные домены**

Домен	Описание и имплементация
char	Символ, имплементируемый как беззнаковый byte. Синтаксически, это символ, заключенный между двумя одиночными кавычками: 'a'.
real	Число с плавающей запятой, имплементируемо как 8 байт в соответствии с соглашением IEEE; эквивалентен типу double в C. Синтаксически, числа с необязательным знаком (+ или -), за которым следует несколько цифр DDDDDDD, затем – необязательная десятичная точка(.) и еще цифры DDDDDDD, за которыми необязательная экспоненциальная часть ($e^{(+ \text{ или } -)}$DDD): <+|-> DDDDD <.> DDDDDDD <e $^{<+\|->}$ DDD> Примеры действительных (real) чисел: 42705 9999 86.72 9111.929437521e^{238} 79.83e^{+21} Здесь 79.83e^{+21} означает 79.83×10^{21}, как и в других языках. Допустимый диапазон действительных чисел: от 1*10^{-307} до 1*10^{+308} (от 1e^{-307} до 1e^{+308}). При необходимости, целые автоматически преобразуются в действительные
string	Последовательность символов, имплементируемых как указатель на массив байт, завершаемый нулем, как в C. Для строк допускается два формата: 1. Последовательность букв, цифр и символов подчеркивания, причем первый символ должен быть строчной буквой. 2. Последовательность символов, заключенных в двойные кавычки. Примеры строк: `telephone_number "railway ticket" "Dorid Inc"` Строки, которые вы пишете в программе, могут достигать длины в 255 символов, в то время как строки, которые система Visual Prolog считывает из файла или строит внутри себя, могут достигать (теоретически) до 4 Гбайт на 32-битных платформах
symbol	Последовательность символов, имплементируемых как указатель на вход в таблице идентификаторов, хранящей строки идентификаторов. Синтаксис такой, как для строк

Visual Prolog знает и несколько других стандартных доменов, но мы расскажем о них в других подразделах, когда вы уже будете знать лучше основы языка.

Идентификаторы и строки взаимозаменяемы в вашей программе, однако Visual Prolog хранит их раздельно. Идентификаторы хранятся в таблице идентификаторов, а для представления ваших идентификаторов используются лишь их индексы в этой таблице, а не сами строки идентификаторов.

Это означает, что сопоставление идентификаторов выполняется очень быстро, а в случае, если они встречаются в программе несколько раз, то и хранение их выполняется очень компактно. Строки же не хранятся в поисковой таблице, и при необходимости сопоставления Visual Prolog проверяет их символ за символом. Вы сами должны определять, какой домен лучше использовать в каждой конкретной программе.

В табл. 8.6 мы приводим несколько примеров простых объектов, принадлежащих к основным стандартным доменам.

Таблица 8.6 **Примеры простых объектов**

"&&", caitlin, "animal lover", b_l_t	(symbol или string)
-1, 3, 5, 0	(integer)
3.45, 0.01, -30.5, 123.4e+5	(real)
'a', 'b', 'c', '/', '&'	(char)

Задание типов аргументов при декларации предикатов

Объявление доменов аргументов в разделе predicates называется заданием типов аргументов. Например, предположим, что у вас имеется следующая связь объектов:

Франк – мужчина, которому 45 лет.

Факт Пролога, соответствующий этому предложению естественного языка, может быть следующим:

person (frank, male, 45).

Для того, чтобы объявить person (человек), как предикат с этими тремя аргументами, вы можете разместить в разделе predicates следующую строку:

person (symbol, symbol, unsigned).

Здесь для всех трех аргументов вы использовали стандартные домены. Отныне, всякий раз, когда бы вы не использовали предикат person, вы должны передавать ему три аргумента, причем первые два должны быть типа symbol, а третий – типа integer.

Если в вашей программе используются только стандартные домены, то нет необходимости использовать раздел domain; вы уже видели несколько программ такого типа.

Или, предположим, что вы хотите описать предикат, который сообщал бы вам позицию буквы в алфавите, т. е. цель

alphabet_position (Letter, Position)

должна вернуть вам Position = 1, если Letter = a, Position = 2, если Letter = b и т. д. Предложения этого предиката могут выглядеть следующим образом:

alphabet_position (A_character, N).

Если при объявлении предиката используются только стандартные домены, то программе не нужен раздел domains.

Предположим, что вы хотите описать предикат так, что цель будет истинна, если A_character является N-м символом алфавита. Предложения этого предиката будут такими:

```
alphabet_position ('a', 1).
alphabet_position ('b', 2).
alphabet_position ('c', 3).
...
alphabet_position ('z', 26).
```

Вы можете объявить данный предикат следующим образом:

```
predicates
alphabet_position (char, unsigned)
```

и тогда вам не будет нужен раздел domains. Если разместить все фрагменты программы вместе, получим:

```
predicates
alphabet_position (char, integer)
clauses
alphabet_position ('a', 1).
alphabet_position ('b', 2).
alphabet_position ('c', 3).
%    здесь находятся остальные буквы
alphabet_position ('z', 26).
```

Ниже представлено несколько простых целей, которые вы можете использовать:

```
alphabet_position ('a', 1).
alphabet_position (X, 3).
alphabet_position ('z', What).
```

Упражнения

1 Программа ch03e04.pro представляет собой законченную программу на Visual Prolog, служащую небольшим телефонным справочником. Так как

используются только стандартные предикаты, раздел domains в этой программе не нужен.

Листинг 7. Программа ch03e04.pro

```
predicates
phone_number (symbol,symbol)
clauses
phone_number ("Albert","EZY-3665").
phone_number ("Betty","555-5233").
phone_number ("Carol","909-1010").
phone_number ("Dorothy","438-8400").
goal
```

Загрузите и запустите программу ch03e04.pro, а затем, по очереди, задайте ей следующие цели:

```
phone_number ("Carol", Number).
phone_number (Who, "438-8400").
phone_number ("Albert", Number).
phone_number (Who, Number).
```

Теперь измените предложения.

Предположим, что Kim и Dorothy имеют один номер телефона. Добавим этот факт в разделю clauses и введем цель:

```
phone_number (Who, "438-8400").
```

На этот запрос вы должны получить два решения:

```
Who=Dorothy
Who=Kim
2 Solutions .
```

2 Для иллюстрации домена char в программе ch03e05.pro описан предикат isletter (является_буквой), который в случае задания ему нижеприведенных целей

```
isletter ('%').
isletter ('Q').
```

возвращает «No» и «Yes» соответственно.

Листинг 8. Программа ch03e05.pro

```
predicates
isletter (char)
```

```
clauses
  % Будучи примененным к символам, '<=' обозначает
  % "предшествование или равенство позиций в алфавите"
isletter (Ch):-
'a' <= Ch,
Ch <= 'z'.
isletter (Ch):-
'A' <= Ch,
Ch <= 'Z'.
```

Загрузите и запустите программу ch03e05.pro в TestGoalи испытайте, по очереди, каждую из этих целей:

```
isletter ('x').
isletter ('2').
isletter ("hello").
isletter (a).
isletter (X).
```

Цели (c) и (d) приведут к сообщению об ошибке типа, а цель (e) вернет сообщение «Freevariable»(несвязанная переменная), так как вы не можете проверить старшинство незаданного объекта по отношению к «a» или «z».

Арность (размерность)

Арность предиката – это количество аргументов, которые он принимает. Вы можете иметь два предиката с одним и тем же именем, но разной арностью. В разделах predicates и clauses версии предикатов с одним именем и разной арностью должны собираться вместе; за исключением этого ограничения, различная арность всегда понимается как полное различие предикатов.

Листинг 9. Программа ch03e06.pro

```
domains
person = symbol
predicates
father (person)    % этот person – отец
father (person, person)    % первый person является отцом
другого
clauses
father (Man):-
father (Man,_).
father (adam,seth).
father (abraham,isaac).
```

Синтаксис правил

Правила используются в Прологе в случае, когда какой-либо факт зависит от истинности другого факта или группы фактов. Как мы объясняли в главе 8, в правиле Пролога есть две части: заголовок и тело. Ниже представлен обобщенный синтаксис правила в Visual Prolog:

HEAD :- <Subgoal>, <Subgoal>, ...,<Subgoal>.

Заголовок :- <Подцель>, <Подцель>, ..., <Подцель>.

Тело правила состоит из одной или более подцелей. Подцели разделяются запятыми, определяя конъюнкцию, а за последней подцелью правила следует точка.

Каждая подцель выполняет вызов другого предиката Пролога, который может быть истинным или ложным. После того, как программа осуществила этот вызов, Visual Prolog проверяет истинность вызванного предиката, и если это так, то работа продолжается, но уже со следующей подцелью. Если же в процессе такой работы была достигнута точка, то все правило считается истинным; если хоть одна из подцелей ложна, то все правило ложно.

Для успешного разрешения правила Пролог должен разрешить все его подцели и создать последовательный список переменных, должным образом связав их. Если же одна из подцелей ложна, Пролог вернется назад для поиска альтернативы предыдущей подцели, а затем вновь двинется вперед, но уже с другими значениями переменных. Этот процесс называется поиск с возвратом.

Ключевое слово Пролога "if" против "IF" в других языках

Кау уже упоминалось выше, в качестве разделителя заголовка и тела правила используют знак :-, который читается как «если» (if). Однако прологовский if отличается от IF, написанного в других языках, например в Паскале.

В Паскале, например, условие, содержащееся в операторе IF, должно быть указано перед телом оператора, который может быть выполнен, другими словами:

"if HEAD is true, then BODY is true (or: then do BODY)"

(если ЗАГОЛОВОК истинен, тогда ТЕЛО истинно (или: тогда выполнить ТЕЛО)).

Данный тип оператора известен как *условный оператор если/тогда* (if/then). Пролог же использует другую форму логики в таких правилах. Вывод об истинности заголовка правила Пролога делается, если (после того, как) тело этого правила истинно; другими словами:

"HEAD is true if BODY is true (or: if BODY can be done)".

ЗАГОЛОВОК истиннен, если ТЕЛО – истинно (или: если ТЕЛО может быть выполнено).

Сообразуясь со сказанным, правило Пролога соответствует условной форме тогда/если.

Автоматическое преобразование типов

Совсем не обязательно, чтобы при сопоставлении двух Visual Prolog переменных они принадлежали одному и тому же домену. Иногда переменные могут быть связаны с константами из разных доменов. Такое (избирательное) смешение допускается, так как Visual Prolog автоматически выполняет преобразование типов (из одного домена в другой), но только в следующих случаях:

1 Между строками (string) и идентификаторами (symbol).

2 Между целыми, действительными и символами (char). При преобразовании символа в числовое значение этим значением является величина символа в коде ASCII.

Аргумент из домена my_dom, который объявлен следующим образом:

domains

my_dom = <basedomain>% <basedomain> – это стандартный домен может свободно смешиваться с аргументами из этого основного домена и всех совместимых с ним стандартных доменов. Если основной домен – string, то с ним совместимы аргументы из домена symbol; если же основной домен integer, то с ним совместимы домены real, char, word и др.

Такое преобразование типов означает, например, что вы можете:

- вызвать предикат с аргументами типа string, задавая ему аргументы типа symbol, и наоборот;
- передавать предикату с аргументами типа real параметры типа integer;
- передавать предикату с аргументами типа char параметры типа integer;
- использовать в выражениях и сравнениях символы без необходимости получения их кодов в ASCII.

Существует набор правил, определяющих, к каком домену принадлежит результат смешивания разных доменов.

Другие разделы программ

Теперь, когда вы достаточно знакомы с такими разделами программ Visual Prolog, как clauses, predicates, domains и goal, немного поговорим о некоторых других часто используемых разделах программ: facts, constants и различных глобальных (global) разделах. Это будет только введение – по мере дальнейшей работы с остальным обучающим материалом в этом руководстве, вы более подробно изучите эти разделы и их использование в ваших программах.

Раздел фактов

Программа на Visual Prolog представляет собой набор фактов и правил. Иногда, в процессе работы такой программы, вы можете захотеть модифицировать (изменить, удалить или добавить) некоторые из фактов, с которыми она работает. В этом случае факты рассматриваются как *динамическая* или *внутренняя* база данных, которая в процессе работы программы может изменяться. Для объявления фактов программы, рассматривающихся как часть динамической (или изменяющейся) базы данных, Visual Prolog включает специальный раздел – facts.

Ключевое слово factsобъявляет раздел фактов. Именно в этой секции вы объявляете факты, включаемые в динамическую базу данных. Отметим что в ранних версиях Visual Prolog для обьявления раздела фактов использовалось ключевое слово database. То есть, ключевое слово facts – синоним устаревшего ключевого слова database**.** В Visual Prolog есть несколько встроенных предикатов, облегчающих вам использование динамических фактов.

Раздел констант

В своих программах на Visual Prolog вы можете объявлять и использовать символические константы. Раздел для объявления констант обозначается ключевым словом constants, за которым следуют сами объявления, использующие следующий синтаксис:

<Id> = <Макроопределение>,

где <Id> – имя символической константы, а <Макроопределение> – это то, что вы присваиваете этой константе. Каждое <Макроопределение> завершается символом новой строки и, следовательно, на одной строке может быть только одно описание константы. Объявленные таким образом константы могут позже использоваться в программах.

Рассмотрим следующий фрагмент программы:

```
constants
zero = 0
one  = 1
two  = 2
hundred = (10*(10-1)+10)
pi   = 3.141592653
ega  = 3
slash_fill = 4
red  = 4
```

Перед компиляцией программы Visual Prolog заменит каждую константу на соответствующую ей строку.

Например, фрагмент программы

```
...,
A = hundred*34, delay (A),
setfillstyle (slash_fill, red),
Circumf = pi*Diam,
...
```

будет обрабатываться компилятором как следующий фрагмент:

```
...,
A = (10*(10-1)+10)*34, delay (A),
setfillstyle (4, 4),
Circumf = 3.141592653*Diam,
...
```

На использование символических констант накладываются следующие ограничения:

1) описание константы не может ссылаться само на себя:

```
my_number = 2*my_number/2    % недопускается
```

приведет к сообщению об ошибке «Recursion in constant definition» (рекурсия в описании константы).

2) в описаниях констант система не различает верхний и нижний регистры. Следовательно, при использовании в разделе программы clauses идентификатора типа constants его первая буква должна быть строчной для того, чтобы избежать путаницы между константами и переменными. Поэтому следующий фрагмент программы, например, является допустимой конструкцией:

```
constants
Two = 2
goal
A=two, write (A).
```

3) в программе может быть несколько разделов constants, однако объявление константы должно производиться перед ее использованием;

4) идентификаторы констант являются глобальными и могут объявляться только один раз. Множественное объявление одного и того же идентификатора приведет к сообщению об ошибке «Constant identifier can only be declared once» (идентификатор константы может объявляться только один раз).

Глобальные разделы

Visual Prolog позволяет вам объявлять в вашей программе некоторые разделы domains, predicates, clauses глобальными (а не локальными); сделать это вы можете, обьявив в своей программе специальные разделы global domains, global predicates и global facts.

Директивы компилятора

Visual Prolog поддерживает несколько *директив компилятора*, которые вы можете добавлять в свою программу для сообщения компилятору специальных инструкций по обработке вашей программы при ее компиляции.

Кроме этого, вы можете устанавливать большинство директив компилятора с помощью команды меню среды визуальной разработки Visual Prolog **Options** | **Project** | **Compiler Options**.

Директива include

После более общего знакомства с использованием Visual Prolog, вы, возможно, захотите вновь и вновь использовать в своих программах некоторые процедуры. Для того, чтобы избежать многократного набора этих процедур, вы можете использовать директиву include.

Ниже приведен пример того, как это делается.

1 Вы создаете файл (например, MYSTUFF.PRO), в котором объявляете свои наиболее часто используемые предикаты (с помощью разделов domains и predicates) и даете их описание в разделе clauses.

2 Вы пишете исходный текст программы, которая будет использовать эти процедуры.

3 В «допустимых областях» исходного текста вашей программы размещаете строку:

include "mystuff.pro".

«Допустимые области» – это любое место вашей программы, в котором вы можете расположить декларацию разделов domains, facts, predicates, clauses или goal.

4 При компиляции ваших исходников, Visual Prolog вставит содержание файла MYSTUFF.PRO прямо в окончательный текст файла для компиляции.

Директиву include вы можете использовать для включения в свой исходный текст практически любого часто используемого фрагмента, и, кроме того, один включаемый файл может, в свою очередь, включать другой (однако каждый файл может быть включен в вашу программу только один раз). Директива include может располагаться в любых «допустимых областях» вашей программы. Однако, при включении файла в исходный текст, вы должны следить за тем, чтобы этот файл не нарушил структуру программы.

Выводы

Ниже приведены основные идеи, представленные нами в этой главе:

1 Программа на Visual Prolog имеет следующую обобщенную структуру:

```
domains
    /* ...
    объявления доменов
...   */
predicates
    /* ...
    объявления предикатов
...   */
clauses
    /* ...
    предложения (правила и факты)
...   */
goal
    /* ...
    подцель_1,
    подцель _2,
    и т. д. */
```

2 Раздел clauses – в котором вы размещаете факты и правила, с которыми будет работать Visual Prolog, пытаясь разрешить цель программы.

3 Раздел predicates – в котором вы объявляете свои предикаты и домены (типы) аргументов этих предикатов. Имена предикатов должны начинаться с буквы (желательно строчной), за которой следует последовательность букв, цифр и символов подчеркивания (до 250 знаков). В именах предикатов нельзя использовать символы пробел, минус, звездочка, слэш. Объявление предиката имеет следующую форму:

```
predicates
predicateName (argument Type1 Optional Name1,
               argument Type2 Optional Name2,
               ...,
               argument TypeN Optional NameN)
```

Здесь *argument_type1, ..., argument_typeN* – либо стандартные домены, либо домены, объявленные вами в раздели domains. Объявление домена аргумента и описание типа аргумента – суть одно и то же. Имена аргументов *Optional Name1* будут игнорироваться компилятором.

4 Раздел domains – это тот, в котором вы объявляете любые нестандартные домены, используемые вами для аргументов ваших предикатов. Домены в Прологе являются аналогами типов в других языках. Основными стандартными доменами Visual Prolog являются char, byte, short, ushort, word, integer, unsigned, long, ulong, dword, real, string и symbol. Основная форма объявления доменов имеет следующий вид:

```
domains
my Domain1,..., my DomainN   = <standard Domain>
```

Форма объявления составных доменов имеет следующий вид:

my Domain1,..., my DomainN= <compound Domain_1>;
<compound Domain_2>;
< ... >;
<compound Domain_M>

5 Раздел goal – в котором вы задаете внутреннюю цель своей программы; это позволяет программе быть скомпилированной, запускаться и выполняться независимо от среды визуальной разработки (VDE).

6 Арность (размерность) предиката – это число принимаемых им аргументов; два предиката с одним именем могут иметь различную арность. Предикаты с различными арностями должны собираться вместе, причем и в разделе predicates и в разделе clauses; однако предикаты с различной арностью рассматриваются как абсолютно разные.

7 Правила имеют форму:

HEAD :- <Subgoal1>, <Subgoal2>, ...,<SubgoalN>.

Для разрешения правила Пролог должен разрешить все его подцели, создав при этом соответствующее множество связанных переменных. Если же одна из подцелей ложна, Пролог возвратится назад и просмотрит альтернативные решения предыдущих подцелей, а затем вновь пойдет вперед, но с другими значениями переменных. Этот процесс называется поиск с возвратом.

8 Символ Пролога :- (if) отличается от "IF", используемых в других языках: правило Пролога работает в соответствии с условной формой тогда/если, тогда как оператор других языков IF работает в соответствии с условной формой если/тогда.

8.4 Сопоставление и унификация

Рассмотрим программу ch04e01.pro с точки зрения того, как утилита Test Goal будет отыскивать все решения следующей цели:

written_by (X, Y).

Пытаясь выполнить целевое утверждение *written_by (X, Y),* Visual Prolog должен проверить каждое предложение written_by в программе. Сопоставляя аргументы X и Y с аргументами каждого предложения written_by, Visual Prolog выполняет поиск от начала программы до ее конца. Обнаружив предложение, соответствующее целевому утверждению, Visual Prolog присваивает значения свободным переменным таким образом, что целевое утверждение и предложение становятся идентичными; говорят, что целевое утверждение

унифицируется с предложением. Такая операция сопоставления называется *унификацией*.

Листинг 10. Программа ch04e01.pro

```
domains
title, author = symbol
pages       = unsigned
predicates
  book (title, pages)
  written_by (author, title)
  long_novel (title)
clauses
  written_by (fleming, "DR NO").
  written_by (melville, "MOBY DICK").
book ("MOBY DICK", 250).
book ("DR NO", 310).
  long_novel (Title):-
    written_by (_, Title),
    book (Title, Length),
    Length > 300.
```

Поскольку X и Y являются свободными переменными в целевом утверждении, а свободная переменная может быть унифицирована с любым другим аргументом (даже с другой свободной переменной), целевое утверждение может быть унифицировано с первым предложением written_by в программе, как показано ниже:

```
written_by (  X  ,  Y  ).
            |     |
written_by (fleming, "DR NO").
```

Visual Prolog устанавливает соответствие, X становится связанным с fleming, а Y – с "DR NO".

В этот момент Visual Prolog напечатает

```
X = fleming, Y = "DR NO".
```

Поскольку Test Goal ищет все решения для заданной цели, целевое утверждение также будет унифицировано и со вторым предложением written_by:

```
written_by (melville, "MOBY DICK").
```

Test Goal печатает второе решение:

```
X = melville, Y = MOBY DICK
2 Solutions
```

Теперь предположим, что вы задали программе целевое утверждение

written_by (X, "MOBY DICK").

Visual Prolog произведет сопоставление с первым предложением written_by:

written_by (X ,"MOBY DICK").

| |

written_by (fleming,"DR NO").

Так как MOBY DICK и "DR NO" не соответствуют друг другу, попытка унификации завершается неудачно. Затем Visual Prolog проверяет следующий факт в программе:

written_by (melville, "MOBY DICK").

Этот факт действительно унифицируется, и X становится связанным с melville.

Рассмотрим, как Visual Prolog выполнит следующее целевое утверждение:

long_novel (X).

Когда Visual Prolog пытается выполнить целевое утверждение, он проверяет, действительно ли обращение может соответствовать факту или заголовку правила. В нашем случае устанавливается соответствие с высказыванием

long_novel (Title).

Visual Prolog проверяет предложение для long_novel, пытаясь завершить сопоставление унификацией аргументов. Поскольку в целевом утверждении X не связан, свободная переменная X может быть унифицирована с любым другим аргументом. Title также не является связанным в заголовке предложения long_novel. Целевое утверждение соответствует заголовку правила, и унификация выполняется.

Впоследствии Visual Prolog будет пытаться согласовывать подцели с правилом:

long_novel (Title):-
written_by (_, Title),
book (Title, Length),
Length > 300.

Пытаясь выполнить согласование тела правила, Visual Prolog обратится к первой подцели в теле правила, written_by (_, Title). Заметим, что, поскольку авторство книги является несущественным, анонимная переменная (_) появляется на месте аргумента author. Обращение written_by (_, Title) становится текущей подцелью, и Пролог ищет решение для этого обращения.

Пролог ищет соответствие с данной подцелью от вершины и до конца программы. В результате достигается унификация с первым фактом для written_by, а именно:

written_by (_ , Title),
| |
written_by (fleming,"DR NO").

Переменная Title связывается с DR NO, и к следующей подцели book (Title, Length) обращение выполняется уже с этим значением переменной.

Теперь Visual Prolog начинает очередной процесс поиска, пытаясь найти соответствие с обращением к book. Так как Title связан с "DR NO", фактическое обращение выглядит как book ("DR NO", Length). Процесс поиска опять-таки начинается с вершины программы. Заметим, что первая попытка сопоставить с предложением book ("MOBY DICK", 250) завершится неудачно, и Visual Prolog перейдет ко второму предложению book в поиске соответствия. Здесь заголовок книги соответствует подцели, и Visual Prolog связывает переменную Length с величиной 310.

Теперь третье предложение в теле long_novel становится текущей подцелью:

Length > 300.

Visual Prolog выполняет сравнение, завершающееся успешно; 310 больше, чем 300. В этот момент все подцели в теле правила выполнены, и следовательно, обращение long_novel (X) успешно. Так как X в обращении был унифицирован с переменной Title в правиле, то значение, с которым связывается Title при подтверждении правила, возвращается и унифицируется с переменной X. Title в случае подтверждения правила имеет значение "DR NO", поэтому Visual Prolog выведет:

X="DR NO"
1 Solution.

В следующих подразделах показано несколько прикладных примеров унификации. Однако существует еще несколько основополагающих понятий, с которыми необходимо ознакомиться прежде; это, например, сложные структуры. В следующем подразделе обсуждается, как Пролог выполняет поиск решений для этих случаев.

Поиск с возвратом

Часто при решении реальной задачи необходимо придерживаться некоторого пути для ее логического завершения. Если полученный результат не дает искомого ответа, вы должны выбрать другой путь. Так, в детстве вам, возможно, приходилось играть в лабиринт. Один из верных способов найти конец лабиринта — это поворачивать налево на каждой развилке лабиринта до тех пор, пока вы не попадете в тупик. Тогда вам следует вернуться к последней развилке и попробовать свернуть вправо, после чего опять поворачивать налево на каждом встречающемся распутье. Путем методичного перебора всевозможных путей вы в конце концов найдете верный путь.

Visual Prolog при поиске решения задачи использует именно такой метод проб и возвращений назад; этот метод называется *поиск с возвратом*. Начиная поиск решения задачи (или целевого утверждения), Visual Prolog, возможно, должен выбрать между двумя альтернативными путями. Тогда он ставит маркер у места ветвления (называемый *точкой отката*) и выбирает первую подцель, которую и станет проверять. Если данная подцель не выполнится (что эквивалентно достижению тупика в лабиринте), Visual Prolog вернется к точке отката и попробует другую подцель.

Вот простой пример (для выполнения этого примера используйте Test Goal).

Листинг 11. Программа ch04e02.pro

```
predicates
likes (symbol, symbol)
tastes (symbol, symbol)
food (symbol)
clauses
likes (bill, X):-
food (X),
tastes (X, good).
tastes (pizza, good).
tastes (brussels_sprouts, bad).
food (brussels_sprouts).
food (pizza).
```

Эта маленькая программа составлена из двух множеств фактов и одного правила. Правило, представленное отношением likes, просто утверждает, что Билл любит вкусную пищу.

Чтобы увидеть, как работает поиск с возвратом, дадим программе для решения следующее целевое утверждение:

```
likes (bill, What).
```

Когда Пролог пытается произвести согласование целевого утверждения, он начинает поиск с вершины программы.

В данном случае он начнет поиск решения, производя с вершины программы поиск соответствия с подцелью likes (bill, What).

Он обнаруживает соответствие с первым предложением в программе, и переменная What унифицируется с переменной X. Сопоставление с заголовком правила заставляет Visual Prolog попытаться удовлетворить это правило. Производя это, он двигается по телу правила и обращается к первой находящейся здесь подцели: food (X).

Замечание. Когда выполняется новое обращение, поиск соответствия для этого обращения вновь начинается с вершины программы.

Пытаясь согласовать первую подцель, Visual Prolog начинает с вершины, производя сопоставление с каждым фактом или заголовком правила, встреченным в процессе прохождения через программу.

Он обнаруживает соответствие с запросом у первого же факта, представляющего отношение food. Таким образом, переменная X связывается со значением brussels_sprouts. Поскольку существует более, чем один возможный ответ на обращение food (X), Visual Prolog помещает точку возврата (маркер) возле факта food (brussels_sprouts). Эта точка поиска с возвратом указывает на то место, откуда Пролог начнет поиск следующего возможного соответствия для food (X).

Замечание. Когда установление соответствия обращения завершается успешно, говорят, что обращение возвращается, и очередная подцель может быть испытана.

Имея *X* связанным с brussels_sprouts, следующее обращение будет выполняться как

tastes (brussels_sprouts, good)

и Visual Prolog начнет поиск, пытаясь согласовать это обращение – вновь с вершины программы. Так как соответствующих предложений не обнаруживается, обращение завершается неудачно, и Visual Prolog запускает свой механизм возврата. Начиная поиск с возвратом, Пролог отступает к последней позиции, где была поставлена точка отката. В данном случае Пролог возвращается к факту food (brussels_sprouts).

Замечание. Единственным способом освободить переменную, однажды связанную в предложении, является поиск с возвратом.

Когда Пролог отступает к точке поиска с возвратом, он освобождает все переменные, расположенные после этой точки, и намеревается найти другое решение для исходного обращения.

Обращение было food (X), так что связанность brussels_sproutsc X отменена. Теперь Пролог пытается заново произвести решение для этого обращенияю. Он обнаруживает соответствие с фактом food (pizza); на этот раз переменная X связывается со значением pizza.

Теперь Пролог переходит к следующей подцели в правиле, имея при этом новую связанную переменную. Производится новое обращение, tastes (pizza, good), и начинается поиск с вершины программы. На этот раз соответствие найдено и целевое утверждение успешно выполняется.

Поскольку переменная What в целевом утверждении унифицирована с переменной X в правиле likes, а переменная X связана со значением pizza,

переменная What отныне связана со значением pizza и Visual Prolog сообщает решение:

```
What=pizza
1 Solution.
```

Поиск всех решений в Test Goal

Как было описано выше, при помощи поиска с возвратом Visual Prolog не только найдет первое решение задачи, но при использовании режима Test Goal будет также способен найти все возможные решения.

Рассмотрим программу ch04e03.pro, которая содержит сведения об именах и возрастах нескольких игроков в теннисном клубе.

Листинг 12. Программа ch04e03.pro

```
domains
child = symbol
age   = integer
predicates
player (child, age)
clauses
player (peter,9).
player (paul,10).
player (chris,9).
player (susan,9).
```

Вам предстоит использовать Visual Prolog для того, чтобы устроить турнир по пинг-понгу между девятилетними членами теннисного клуба. Каждая пара игроков должна провести между собой две игры. Ваша задача – найти все возможные пары из тех игроков клуба, кому по девять лет. Это может быть достигнуто при помощи задания режиму Test Goal составной цели:

```
goal
player (Person1, 9),
player (Person2, 9),
Person1 <> Person2.
```

На естественном языке это прозвучало бы так: найти Person 1 в возрасте 9 лет и Person 2 в возрасте 9 лет, отличное от Person 1.

1 Visual Prolog попытается найти решение для первой подцели player (Person1, 9) и перейдет к следующей подцели только после того, как первая подцель будет достигнута. Первая подцель согласуется сопоставлением Person1 с peter. Теперь Visual Prolog может попытаться согласовать следующую подцель:

player (Person2, 9).

Опять же Person 2 сопоставляется с peter. Теперь Пролог переходит к третьей и последней подцели:

Person1 <> Person2.

2 Так как и Person1 и Person 2 связаны с peter, эта подцель не выполняется. Вследствие этого Visual Prolog выполняет поиск с возвратом к предыдущей подцели и ищет другое решение для второй подцели:

player (Person2, 9).

Эта подцель выполняется при сопоставлении Person 2 с chris.

3 Теперь третья подцель:

Person1 <>Person2

может быть выполнена, так как peter и chris отличны. Таким образом, целевое утверждение полностью согласовано путем образования турнирной пары из Криса и Питера.

4 Однако, поскольку Test Goal должен найти все возможные решения целевого утверждения, он находит точку поиска с возвратом предыдущей цели в надежде вновь добиться успеха.

Так как

player (Person2, 9)

также может быть согласовано, если принять Person 2 за susan, то Visual Prolog еще раз проверяет третью подцель. Достигается успех (так как peter отличен от susan), и другое решение для всего целевого утверждения найдено.

5 В дальнейшем поиске решений Test Goal вновь возвращается к точке поиска с возвратом второй подцели, но все возможности для этой подцели уже исчерпаны. Вследствие этого поиск с возвратом теперь выполняется от первой подцели. Она вновь может быть согласована сопоставлением Person1 с chris. Вторая подцель теперь имеет успех в результате сопоставления Person2 с peter, так что третья подцель согласована, и все целевое утверждение опять выполнено. Итак, запланирована еще одна встреча, на этот раз между Крисом и Питером.

6 В поисках еще одного решения целевого утверждения Test Goal возвращается к точке поиска с возвратом второй подцели в правиле. Здесь Person2 ставится в соответсвие chris, и при этом условии проверяется третья подцель. Она не выполняется, так как Person1 и Person2 эквивалентны, и тогда выполняется поиск с возвратом от второй подцели в поисках другого решения. Person2 сопоставляется с susan, и теперь третья подцель выполняется, вызывая к жизни очередную встречу в теннисном клубе (Крис против Сюзан).

7 И вновь, памятуя о необходимости найти все решения, Test Goal возвращается ко второй подцели, но на этот раз безуспешно. Когда вторая подцель не выполняется, процесс возвращается к первой подцели, на этот раз

находя соответствие Person1 с susan. Пытаясь выполнить вторую подцель, Test Goal сопоставляет Person 2 с peter, и впоследствии третья подцель выполнится при этих условиях. Итак, назначена пятая встреча.

8 Снова возврат ко второй подцели, где Person 2 сопоставляется с chris. Найдено шестое решение задачи о теннисном клубе, и получено полное множество турнирных пар.

9 Последнее исследуемое решение связывает с susan как Person1, так и Person 2. Так как это приводит к невыполнению последней подцели, Visual Prolog должен вернуться ко второй подцели, но там не осталось никаких новых вариантов. Тогда Test Goal возвращается для поиска к первой подцели, но все возможности для Person1 уже исчерпаны. Для данного целевого утверждения не может быть найдено других решений, и работа программы завершается.

Убедитесь, что Test Goal ответит следующим:

Person1=peter, Person2=chris
Person1=peter, Person2=susan
Person1=chris, Person2=peter
Person1=chris, Person2=susan
Person1=susan, Person2=peter
Person1=susan, Person2=chris
6 Solutions

Обратите внимание, как поиск с возвратом может вызывать вывод Test Goal избыточных решений. В нашем примере Test Goal не отметил, что Person1 = peter тоже самое, что и Person2 = peter. Далее в этой главе мы покажем вам, как управлять поиском в Visual Prolog.

Упражнение на поиск с возвратом

Используя программу ch04e04.pro, решите, что ответит Visual Prolog на следующее целевое утверждение:

player (Person1, 9), player (Person2, 10).

Проверьте ваш ответ, выполнив программу с данным целевым утверждением с помощью Test Goal.

Детальный обзор поиска с возвратом

После разбора предыдущего простого примера, вы можете теперь взглянуть более подробно на то, как работает механизм поиска с возвратом в Visual Prolog. Начнем с того, что рассмотрим программу ch04e04.pro в свете следующего целевого утверждения, состоящего из двух подцелей:

likes (X, wine) , like (X, books).

При исследовании целевого утверждения Visual Prolog отмечает, какие подцели согласовались, а какие нет.

Процесс поиска может быть представлен целевым деревом, как на рис. 8.7.

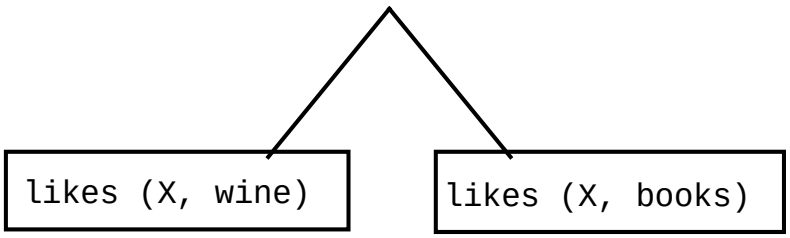


Рис. 8.7 Целевое дерево процесса поиска

Перед началом исследования целевого утверждения целевое дерево состоит из двух несогласованных подцелей. На последующих изображениях целевого дерева согласованная подцель в целевом дереве будет отмечаться подчеркиванием, а соответствующее предложение – записываться под этой подцелью.

Листинг 13. Программа ch04e04.pro

```
domains
name,thing = symbol
predicates
likes (name, thing)
reads (name)
is_inquisitive (name)
clauses
likes (john,wine).
likes (lance,skiing).
likes (lance,books).
likes (lance,films).
likes (Z, books):-
reads (Z),
is_inquisitive (Z).
reads (john).
is_inquisitive (john).
goal
likes (X,wine), likes (X, books).
```

Четыре основных принципа поиска с возвратом

В данном примере целевое дерево демонстрирует, что должны быть согласованы две подцели. Чтобы достичь этого, Visual Prolog придерживается первого фундаментального правила поиска с возвратом.

Правило. Подцели должны быть согласованы по порядку, сверху вниз.

Visual Prolog определяет, какую подцель ему использовать при попытке сопоставления предложения, исходя из второго основного правила поиска с возвратом.

Правило. Предикатные предложения проверяются в том порядке, в каком они появляются в программе, сверху вниз.

Выполняя программу ch04e04.pro, Visual Prologнаходит предложение, соответствующее первому факту, определяющему предикат likes. Посмотрите на рис. 8.8. на целевое дерево теперь.

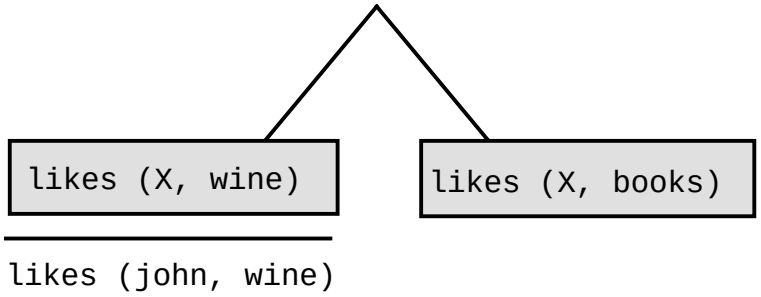


Рис. 8.8 Целевое дерево 2

Подцель likes (X, wine) соответствует факту likes (john, wine), и X связывается со значением john. Visual Prolog пытается согласовать следующую справа подцель.

Обращение ко второй подцели начинает совершенно новый поиск с условием X=john.

Первое предложение

likes (john, wine)

не соответствует подцели

likes (X, books),

так как wine (вино) – это вовсе не то же самое, что books (книги). Поэтому Visual Prolog должен проверить следующее предложение, но lance не соответствует значению X (потому что в данном случае X связан с john), так что поиск продолжается с третьим предложением, определяющим предикат likes:

likes (Z, books):- reads(Z), is_inquisitive(Z).

Аргумент Z суть переменная, поэтому она может соответствовать X. Вторые аргументы находятся в согласии, так что вызов соответствует заголовку правила. Когда X согласуется с Z, аргументы *унифицируются*. В результате унификации аргументов Visual Prolog приравняет значение, которое имеет X (то есть john), и переменную Z. В результате переменная Z теперь также имеет значение john.

Теперь подцель соответствует левой части (заголовку) правила. Продолжение поиска определяется третьим фундаментальным правилом поиска с возвратом.

Правило. Когда подцель соответствует заголовку правила, тело этого правила должно быть согласовано следующим. Тело правила теперь образует новое множество подцелей для согласования.

Это дает следующее целевое дерево, изображенное на рис. 8.9.

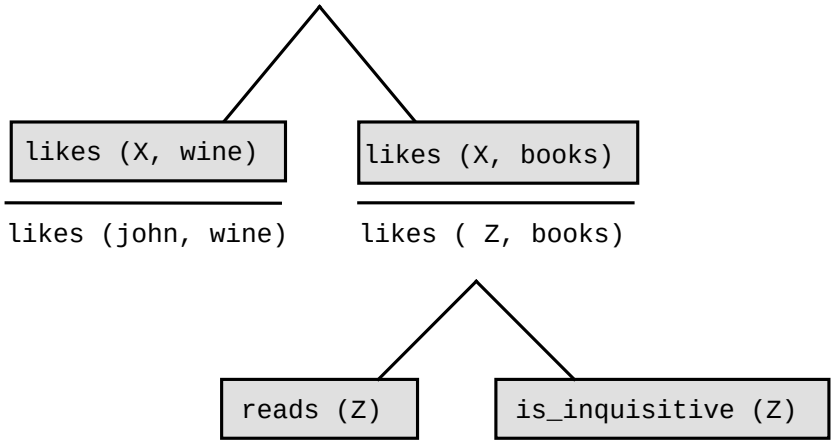


Рис. 8.9 Целевое дерево 3

Теперь целевое дерево включает в себя подцели

reads (Z) and is_inquisitive (Z),

где Z связана со значением john. Теперь Visual Prolog будет искать факты, соответствующие обеим подцелям. Последнее результирующее целевое дерево изображено на рис. 8.10.

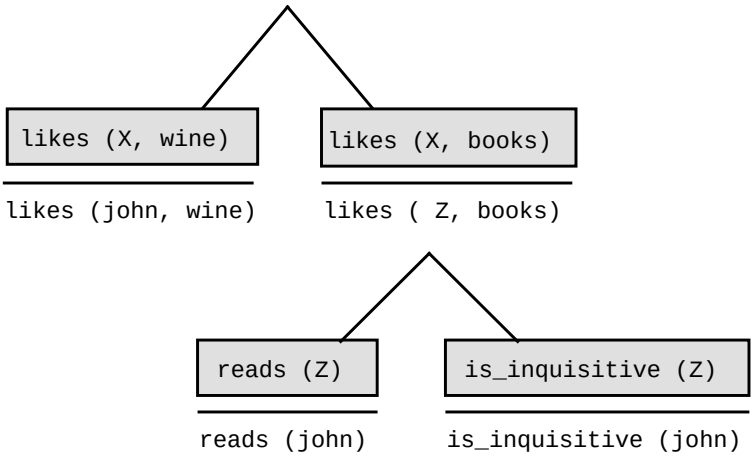


Рис. 8.10 Результирующее целевое дерево

Правило. Целевое утверждение считается согласованным, когда соответствующий факт найден для каждой оконечности (листа) целевого дерева.

Таким образом, теперь начальное целевое утверждение согласовано.

Visual Prolog использует результат процедуры поиска по-разному, в зависимости от того, как был начат поиск. Если целевое утверждение является обращением из подцели в теле правила, то Visual Prolog пытается согласовать следующую подцель в правиле, после того как обращение возвращено. Если целевое утверждение является вопросом пользователя, то Visual Prolog непосредственно отвечает:

X=john
1 Solution.

Как вы видели в программе ch04e04.pro, однажды согласовав целевое утверждение, Visual Prolog утилита TestGoal возвращается (откатывается) назад для поиска всех альтернативных решений. TestGoal также возвращается назад, если подцель не выполняется, надеясь пересогласовать предыдущую подцель таким образом, чтобы невыполненная подцель была согласована с другими предложениями.

Выполняя подцель, Visual Prolog начинает поиск с первого предложения, определяющего предикат. Затем может произойти одно из двух:

1. Он находит соответствующее предложение, тогда происходит следующее:

– если имеется другое предложение, которое, возможно, может вновь согласовать подцель, Visual Prolog выставляет указатель (с тем, чтобы отметить точку возврата) возле этого предложения;

– все свободные переменные в подцели, которые соответствуют значениям в предложении, связываются с соответствующими значениями;

– если данное предложение является заголовком правила, то затем оценивается тело этого правила; подцели в теле правила должны быть удовлетворены для успешного завершения обращения.

2. Visual Prolog не может найти соответствующее предложение, таким образом целевое утверждение не согласуется. Visual Prolog выполняет поиск с возвратом в попытке вновь согласовать предыдущую подцель. Когда процесс достигает последней точки возврата, Visual Prolog освобождает все переменные, которым были присвоены новые значения после того, как была проставлена точка возврата; затем пытается вновь согласовать исходное обращение.

Visual Prolog начинает поиск с вершины программы. Когда он выполняет возврат к обращению, новый процесс поиска начинается с точки поиска с возвратом, выставленной последней. Если поиск безуспешен, то вновь выполняется поиск с возвратом. Если процесс поиска с возвратом исчерпал все предложения для всех подцелей, то значит целевое утверждение не согласуется.

Поиск с возвратом для внутреннего целевого утверждения

Вот еще один, слегка усложненный пример, иллюстрирующий, как в Visual Prolog происходит поиск с возвратом, когда программа скомпилирована и выполняется как автономная исполняемая программа.

Листинг 14. Программа ch04e05.pro

```
predicates
type (symbol, symbol)
is_a (symbol, symbol)
lives (symbol, symbol)
can_swim (symbol)
clauses
type (ungulate,animal).
type (fish,animal).
is_a (zebra,ungulate).
is_a (herring,fish).
is_a (shark,fish).
lives (zebra,on_land).
lives (frog,on_land).
lives (frog,in_water).
lives (shark,in_water).
can_swim (Y):-
type (X,animal),
is_a (Y,X),
lives (Y,in_water).
goal
  can_swim (What),
  write ("A ",What," can swim\n"),
  readchar (_).
```

После того как программма скомпилирована и запущена (например, с использованием команды меню **Project | Run**), Visual Prolog автоматически начнет выполнение целевого утверждения, пытаясь согласовать все подцели в разделе программы goal.

10 Visual Prolog обращается к предикату can_swim со свободной переменной What. Пытаясь выполнить это обращение, Visual Prolog просматривает программу в поисках соответствия. Он обнаруживает соответствие с предложением, определяющим can_swim, и переменная What унифицируется с переменной Y.

11 Затем Visual Prolog пытается согласовать тело правила. При этом Visual Prolog обращается к первой подцели в теле правила, type (X, animal), и ищет соответствие для этого обращения. Он обнаруживает соответствие с первым фактом, определяющим отношение type.

12 В этот момент X связывается с ungulate. Поскольку здесь налицо более чем одно возможное решение, Visual Prolog проставляет точку возврата возле факта type (ungulate, animal).

13 Имея X, связанным с ungulate, Visual Prolog производит обращение ко второй подцели в правиле (is_a (Y, ungulate)) и снова ищет соответствие. Он находит его с первым фактом, is_a (zebra, ungulate). Y связывается с zebra, и Visual Prolog выставляет точку возврата около is_a (zebra, ungulate).

14 Теперь, имея X связанным с ungulate, и Y – с zebra, Visual Prolog пытается согласовать последнюю подцель, lives (zebra, in_water). Visual Prolog проверяет каждое предложение lives, но в программе нет предложения lives (zebra, in_water), поэтому обращение завершается неудачно и Visual Prolog начинает поиск с возвратом другого решения.

15 Когда Visual Prolog совершает обратный ход, процесс возвращается к последней позиции, где была помещена точка возврата. В данном случае последняя точка возврата была поставлена у второй подцели в правиле, на факте is_a (zebra, ungulate).

16 При достижении точки возврата Visual Prolog освобождает переменные, которым были присвоены новые значения после последней точки возврата, и пытается найти другое решение для обрабатываемого обращения. В данном случае обращение было is_a (Y, ungulate).

17 Visual Prolog продолжает спуск по предложениям в поиске другого соответствующего предложения, начиная с того места, где поиск был прекращен. Так как в программе больше нет соответствующих предложений, обращение завершается неудачно, и Visual Prolog вновь ведет поиск с возвратом в попытке решить исходное целевое утверждение.

18 Для теперешней позиции последней точкой возврата является type (ungulate, animal).

19 Visual Prolog освобождает переменные, использованные в исходном обращении, и пытается найти другое решение для обращения type (X, animal). Поиск начинается после точки возврата. Visual Prolog находит соответствие со следующим фактом type в программе (type (fish, animal)); X связывается с fish, и новая точка возврата ставится возле этого факта.

20 Теперь Visual Prolog продвигается вниз, к следующей подцели в правиле; поскольку это уже новое обращение, поиск начинается с вершины программы для is_a (Y, fish).

21 Visual Prolog находит соответствие для этого обращения, и Y становится связанным с herring.

22 Так как Y теперь связан с herring, следующая подцель, к которой происходит обращение, суть lives (herring, in_water). Это опять-таки новое обращение, и поиск начинается с вершины программы.

23 Visual Prolog исследует каждый факт lives, но не находит соответствия, и подцель не выполняется.

24 Теперь Visual Prolog возвращается к последней точке возврата is_a (herring, fish).

25 Переменные, которые были связаны этим сопоставлением, теперь освобождены. Начиная с того места, где процесс был прекращен, Visual Prolog теперь ищет новое решение для обращения is_a (Y, fish).

26 Visual Prolog находит соответствие со следующим предложением is_a, Y становится связанным с идентификатором shark.

27 Visual Prolog опять исследует последнюю подцель, имея переменную Y, связанную с shark. Он выполняет обращение lives (shark, in_water); поиск начинается с вершины программы, так как это уже новое обращение. Он обнаруживает соответствие, и последняя для правила подцель выполняется.

28 К этому моменту тело правила can_swim (Y) согласовано. Visual Prolog возвращает Y вызову can_swim (What). Так как What связана с Y, а Y связан с shark, отныне в целевом утверждении What связана с shark.

29 Visual Prolog продолжает процесс с того места в разделе goal, где он был остановлен, и обращается ко второй подцели в целевом утверждении.

30 Visual Prolog завершает программу выводом: **Asharkcanswim**.
и программа завершается успешно (рис. 8.11).

Попытайтесь проследить за этими шагами, используя Visual Prolog отладчик (Debugger).

Запустите отладчик из VDE с помощью команды **Project | Debug.**

Когда появится окно отладчика, выберите команду меню отладчика **View | Local Variables,** и используйте команду **Run | Trace Into** (или горячую клавишу <F7>) для пошагового выполнения программы и наблюдения за изменениями значений переменных.

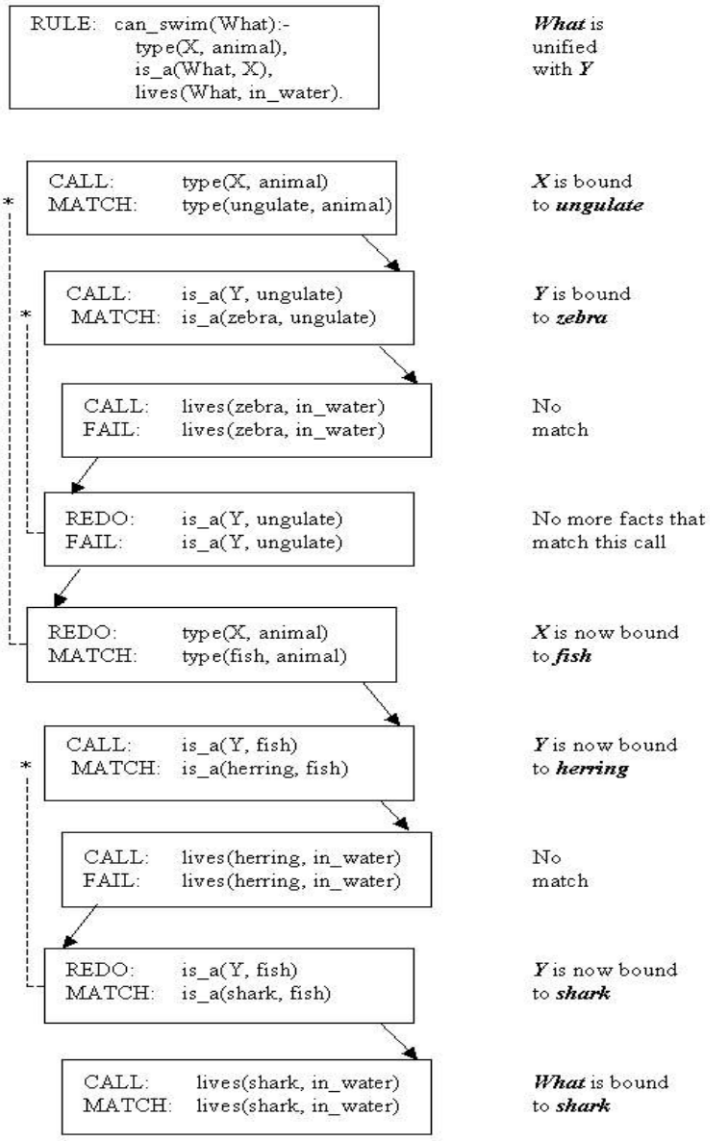


Рис. 8.11 Как работает программа can_swim

Управление поиском решений

В Прологе встроенный механизм поиска с возвратом может привести к поиску ненужных решений, в результате чего теряется эффективность. Например, когда желательно найти только единственное решение по данному вопросу. В других случаях может быть необходимым заставить Visual Prolog продолжать поиск дополнительных решений, даже если некоторое целевое утверждение уже согласовано. В подобных ситуациях необходимо управлять процессом поиска с возвратом. В этом разделе показаны некоторые методы, которые вы сможете использовать для управления поиском решений ваших целевых утверждений, выполняемым Visual Prolog.

Visual Prolog обеспечивает два инструментальных средства, которые дают возможность управлять механизмом поиска с возвратом: предикат fail, который используется для инициализации поиска с возвратом, и cut, или отсечение

(обозначается !) – используется для предотвращения (запрета) возможности возврата.

Использование предиката fail

Visual Prolog начинает поиск с возвратом, когда вызов завершается неудачно. В определенных ситуациях бывает необходимо инициализировать выполнение поиска с возвратом, чтобы найти другие решения. Visual Prolog поддерживает специальный предикат fail, вызывающий неуспешное завершение, и, следовательно, инициализирует возврат. Действие предиката fail равносильно эффекту от сравнения 2 = 3 или другой невозможной подцели. Программа ch04e06.pro иллюстрирует использование этого специального предиката.

Листинг 15. Программа ch04e06.pro

```
domains
  name = symbol
predicates
  father (name, name)
  everybody
clauses
  father (leonard,katherine).
father (carl,jason).
father (carl,marilyn).
everybody:-
    father (X,Y),
    write (X," is ",Y,"'s father\n"),
    fail.
```

Пусть некто хочет найти все решения цели father(X,Y). Используя утилиту TestGoal**,** можно записать цель как

```
goal
father (X,Y).
```

Утилита TestGoal найдет ВСЕ решения цели father(X,Y) и отобразит значения всех переменных следующим образом:

```
X=leonard, Y=katherine
  X=carl, Y=jason
  X=carl, Y=marilyn
  3 Solutions
```

Но если вы скомпилируете эту программу и запустите ее (клавишей <F9> или командой меню **Project | Run**), то Visual Prolog найдет только первое подходящее решение для father (X,Y). После того, как целевое утверждение, определенное в разделе goal, однажды полностью выполнено, ничто не говорит

Visual Prolog о необходимости продолжения поиска с возвратом. Поэтому обращение к father приведет только к одному решению и никакие переменные не будут отображены. Это, определенно, не то, что вам нужно. Однако предикат everybody в программе ch04e06.pro использует fail для поддержки поиска с возвратом, и следовательно находит все возможные решения.

Задача предиката everybody − найти ВСЕ решения для father и выдать четкий ответ в результате работы программы. Сравните предыдущие ответы утилиты Test Goal с целью father (X,Y) и ответы на выполнение следующей цели:

```
goal
everybody,
```

отображенные сгенерированной программой:

```
leonard is katherine's father
carl is jason's father
carl is marilyn's father.
```

Предикат everybody использует поиск с возвратом с тем, чтобы получить все решения для father (X, Y), заставляя Пролог выполнять поиск с возвратом сквозь тело правила everybody:

```
father (X, Y),
  write (X," is ",Y,"'s father\n"),
  fail.
```

Предикат fail никогда не может быть согласован (он всегда неуспешен), поэтому Visual Prolog вынужден выполнять поиск с возвратом. При поиске с возвратом Пролог возвращается к последнему обращению, которое может произвести множественные решения.

Такое обращениеназывают *недетерминированным*. Недетерминированное обращение является противоположностью детерминированному обращению, которое может произвести только одно решение.

Предикат write не может быть вновь согласован (он не может предложить новых решений), поэтому Visual Prolog должен выполнить откат дальше, на этот раз к первой подцели в правиле.

Обратите внимание, что помещать подцель после fail в теле правила бесполезно. Так как предикат fail все время завершается неудачно, нет никакой возможности для достижения подцели, расположенной после fail.

Упражнения

1 Загрузите и запустите программу ch04e06.pro и исследуйте следующие целевые утверждения:

```
father (X, Y).
everybody.
```

2 Измените тело правила, определяющего everybody, таким образом, чтобы правило заканчивалось предикатом write (удалите обращение к fail). Теперь скомпилируйте и запустите программу, задавая everybody в качестве цели. Почему Test Goal Visual Prolog не находит всех решений, как в случае вопроса father (X, Y)?

3 Восстановите обращение к fail в конце правила everybody. Опять задайте вопрос everybodyв качестве цели и запустите Test Goal. Почему решения для everybody прерваны no?

Прерывание поиска с возвратом: отсечение

Visual Prolog содержит в себе возможность отсечения, которая используется для прерывания поиска с возвратом; отсечение обозначается восклицательным знаком (!). Действует отсечение просто: через отсечение невозможно совершить откат (поиск с возвратом).

Отсечение помещается в программу таким же образом, как и подцель в теле правила. Когда процесс проходит через отсечение, немедленно удовлетворяется обращение к cut и выполняется обращение к очередной подцели (если таковая имеется). Однажды пройдя через отсечение, уже невозможно произвести откат к подцелям, расположенным в обрабатываемом предложении перед отсечением, и также невозможно возвратиться к другим предложениям, определяющим обрабатывающий предикат (предикат, содержащий отсечение).

Существуют два основных случая применения отсечения:

- Если вы заранее знаете, что определенные посылки никогда не приведут к осмысленным решениям, то поиск решений в этом случае будет лишней тратой времени и памяти. Если в такой ситуации вы примените отсечение, то программа станет быстрее и экономичнее. Такой прием называют *зеленым отсечением*.
- Когда отсечения требует сама логика программы для исключения из рассмотрения альтернативных подцелей. Это называют *красным отсечением*.

Как использовать отсечение

В этом разделе даются примеры, показывающие как следует использовать отсечение в ваших программах. В этих примерах используется несколько условных правил (r1, r2 и r3), которые определяют условный предикат r, плюс несколько подцелей (a, b, c и т. д.).

Предотвращение поиска с возвратом к предыдущей подцели в правиле

```
r1 :- a, b,
      !,
c.
```

Такая запись является способом сообщить Visual Prolog о том, что вас удовлетворит первое решение, найденное им для подцелей a и b. Имея возможность найти множественные решения при обращении к с путем поиска с возвратом, Visual Prolog при этом не может произвести откат (поиск с возвратом) через отсечение и найти альтернативное решение для обращений a и b. Он также не может возвратиться к другому предложению, определяющему предикат r1.

В качестве конкретного примера рассмотрим программу ch04e07.pro.

Листинг 16. Программа ch04e07.pro

```
predicates
  buy_car (symbol,symbol)
  car (symbol,symbol,integer)
  colors (symbol,symbol)
clauses
  buy_car (Model,Color):-
    car (Model,Color,Price),
    colors (Color,sexy),
    !,
    Price < 25000.
car (maserati,green,25000).
car (corvette,black,24000).
car (corvette,red,26000).
car (porsche,red,24000).
colors (red,sexy).
colors (black,mean).
colors (green,preppy).
goal
  buy_car (corvette,Y).
```

В данном примере поставлена цель найти Corvette (Корвет) приятного цвета, который можно себе позволить по стоимости. Отсечение в правиле buy_car означает, что поскольку в базе данных содержится только один «Корвет» приятного цвета, хоть и со слишком высокой ценой, то нет нужды искать другую машину.

Получив целевое утверждение

buy_car (corvette, Y)

Visual Prolog обращается к car, первой подцели для предиката buy_car, выполняет проверку для первой машины, Maserati, которая завершается неудачно. Затем он проверяет следующее предложение car и находит соответствие, связывая переменную Color со значением black.

4 Он переходит к следующему обращению и проверяет, имеет ли выбранная машина приятный цвет. Черный цвет не является приятным в данной программе, таким образом проверка завершается неудачно.

5 Visual Prolog выполняет поиск с возвратом к обращению car и снова ищет Corvette, удовлетворяющий этому критерию.

6 Он находит соответствие и снова проверяет цвет. На этот раз цвет оказывается приятным, и Visual Prolog переходит к следующей подцели в правиле: к отсечению. Отсечение немедленно выполняется, «замораживая» все переменные, ранее связанные в этом предложении.

7 Теперь Visual Prolog переходит к следующей (и последней) подцели в правиле, к сравнению

Price < 25 000.

8 Эта проверка завершается неудачно, и Visual Prolog пытается совершить поиск с возвратом с целью найти другую машину для проверки. Поскольку отсечение предотвращает поиск с возвратом, то нет способа решить последнюю подцель, и наше целевое утверждение завершается неудачно.

Предотвращение поиска с возвратом к следующему предложению

Отсечение может быть использовано как способ сообщить Visual Prolog, что он выбрал верное предложение для определенного предиката.

Например, рассмотрим следующий фрагмент:

```
r(1):-
        !,
a, b, c.
r(2):-
        !,
d.
r(3):-
        !,
c.
r(_):-
write ("This is a catchall clause.").
```

Использование отсечения делает предикат r детерминированным. В данном случае Visual Prolog выполняет обращение к r с единственным целым аргументом. Предположим, что произведено обращение r(1). Visual Prolog

просматривает программу в поисках соответствия для обращения; он находит его с первым предложением, определяющим r. Поскольку имеется более, чем одно возможное решение для данного обращения, Visual Prolog проставляет точку возврата около этого предложения. Теперь пришел черед правила, и Visual Prolog начинает обработку тела правила. Первое, что происходит, это то, что он проходит через отсечение; таким образом, исключается возможность возвращения к другому предложению r. Это отменяет точки поиска с возвратом, повышая эффективность выполнения программы. Это также гарантирует, что отлавливающее ошибки предложение будет выполнено лишь в том случае, если ни одно из условий не будет соответствовать обращению к r.

Обратите внимание, что конструкция такого типа весьма похожа на конструкцию «case» в других языках программирования. Также обратите внимание на то, что условие проверки записывается в заголовке правил.

Вы могли бы попросту написать такие предложения

```
r (X):-
X = 1,
    !,
a, b, c.
r(X):-
X = 2,
 !,
 d.
  r (X):-
    X = 3,
    !,
    c.
  r(_):-
    write ("This is a catchall clause.").
```

Однако вам следует помещать проверочное условие в заголовок правила, когда только возможно: это повышает эффективность программы и упрощает ее чтение.

В качестве другого примера рассмотрим следующую программу. Запустите ее с помощью Test Goal.

Листинг 17. Программа ch04e08.pro

```
predicates
  friend (symbol,symbol)
  girl (symbol)
  likes (symbol,symbol)
```

```
clauses
  friend (bill,jane):-
    girl (jane),
    likes (bill,jane),
    !.
friend (bill,jim):-
    likes (jim,baseball),
    !.
friend (bill,sue):-
    girl (sue).
girl (mary).
girl (jane).
girl (sue).
likes (jim,baseball).
likes (bill,sue).
goal
friend (bill, Who).
```

Если бы в программе не было отсечения, то Visual Prolog предложил бы два решения: Билл является другом как Джейн, так и Сью. Однако отсечение в первом предложении, определяющем friend, говорит Visual Prolog, что если это предложение согласовано, то друг Билла уже найден, и нет нужды продолжать поиск других кандидатур. В сущности, отсечение такого рода говорит, что вы удовлетворены найденным решением, и нет смысла продолжать поиск другого друга.

Поиск с возвратом может иметь место внутри предложений в попытке согласовать обращение, но, однажды обнаружив решение, Visual Prolog проходит через отсечение. Предложения friend, записанные так, как показано выше, возвратят одного и только одного друга Билла (если друг вообще может быть найден).

Детерминизм и отсечение

Если бы предикат friend, определенный в предыдущей программе, не содержал отсечений, то это был бы недетерминированный предикат (предикат, способный произвести множественные решения при помощи поиска с возвратом). Во многих реализациях Пролога программисты должны быть особенно внимательными с недетерминированными предложениями из-за сопутствующих им дополнительных требований к ресурсам памяти во время работы программы. Однако Visual Prolog выполняет проверку на недетерминированные предложения, облегчая бремя программиста.

В Visual Prolog существует директива компилятора check_determ. Если вставить эту директиву в самое начало программы, то Visual Prolog будет

выдавать предупреждение в случае обнаружения недетерминированных предложений в процессе компиляции.

Вы можете превратить недетерминированные предложения в детерминированные, вставляя отсечения в тело правил, определяющих данный предикат. Например, помещение отсечений в предложения, определяющие предикат friend, делает этот предикат детерминированным, поскольку с этими отсечениями обращение к friend может возвратить одно и только одно решение.

Предикат not

Следующая программа демонстрирует, как вы можете ипользовать предикат not для того, чтобы выявить успевающего студента: студента, у которого средний балл (GPA) не менее 3.5 и у которого в настоящее время не продолжается испытательный срок.

Листинг 18. Программа ch04e10.pro

```
domains
  name = symbol
  gpa  = real
predicates
  honor_student (name)
  student (name, gpa)
  probation (name)
clauses
  honor_student (Name):-
    student (Name, GPA),
    GPA > =3.5,
    not (probation(Name)).
student ("Betty Blue", 3.5).
student ("David Smith", 2.0).
student ("John Johnson", 3.7).
probation ("Betty Blue").
probation ("David Smith").
goal
  honor_student (X).
```

При использовании **not** необходимо иметь в виду один момент: *предикат* **not** *будет успешным, если не может быть доказана истинность данной подцели*. Это приводит к предотвращению связывания внутри **not** несвязанных переменных. При вызове изнутри **not** подцели со свободными переменными, Visual Prologвозвратит сообщение об ошибке: "Free variables not allowed in "not"

or "retractall" (свободные переменные не разрешены в **not** или **retract**). Это происходит вследствие того, что для связывания Прологом свободных переменных в подцели, эта подцель должна унифицироваться с каким-либо другим предложением и эта подцель должна выполняться. Правильным способом управления несвязанными переменными внутри **not** подцели является использование анонимных переменных.

Вот несколько примеров корректных и некорректных предложений:

```
likes (bill, Anyone):-      % 'Anyone' – выходной аргумент
likes (sue, Anyone),
not (hates(bill, Anyone).
```

В этом примере Anyone связывается посредством likes (sue, Anyone) до того, как Visual Prolog делает вывод о том, что hates(bill, Anyone)не является истиной. Данное предложение работает как положено.

Если вы измените его таким образом, что обращение к **not** будет выполняться первым, то получите ошибку о том, что "free variable are not allowed in **not**" (свободные переменные в **not** не разрешены).

```
likes (bill, Anyone):-      % Это не будет работать правильно
not (hates (bill, Anyone)),
likes (sue, Anyone).
```

Даже если вы исправите это, заменив в not (hates (bill, Anyone)) Anyone на анонимную переменную, и предложение, таким образом, не будет возвращать ошибку, все равно это будет давать неправильный результат.

```
likes (bill, Anyone):-      % Это не будет работать правильно
not (hates (bill, _)),
likes (sue, Anyone).
```

Это предложение утверждает, что Биллу нравится кто угодно, если неизвестно ничего о том, кого Билл ненавидит, и если этот «кто-то» нравится Сью. Подлинное предложение утверждало, что Биллу нравится тот, кто нравится Сью, и при этом Билл не испытывает к этому человеку ненависти.

Пример. Каждый раз дважды подумайте перед тем, как использовать предикат **not**. Неверное использование приведет к сообщению об ошибке или к ошибкам в логике вашей программы. Следующая программа является примером правильного использования предиката **not**.

Листинг 19. Программа ch04e11.pro

```
predicates
  likes_shopping (symbol)
  has_credit_card (symbol,symbol)
  bottomed_out (symbol,symbol)
```

```
clauses
  likes_shopping (Who):-
    has_credit_card  (Who,Card),
    not (bottomed_out (Who,Card)),
    write (Who," can shop with the ",Card, " credit card.\n").
  has_credit_card (chris,visa).
  has_credit_card (chris,diners).
  has_credit_card (joe,shell).
  has_credit_card (sam,mastercard).
  has_credit_card (sam,citibank).
  bottomed_out (chris,diners).
  bottomed_out (sam,mastercard).
  bottomed_ou (chris,visa).
goal
  likes_shopping (Who).
```

Упражнения

1. Предположим, что средний налогоплательщик в США – это женатый человек, имеющий двоих детей, который зарабатывает не менее 500 и не более 2000 долларов в месяц. Определите предикат special_taxpayer, который, при целевом утверждении special_taxpayer (fred), выполнится лишь в том случае, если fred нарушит одно из условий для среднего налогоплательщика. Используйте отсечение для гарантии того, что не выполняется ненужный поиск с возвратом.

2. Игроки в некотором теннисном клубе разбиты на три лиги, и могут вызывать на состязание только членов своей лиги, или же состоящих лигой ниже (если таковая имеется).

Напишите на Visual Prolog программу, которая будет печатать все возможные пары между игроками клуба в следующей форме:

tom versus bill

marjory versus Annette.

Используйте отсечение, чтобы, например:

tomversusbill

и

billversustom

не выводились одновременно.

3. Следующая программа – это упражнение на поиск с возвратом, а не проверка вашей способности раскрывать таинственные убийства. Загрузите и запустите эту программу с помощью Test Goal.

Замечание. Виновен Берт, так как он имеет мотив и испачкан тем же веществом, что и жертва.

Листинг 20. Программа ch04e12.pro

```
                    domains
  name,sex,occupation,object,vice,substance = symbol
  age=integer
                    predicates
  person (name, age, sex, occupation)
  had_affair (name, name)
  killed_with (name, object)
  killed (name)
  killer (name)
  motive (vice)
  smeared_in (name, substance)
  owns (name, object)
  operates_identically (object, object)
  owns_probably (name, object)
  suspect (name)
                    /* * * Факты об убийстве* * */
CLAUSES
person (bert,55,m,carpenter).
person (allan,25,m,football_player).
person (allan,25,m,butcher).
person (john,25,m,pickpocket).
                    had_affair (barbara,john).
  had_affair (barbara,bert).
  had_affair (susan,john).
                    killed_with (susan,club).
killed (susan).
                    motive (money).
motive (jealousy).
motive (righteousness).
                    smeared_in (bert, blood).
  smeared_in (susan, blood).
  smeared_in (allan, mud).
  smeared_in (john, chocolate).
  smeared_in (barbara,chocolate).
```

```
                    owns (bert,wooden_leg).
owns (john,pistol).
                    /* * * Подоплека * * */
  operates_identically (wooden_leg, club).
  operates_identically (bar, club).
  operates_identically (pair_of_scissors, knife).
  operates_identically (football_boot, club).
                        owns_probably (X,football_boot):-
    person (X,_,_,football_player).
  owns_probably (X,pair_of_scissors):-
    person (X,_,_,hairdresser).
  owns_probably (X,Object):-
    owns (X,Object).
                    /* * * * * * * * * * * * * * * * * * * * * * *
 * Подозреваем всех тех, кто имеет оружие,   *
 * которым могла быть убита Сюзан.*
 * * * * * * * * * * * * * * * * * * * * * * */
  suspect (X):-
    killed_with (susan,Weapon) ,
    operates_identically (Object,Weapon) ,
    owns_probably (X,Object).
                    /* * * * * * * * * * * * * * * * * * * * * * * * * *
 * Подозреваеммужчин, которыеимелисвязисСюзан.*
 * * * * * * * * * * * * * * * * * * * * * * * * * */
  suspect (X):-
motive (jealousy),
    person (X,_,m,_),
    had_affair (susan,X).
                    /* * * * * * * * * * * * * * * * * * * * *
 * Подозреваем женщин, которые имели связи*
 * с кем-либо из тех, кого знала Сюзан.*
 * * * * * * * * * * * * * * * * * * * * */
  suspect (X):-
    motive (jealousy),
    person (X,_,f,_),
    had_affair (X,Man),
    had_affair (susan,Man).
                    /* * * * * * * * * * * * * * * * * * * * * * * * * * *
*
 * Подозрительные преступники, мотив которых – деньги.*
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * */
  suspect (X):-
    motive (money),
    person (X,_,_,pickpocket).
                          killer (Killer):-
    person (Killer,_,_,_),
    killed (Killed),
    Killed <> Killer,     % Это не самоубийство
    suspect (Killer),
    smeared_in (Killer,Goo),
    smeared_in (Killed,Goo).
                        goal
killer (X).
```

Пролог с процедурной точки зрения

Пролог – это декларативный язык, то есть вы описываете задачу в терминах фактов и правил, и предоставляете компьютеру самому искать способ решения. Другие языки программирования, такие как Паскаль, Бэйсик и С – процедурные. Это означает, что вы должны писать подпрограммы и функции, которые подробно объяснят компьютеру какие шаги должны быть сделаны для решения задачи.

Сейчас мы собираемся оглянуться назад и рассмотреть некоторые материалы, которые вы изучили, но уже представив их с точки зрения процедурного программирования.

Как представлять факты и правила в качестве процедур

Можно рассматривать правила Пролога как определения процедур. Например, правило:

```
likes (bill,Something):-
likes (cindy,Something)
```

означает, что «Для того чтобы доказать, что Билл любит что-то, необходимо доказать, что Синди любит это».

Имея это в виду, вы можете увидеть, что предикаты типа

```
say_hello:-
write ("Hello"), nl
```

и

```
  greet:-
  write ("Hello, Earthlings!"),
  nl
```

соответствуют подпрограммам и функциям в других языках программирования.

Вы можете рассматривать даже факты Пролога, как процедуры; например, факт

likes (bill, pasta)

означает: «Для того чтобы доказать, что Билл любит pasta, не нужно ничего делать, и если аргументы Whoи What в вашем запросе likes (Who, What) – свободные переменные, то вы можете присвоить им значения bill и pasta, соответственно.»

Вы могли встречаться в других языках с некоторыми процедурами программирования, такими как условное ветвление, булевы выражения, безусловныепереходы и возвращение результата вычисления. В следующих разделах, для того чтобы повторить то что вы уже знаете с процедурной точки зрения, показано как Пролог может представлять те же самые функции.

Использование правил для условного ветвления (case)

Одно из основных различий между правилами в Прологе и процедурами в других языках программирования заключается в том, что Пролог позволяет задавать множество альтернативных определений одной и той же процедуры. Человек может быть предком, будучи отцом или будучи матерью, поэтому определение предка состоит из двух правил.

Вы можете использовать множество определений так же, как вы используете предложение case в Паскале, задавая множество альтернативных определений для каждого значения аргумента (или множества значений аргумента). Пролог будет перебирать одно правило за другим пока он не найдет правило, которое подходит, и затем выполнит действие, которое задается правилом, как в программе ch04e13.pro.

Листинг 21. Программа ch04e13.pro

```
predicates
  action (integer)
clauses
  action (1):-
    nl,
    write ("You typed 1."),nl.
action(2):-
    nl,
    write ("You typed two."),nl.
action (3):-
```

```
    nl,
    write ("Three was what you typed."),nl.
  action (N):-
    nl,
N<>1, N<>2, N<>3,
    write ("I don't know that number!"),nl.
GOAL
write ("Type a number from 1 to 3: "),
  reading (Choice),
  action (Choice).
```

Если пользователь нажмет 1, 2 или 3, action будет вызвана с соответствующим значением аргумента и будет вызвано одно из первых трех правил.

Выполнение проверки в правиле

Посмотрите более внимательно на четвертое правило для action. Оно будет сопоставлено для любого аргумента, переданного правилу. Но вы хотите быть уверенными, что оно не напечатает «I don't know that number (я не знаю такого числа)», если число попадает в правильный диапазон. Это задача для подцелей

X<>1, X<>2, X<>3,

где <> обозначает «не равно». Для того, чтобы напечатать «Я не знаю такого числа», Пролог должен сначала доказать, что X не равен 1, 2 или 3. Если все из этих подцелей будут неуспешны, то Пролог попытается использовать последнюю альтернативу. Но здесь не указано последней альтернативы и поэтому конец предложения никогда не выполнится.

Обратите внимание, что action подразумевает, что Choice уже связана. Если вы вызываете action со свободной переменной в качестве аргумента, то компилятор сгенерирует ошибку.

Отсечение как GoTo

Программа ch04e13.pro не очень хороша из-за того, что после выбора и выполнения нужного правила Пролог все еще продолжает поиск альтернатив.

Вы могли бы сэкономить память и время, если бы могли сообщить Прологу где нужно прекратить поиск альтернатив. И вы можете это сделать, используя отсечение, что означает: «Если вы дошли до этого места, то не нужно производить откаты внутри этого правила и не нужно проверять остальные альтернативы этого правила».

Другими словами, «сжигаем за собой мосты». Возвращение все еще возможно, но только на более высоком уровне. Если текущее правило

вызывается другими правилами, и высшие правила имеют альтернативы, то они могут быть испробованы. Но отсечение отбрасывает альтернативы внутри правила и альтернативы данного правила (предиката).

Используя отсечение, эта программа может быть переписана следующим образом:

Листинг 22. Программа ch04e14.pro

```
predicates
  action (integer)
clauses
  action (1):-!,
    nl,
    write ("You typed 1.").
action (2):-!,
    nl,
    write ("You typed two.").
action (3):-!,
    nl,
    write ("Three was what you typed.").
action (_):-
    write ("I don't know that number!").
GOAL
write ("Type a number from 1 to 3: "),
  reading (Num),
  action (Num),nl.
```

Отсечение не производит эффект во время непосредственного выполнения. Так например, для того, чтобы выполнить отсечение, Пролог должен войти в правило, содержащее отсечение и достичь точки, где расположено отсечение.

Отсечение может быть представлено другими примерами, как этот:

```
action (X) :-
 X>3,
 !,
 write ("Too high.").
```

В этом правиле отсечение не произведет никакого действия, пока не будет достигнута первая подцель X>3.

Заметьте, что порядок правил здесь имеет значение. В ch04e13.pro вы могли написать правила в любом порядке; только одно из них сопоставлялось с конкретным числом. Но в примере ch04e14.pro вы должны быть уверены, что компьютер не сделает попытки выполнить правило, печатающее «Я не знаю

такого числа», раньше чем все предыдущие правила будут испробованы (и не выполнят своих отсечений).

Отсечения в ch04e14.pro некоторые называют *красными отсечениями,* т. к. они меняют логику программы. Если вы сохраните проверки X<>1, X<>2, и X<>3, изменив программу только вставкой отсечений в каждом предложении, то вы сделаете *зеленое отсечение*, которые экономят время и тем не менее оставляют программу такой же правильной, как и без отсечений. Выигрыш при этом не так велик, но риск внести ошибку в программу уменьшается.

Отсечение – это мощный, но и опасный оператор Пролога. В этом отношении он соответствует предложению GoTo в остальных языках программирования – вы можете делать с ним многие вещи, но это делает вашу программу более трудной для понимания.

Возврат вычисленного значения

Как мы уже видели, правила и факты Пролога могут возвращать информацию в цель, которая вызывает их. Это делается путем связывания переменных, которые были ранее не связанными. Факт

likes (bill, cindy)

возвращает информацию в цель

likes (bill, Who)

путем присваивания переменной Who значения cindy.

Правило может возвращать тем же способом результат вычислений.

Здесь приведен простой пример:

Листинг 23. Программа ch04e15.pro

```
predicates
  classify (integer,symbol)
clauses
  classify (0,zero).
classify (X,negative):-
    X < 0.
classify (X,positive):-
X> 0.
```

Первый аргумент classyfy должен всегда получать константу или связанную переменную. Второй аргумент может быть связанной или свободной переменной, он сопоставляется с символами zero, negative, pozitive в зависимости от значения первого аргумента.

Здесь приведены несколько примеров правил которые могут возвращать значения:

1. Вы можете узнать (используя Test Goal) положительно ли 45, спросив:

Goalclassify (45, positive).

yes.

Так как 45 больше 0, только третье предложение classify может быть успешным. Для того, чтобы сделать это, оно сопоставляет второй аргумент с positive. Но второй аргумент уже равен positive, поэтому сопоставление успешно, и вы получаете ответ Yes (да).

2. Наоборот, если сопоставление неуспешно, вы получаете ответ no (нет):

Goalclassify (45, negative).

no.

Что происходит при этом: Пролог проверяет первое предложение, но первый аргумент не равен 0 (а также второй не равен zero); затем он проверяет второе предложение, связав X с 45, но проверка X<0 не успешна; после этого он проверяет третье предложение, но на этот раз второй аргумент не совпадает.

3. Для получения правильного ответа, а не yes или no, вы должны вызвать classify со свободным вторым аргументом.

Goal classify (45, What).

What=positive

1 Solution.

Что реально происходит в этом случае: цель classify (45, What) не сопоставляется с заголовком первого предложения, так как 45 не сопоставляется с 0. Первый класс использовать нельзя; цель classify (45, What) сопоставляется с заголовком следующего предложения, classify (X, negative), связывая X с 45 и negative с What. Но затем подцель X<0 неуспешна, так как X равен 45 и неверно, что 45<0, поэтому Пролог возвращается из этого предложения, освобождая созданные связи; наконец, classify (45, What) сопоставляется с classify (X, positive), связывая X и 45 и What и positive, подцель X>0 правильна. Так как это успешное решение, Пролог не выполняет поиск с возвратом; он возвращается в вызывающую процедуру (которая в данном случае цель, которую вы задали). И так как переменная X принадлежит к вызывающей процедуре, эта процедура может использовать ее значение – в данном случае автоматически напечатать ее значение.

Замечания

В этом разделе мы обсудили унификацию, поиск с возвратом, детерминизм, предикаты not, fail, и cut (!), мы рассмотрели предикаты Visual Prolog с процедурной точки зрения.

1 Факты и правила Пролога получают информацию при вызове с аргументами, которые могут быть константами или связанными

переменными; они возвращают информацию в вызывающую процедуру путем связывания аргументов, которые являются несвязанными переменными.

2 Унификация – это процесс сопоставления двух предикатов и присваивания свободным переменным значений для того, чтобы сделать предикаты идентичными. Этот механизм необходим, чтобы Пролог мог определить какое предложение вызвать и каким переменным присвоить значения.

3 В этом разделе представлены важные моменты, связанные с сопоставлением (унификацией):

– когда Пролог начинает попытки достичь цель, он начинает поиск с начала программы;

– когда вызов завершается успехом, говорят что вызов возвратился, и делается попытка доказать следующую подцель;

– если переменная была связана в предложении, единственный способ сделать ее снова свободной – это откат (поиск с возвратом).

4. Поиск с возвратом – это механизм, который указывает Прологу, где искать решения для программы. Этот процесс дает Прологу возможность перебрать все известные факты и правила для решения. В этом разделе даны четыре основных принципа поиска с возвратом:

– подцели должны проверяться по порядку, сверху вниз;

– предикатные предложения проверяются в том порядке, в котором они появляются в программе, сверху вниз;

– когда подцель сопоставляется с заголовком правила, тело этого правила должно после этого быть доказано. Тело правила состоит из новых подцелей, которые должны быть доказаны;

– цель доказана, когда соответствующие факты найдены для каждой листьевой вершины дерева целей.

5. Вызов, который может дать множество решений – недетерминированный, тогда как вызов, дающий одно и только одно решение – детерминированный.

6. Visual Prolog дает три средства для управления направлением логического поиска в программе: это два предиката **fail** и **not**, и предикат **cut.**

– предикат **fail** всегда дает неуспех, он вызывает поиск с возвратом для того, чтобы искать другое решение;

– предикат **not** дает успех, когда связанная с ним подцель не может быть доказана;

– **cut** отменяет поиск с возвратом.

7. Очень легко представлять правила Пролога как определения процедур. С процедурной точки зрения, правила могут действовать как предложения case,

представляя собой булевы функции, действовать как оператор GoTo, (при использовании **cut**), и возвращать вычисленные значения.

9 ПОСТРОЕНИЕ НЕЧЕТКИХ ПРОДУКЦИОННЫХ СИСТЕМ В СИСТЕМЕ MATLAB

9.1 Общие сведения о программе Matlab

Matlab (сокращение от англ. «Matrix Laboratory») – пакет прикладных программ для решения задач технических вычислений и одноименный язык программирования, используемый в этом пакете. Matlab используют более миллиона инженерных и научных работников, он работает на большинстве современных операционных систем, включая Linux, Mac OS и Microsoft Windows.

Зарождение системы Matlab относится к концу 70-х годов, когда первая версия этой системы была использована в Университете Нью-Мексикои Станфордском университете для преподавания курсов теории матриц, линейной алгебры и численного анализа. Язык программирования Matlab был разработан Кливом Моулером (англ. Cleve Moler) в конце 1970-х годов, когда он был деканом факультета компьютерных наук в Университете Нью-Мексико.

Matlab – это интерактивная система, основным объектом которой является массив, для которого не требуется указывать размерность явно. Это позволяет решать многие вычислительные задачи, связанные с векторно-матричными формулировками, существенно сокращая время, которое понадобилось бы для программирования на скалярных языках типа C или FORTRAN.

Для Matlab имеется возможность создавать специальные наборы инструментов (англ. toolbox), расширяющих его функциональность. Наборы инструментов представляют собой коллекции функций, написанных на языке Matlab для решения определенного класса задач.

Компания Mathworks поставляет наборы инструментов, которые используются во многих областях, включая следующие:

1. Цифровая обработка сигналов, изображений и данных: DSP Toolbox, Image Processing Toolbox, Wavelet Toolbox, Communication Toolbox, Filter Design Toolbox – наборы функций, позволяющих решать широкий спектр задач обработки сигналов, изображений, проектирования цифровых фильтров и систем связи.

2. Системы управления: Control Systems Toolbox, μ-Analysis and Synthesis Toolbox, Robust Control Toolbox, System Identification Toolbox, LMI Control Toolbox, Model Predictive Control Toolbox, Model-Based Calibration Toolbox – наборы функций, облегчающих анализ и синтез динамических систем, проектирование, моделирование и идентификацию систем управления, включая современные алгоритмы управления, такие как робастное управление, H∞-управление, ЛМН-синтез, μ-синтез и другие.

3. Финансовый анализ: GARCH Toolbox, Fixed-Income Toolbox, Financial Time Series Toolbox, Financial Derivatives Toolbox, Financial Toolbox, Datafeed Toolbox – наборы функций, позволяющие быстро и эффективно собирать, обрабатывать и передавать различную финансовую информацию.

4. Анализ и синтез географических карт, включая трехмерные: Mapping Toolbox.

5. Сбор и анализ экспериментальных данных: Data Acquisition Toolbox, Image Acquisition Toolbox, Instrument Control Toolbox, Link for Code Composer Studio – наборы функций, позволяющих сохранять и обрабатывать данные, полученные в ходе экспериментов, в том числе в реальном времени. Поддерживается широкий спектр научного и инженерного измерительного оборудования.

6. Визуализация и представление данных: Virtual Reality Toolbox – позволяет создавать интерактивные миры и визуализировать научную информацию с помощью технологий виртуальной реальности и языка VRML.

7. Средства разработки: Matlab Builder for COM, Matlab Builder for Excel, Matlab Builder for NET, Matlab Compiler, Filter Design HDL Coder – наборы функций, позволяющих создавать независимые приложения из среды Matlab.

8. Взаимодействие с внешними программными продуктами: Matlab Report Generator, Excel Link, Database Toolbox, Matlab Web Server, Link for ModelSim – наборы функций, позволяющие сохранять данные вразличных видов таким образом, чтобы другие программы могли с ними работать.

9. Базы данных: Database Toolbox – инструменты работы с базами данных.

10. Научные и математические пакеты: Bioinformatics Toolbox, Curve Fitting Toolbox, Fixed-Point Toolbox, Genetic Algorithm and Direct Search Toolbox, OPC Toolbox, Optimization Toolbox, Partial Differential Equation Toolbox, Spline Toolbox, Statistic Toolbox, RF Toolbox – наборы специализированных математических функций, позволяющие решать широкий спектр научных и инженерных задач, включая разработку генетических алгоритмов, решения задач в частных производных, целочисленные проблемы, оптимизацию систем и другие.

11. Нейронные сети: Neural Network Toolbox – инструменты для синтеза и анализ нейронных сетей.

12. Нечеткая логика: Fuzzy Logic Toolbox – инструменты для построения и анализа нечетких множеств.

13. Символьные вычисления: Symbolic Math Toolbox – инструменты для символьных вычислений с возможностью взаимодействия с символьным процессором программы Maple.

Помимо вышеперечисленных, существуют тысячи других наборов инструментов для Matlab, написанных другими компаниями и энтузиастами.

9.2 Основные возможности пакета Fuzzy Logic Toolbox

1. Пакет **Fuzzy Logic Toolbox** предлагает инструменты для проектирования систем нечеткого логического вывода. Он позволяет создавать системы нечеткого логического вывода и нечеткой классификации в рамках среды MatLab, с возможностью их интегрирования в Simulink. Базовым понятием Fuzzy Logic Toolbox является FIS-структура – система нечеткого вывода (Fuzzy Inference System). FIS-структура содержит все необходимые данные для реализации функционального отображения «входы-выходы» на основе нечеткого логического вывода согласно схеме, приведенной на рис. 9.1 [25].

2. Fuzzy Logic Toolbox содержит следующие категории программных инструментов:

- функции;
- интерактивные модули с графическим пользовательским интерфейсом (с GUI);
- блоки для пакета Simulink;
- демонстрационные примеры.

3. Модуль fuzzy позволяет строить нечеткие системы двух типов – Мамдани и Сугэно. В системах типа Мамдани база знаний состоит из правил вида «Если x_1 = низкий и x_2 = средний, то y = высокий». В системах типа Сугэно база знаний состоит из правил вида «Если x_1 = низкий и x_2 = средний, то $y = a_0+a_1x_1+a_2x_2$».

Таким образом, основное отличие между системами Мамдани и Сугэно заключается в разных способах задания значений выходной переменной в правилах, образующих базу знаний.

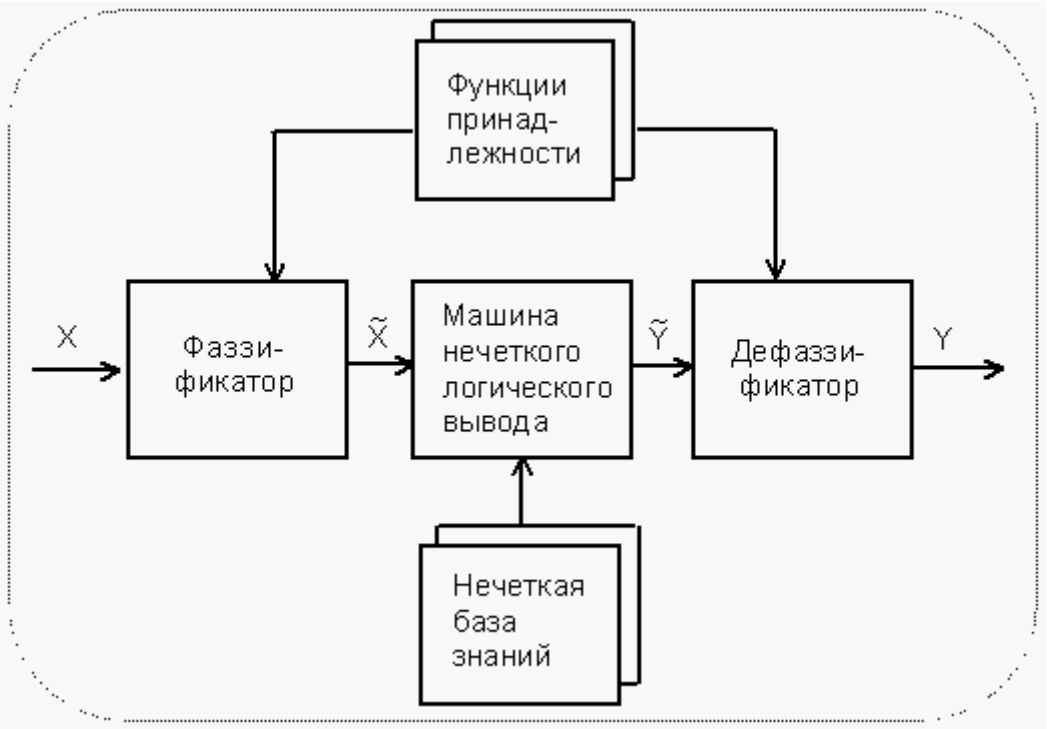


Рис. 9.1 Нечеткий логический вывод

Обозначения:

X – входной четкий вектор;

$\tilde{X}$– вектор нечетких множеств, соответствующий входному вектору X;

$\tilde{Y}$– результат логического вывода в виде вектора нечетких множеств;

Y – выходной четкий вектор.

В системах типа Мамдани значения выходной переменной задаются нечеткими термами, в системах типа Сугэно – как линейная комбинация входных переменных.

4. Для создания системы нечеткого логического вывода с помощью пакета Fuzzy Logic Toolbox программы MatLab нажмите кнопку «**Start**» и пройдите последовательно по вкладкам **Toolboxes**, далее **Fuzzy Logic**, далее **FIS Ediutor GUI (fuzzy)**, как показано на рис. 9.2.

5. Система нечеткого логического вывода представляется в рабочей области MatLab в виде структуры данных, изображенной на рис. 9.3. Существует два способа загрузки FIS Editor в рабочую область: считывание с диска с помощью функции **readfis**; передача из основного **fis-редактора** путем выбора в меню File подменю Export и команды **To workspace**.

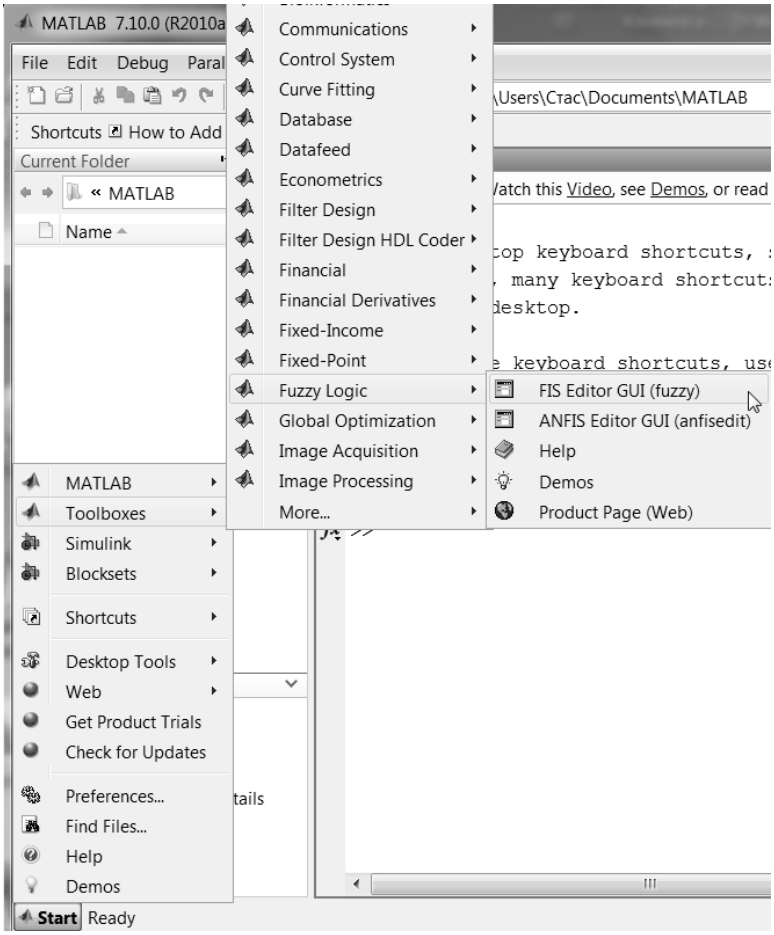


Рис. 9.2 Запуск пакета Fuzzy Logic Toolbox программы MatLab

6. Поля структуры данных системы нечеткого логического вывода (рис. 9.3) предназначены для хранения следующей информации:

1) **name** – наименование системы нечеткого логического вывода;
2) **type** – тип системы. Допустимые значения **'Mamdani'** и **'Sugeno'**;
3) **andMethod** – реализация логической операции И.

Запрограммированные реализации: **'min'** – минимум и **'prod'** – умножение;

4) **orMethod** – реализация логической операции ИЛИ. Запрограммированные реализации: **'max'** – максимум и **'probor'** – вероятностное ИЛИ;
5) **defuzzMethod** – метод дефаззификации.

Запрограммированные методы для систем типа Мамдани:

- **'centroid'** – центр тяжести;
- **'bisector'** – медиана;
- **'lom'** – наибольший из максимумов;
- **'som'** – наименьший из максимумов;
- **'mom'** – среднее из максимумов.

Запрограммированные методы для систем типа Сугэно:

- **'wtaver'** – взвешенное среднее;
- **'wtsum'** – взвешенная сумма;

6) **impMethod** – реализация операции импликации. Запрограммированные реализации: **'min'** – минимум и **'prod'** – умножение;
7) **aggMethod** – реализация операции объединения функций принадлежности выходной переменной. Запрограммированные реализации: **'max'** – максимум; **'sum'** – сумма и **'probor'** – вероятностное ИЛИ;
8) **input** –массив входных переменных;
   input.name – наименование входной переменной;

   input.range – диапазон изменения входной переменной;

   input.mf – массив функций принадлежности входной переменной;

   input.mf.name – наименование функции принадлежности входной переменной;

   input.mf.type – модель функции принадлежности входной переменной.

Запрограммированные модели:

- **dsigmf** – функция принадлежности в виде разности между двумя сигмоидными функциями;
- **gauss2mf** – двухсторонняя гауссовская функция принадлежности;
- **gaussmf** – гауссовская функция принадлежности;
- **gbellmf** – обобщенная колокообразная функция принадлежности;
- **pimf** – пи-подобная функция принадлежности;
- **psigmf** – произведение двух сигмоидных функций принадлежности;
- **sigmf** – сигмоидная функция принадлежности;
- **smf** – s-подобная функция принадлежности;
- **trapmf** – трапециевидная функция принадлежности;
- **trimf** – треугольная функция принадлежности;
- **zmf** – z-подобная функция принадлежности;
- **input.mf.params** – массив параметров функции принадлежности входной переменной;

9) **output** – массив выходных переменных;
   output.name – наименование выходной переменной;

   output.range – диапазон изменения выходной переменной;

output.mf – массив функций принадлежности выходной переменной;

output.mf.name – наименование функции принадлежности выходной переменной;

output.mf.type – модель функции принадлежности выходной переменной.

Запрограммированные модели для системы **типа Мамдани**:

- **dsigmf** – функция принадлежности в виде разности между двумя сигмоидными функциями;
- **gauss2mf** – двухсторонняя гауссовская функция принадлежности;
- **gaussmf** – гауссовская функция принадлежности;
- **gbellmf** – обобщенная колокообразная функция принадлежности;
- **pimf** – пи-подобная функция принадлежности;
- **psigmf** – произведение двух сигмоидных функций принадлежности.

Запрограммированные модели для системы **типаСугэно**:

- **constatnt** – константа (функция принадлежности в виде синглтона);
- **linear** – линейная комбинация входных переменных;
- **output.mf.params** – массив параметров функции принадлежности выходной переменной;

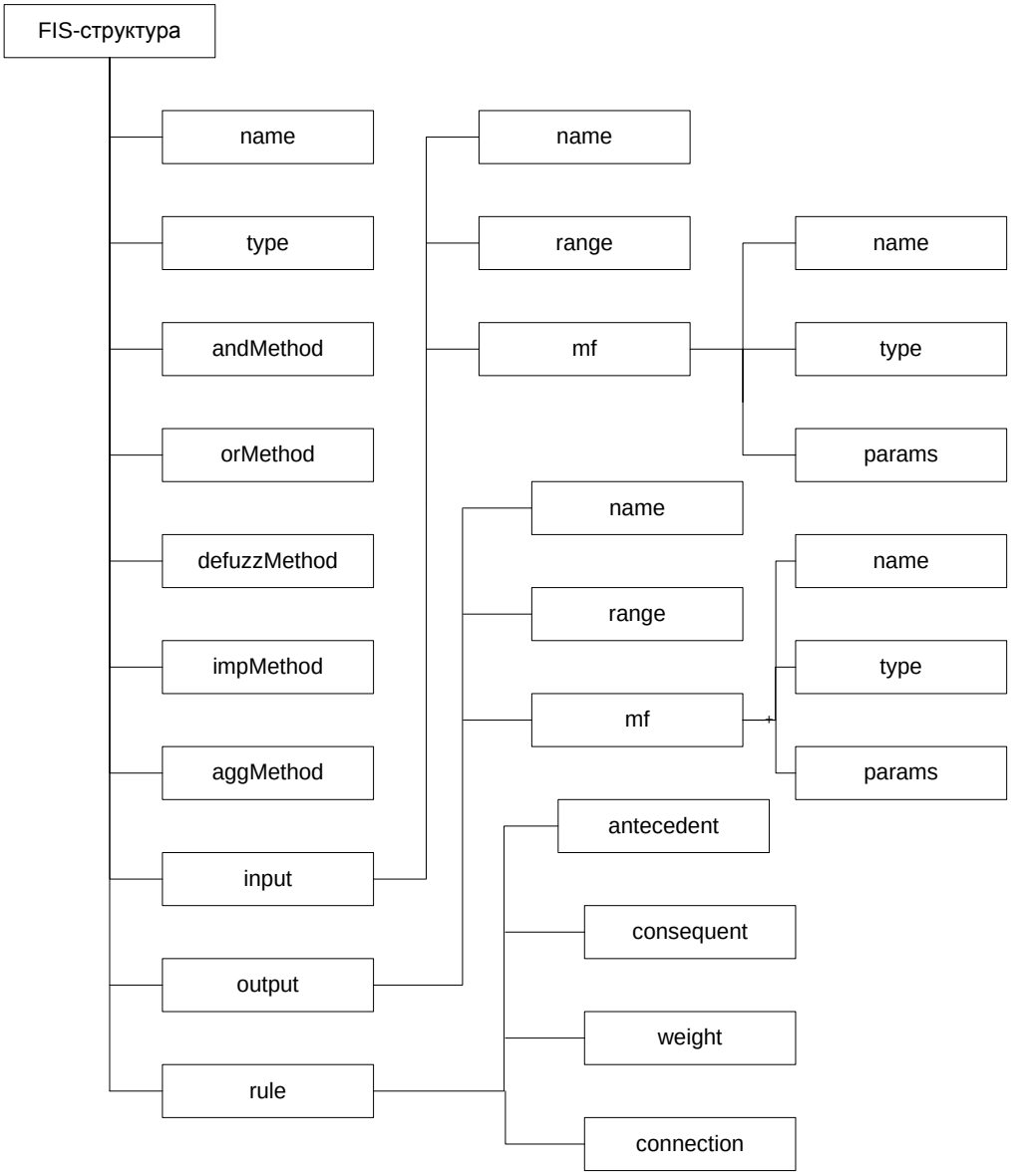


Рис. 9.3 Структура системы нечеткого логического вывода

10) **rule** – массив правил нечеткой базы знаний;

rule.antecedent – посылки правила. Указываются порядковые номера термов в порядке записи входных переменных. Число 0 указывает на то, что значение соответствующей входной переменной не влияет на истинность правила;

rule.consequent – заключения правила. Указываются порядковые номера термов в порядке записи выходных переменных. Число 0 указывает на то, что правило не распространяется на соответствующую выходную переменную;

rule.weight – вес правила. Задается числом из диапазона [0, 1];

rule.connection – логическая связка переменных внутри правила: 1 – логическое И; 2 – логическое ИЛИ.

9.3 Инструменты пакета Fuzzy Logic Toolbox по созданию систем нечеткого вывода

Имеются пять инструментов GUI для создания, редактирования и наблюдения систем нечеткого вывода в среде Fuzzy Logic Toolbox:

1 **Fuzzy Inference System Editor** или **FIS Editor** – редактор системы нечеткого логического вывода.
2 **Member ship Function Editor** – редактор функций принадлежности.
3 **Rule Editor** – редактор правил.
4 **Rule Viewer** – модуль просмотра правил.
5 **Surface Viewer** – модуль просмотра поверхности.

Эти инструменты GUI динамически связаны между собой: используя один из них в FIS и производя изменения, можно затем увидеть их действия в других инструментах GUI.

В дополнение к этим пяти инструментам Fuzzy Logic Toolbox включает в себя графический редактор GUI ANFIS, который используется для создания и анализа Sugeno-типа адаптивных нейросистем с нечеткой логикой – adaptive neuro-FIS (ANFIS). Все пять инструментов GUI могут взаимодействовать и обмениваться информацией между собой. Любой из них может считывать и сохранять данные в рабочем пространстве Matlab и на жестком диске компьютера. Рассмотрим эти инструменты более подробно.

9.3.1 Редактор системы нечеткого логического вывода

1 Редактор системы нечеткого логического вывода (FIS-редактор) предназначен для создания, сохранения, загрузки и вывода на печать систем нечеткого логического вывода, а также для редактирования следующих свойств: тип системы, наименование системы, количество входных и выходных переменных, наименование входных и выходных переменных, параметры нечеткого логического вывода.

2 Загрузка FIS-редактора происходит из меню **Fuzzy Logic** (см. рис. 9.2). В результате появляется интерактивное графическое окно, приведенное на рис. 9.4. На этом же рисунке также указаны функциональные назначения основных полей графичекого окна. В нижней части графического окна FIS-редактора расположены кнопки **Help** и **Close**, которые позволяют вызвать окно справки и закрыть редактор, соответственно.

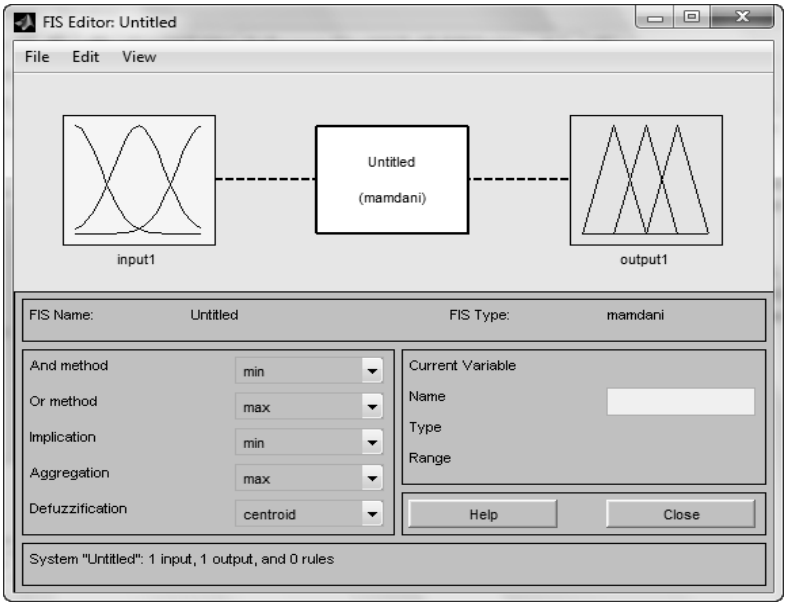


Рис. 9.4 FIS-редактор

3 FIS-редактор содержит 8 меню. Это три общесистемных меню – **File, Edit, View**, и пять меню для выбора параметров нечеткого логического вывода – **And Method, Or Method, Implication, Aggregation и Defuzzification**. (В этом и последующих подразделах приводятся примеры использования программы Fuzzy Logic Toolbox установленной в среде операционной системы Windows 7 64bit).

4 **Меню File** – это общее меню для всех GUI-модулей используемых с системами нечеткого логического вывода. Общий вид меню показан на рис. 9.5.

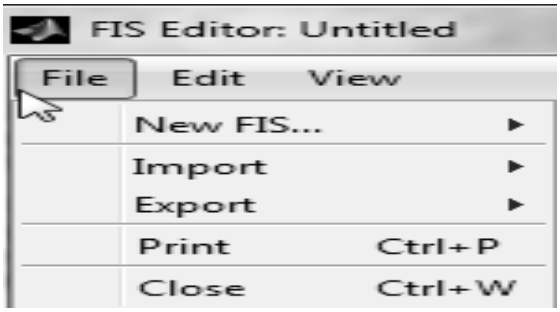


Рис. 9.5 Меню File

С помощью команды **New FIS…** пользователь имеет возможность создать новую систему нечеткого логического вывода. При выборе этой команды появятся две альтернативы: Mamdani и Sugeno, которые определяют тип создаваемой системы. Начать создавать новую систему нечеткого вывода типа Mamdani путем загрузки нового FIS-редактора можно нажатием кнопок **Ctrl+N**.

С помощью команды **Import** пользователь имеет возможность загрузить ранее созданную систему нечеткого логического вывода. При выборе этой

команды появятся две альтернативы **From Workspace…** и **From disk**, которые позволяют загрузить систему нечеткого логического вывода из рабочей области MatLab и с диска, соответственно. При выборе команды **From Workspace…** появится диалоговое окно, в котором необходимо указать идентификатор системы нечеткого логического вывода, находящейся в рабочей области MatLab. При выборе команды **From disk** появится диалоговое окно (рис. 9.6), в котором необходимо указать имя файла системы нечеткого логического вывода. Файлы систем нечеткого логического вывода имеют расширение **.fis**.

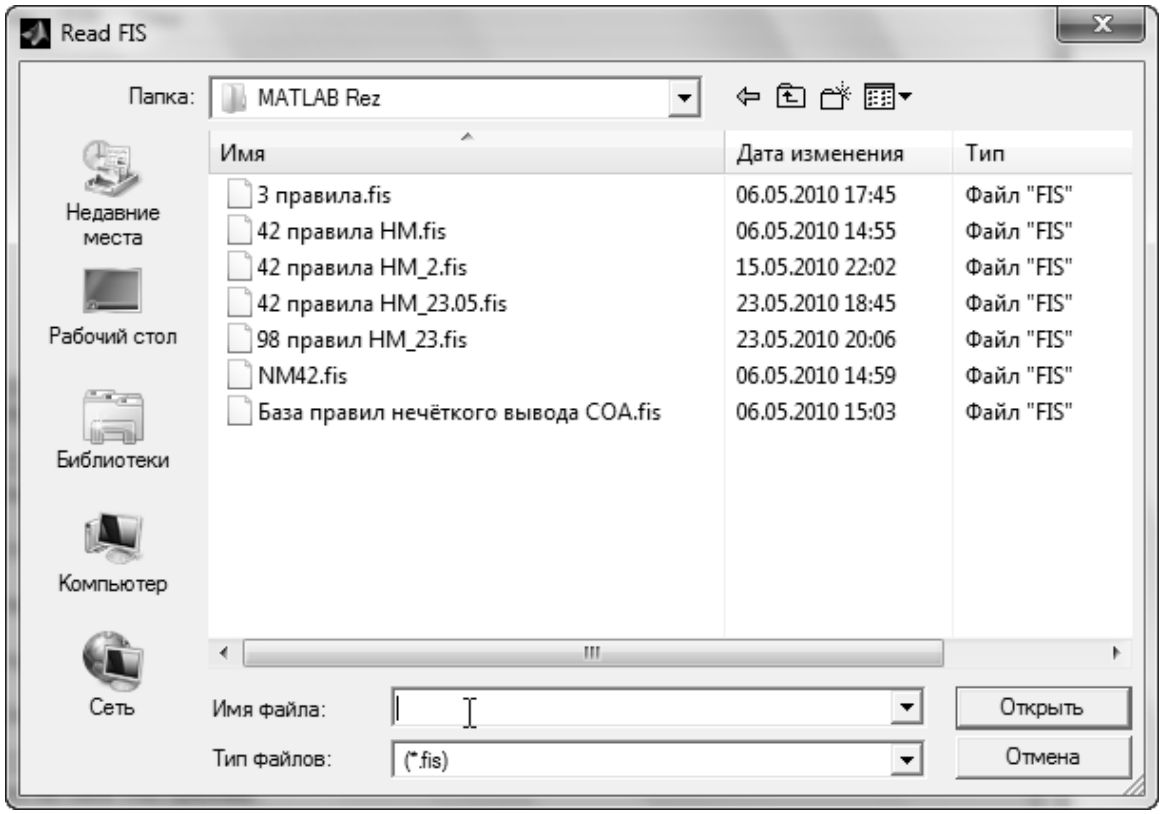


Рис. 9.6 Окно загрузки системы нечеткого логического вывода с диска

При выборе команды **Export** появятся две альтернативы **To Workspace…** и **To disk**, которые позволяют скопировать систему нечеткого логического вывода в рабочую область MatLab и на диск, соответственно. При выборе команды **To Workspace…** появится диалоговое окно, в котором необходимо указать идентификатор системы нечеткого логического вывода, под которым она будет сохранена в рабочей области MatLab. При выборе команды **To disk** появится диалоговое окно, в котором необходимо указать имя файла системы нечеткого логического вывода. Скопировать систему нечеткого логического вывода в рабочую область и на диск можно также нажатием клавиш **Ctrl+S**.

Команда **Print** позволяет вывести на принтер копию графического окна. Печать возможна также по нажатию **Ctrl+P**.

Команда **Close** закрывает графическое окно. Закрытия графического окна происходит по нажатию **Ctrl+W** или однократного щелчка левой кнопки мыши по кнопке **Close**.

5 **Меню Edit.** Общий вид меню приведен на рис. 9.7.

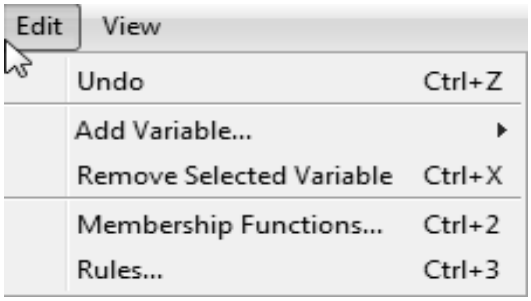


Рис. 9.7 Меню Edit

Команда **Undo** отменяет ранее совершенное действие. Выполняется также по нажатию **Ctrl+Z**.

Команда **Add Variable…** позволяет добавить в систему нечеткого логического вывода еще одну переменную. При выборе этой команды появятся две альтернативы **Input** и **Output**, которые позволяют добавить входную и выходную переменную, соответственно.

Команда **Remove Selected Variable** удаляет текущую переменную из системы. Признаком текущей переменной является красная окантовка ее прямоугольника. Назначение текущей переменной происходит с помощью однократного щелчка левой кнопки мыши по ее прямоугольнику. Удалить текущую переменную можно также с помощью нажатия **Ctrl+X**.

Команда **Membership Function…** открывает редактор функций принадлежностей. Эта команда может быть также выполнена нажатием **Ctrl+2**.

Команда **Rules…** открывает редактор базы знаний. Эта команда может быть также выполнена нажатием **Ctrl+3**.

6 **Меню View.** Это общее меню для всех GUI-модулей, используемых с системами нечеткого логического вывода. Общий вид меню показан на рис. 9.8. Это меню позволяет открыть окно визуализауии нечеткого логического вывода (команда **Rules** или нажатие клавиш **Ctrl+5**) и окно вывода поверхности «входы-выход», соответствующей системе нечеткого логического вывода (команда **Surface** или нажатие клавиш **Ctrl+6**).

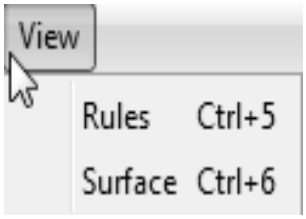


Рис. 9.8 Меню View

Остальные меню для выбора параметров нечеткого логического вывода заполняются в соответствии с подразделом 9.2.6 (см. рис. 9.3).

9.3.2 Редактор функций принадлежности

Редактор функций принадлежности (Member ship Function Editor) предназначен для задания следующей информации о терм-множествах входных и выходных переменных [25]:

- количество термов;
- наименования термов;
- тип и параметры функций принадлежности, которые необходимы для представления лингвистических термов в виде нечетких множеств.

Редактор функций принадлежности может быть вызван из любого GUI-модуля, используемого с системами нечеткого логического вывода, командой **Membership Functions…** меню **Edit** или нажатием клавиш **Ctrl+2**. В FIS-редакторе открыть редактор функций принадлежности можно также двойным щелчком левой кнопкой мыши по полю входной или выходной переменных. Общий вид редактора функций принадлежности с указанием функционального назначения основных полей графичекого окна приведен на рис. 9.9. В нижней части графического окна расположены кнопки Help и Close, которые позволяют вызвать окно справки и закрыть редактор, соответственно.

Редактор функций принадлежности содержит четыре меню – File, Edit, View, Type и четыре окна ввода информации – Range, Display Range, Name и Params. Эти четыре окно предназначенны для задания диапазона изменения текущей переменной, диапазона вывода функций принадлежности, наименования текущего лингвистического терма и параметров его функции принадлежности, соответственно. Параметры функции принадлежности можно подбирать и в графическом режиме, путем изменения формы функции принадлежности с помощью технологии «Drug and drop». Для этого необходимо позиционировать курсор мыши на знаке режима «Drug and drop» (см. рис. 9.9), нажать на левую кнопку мыши иизменять форму функции принадлежности. Параметры функции принадлежности будут пересчитываться автоматически.

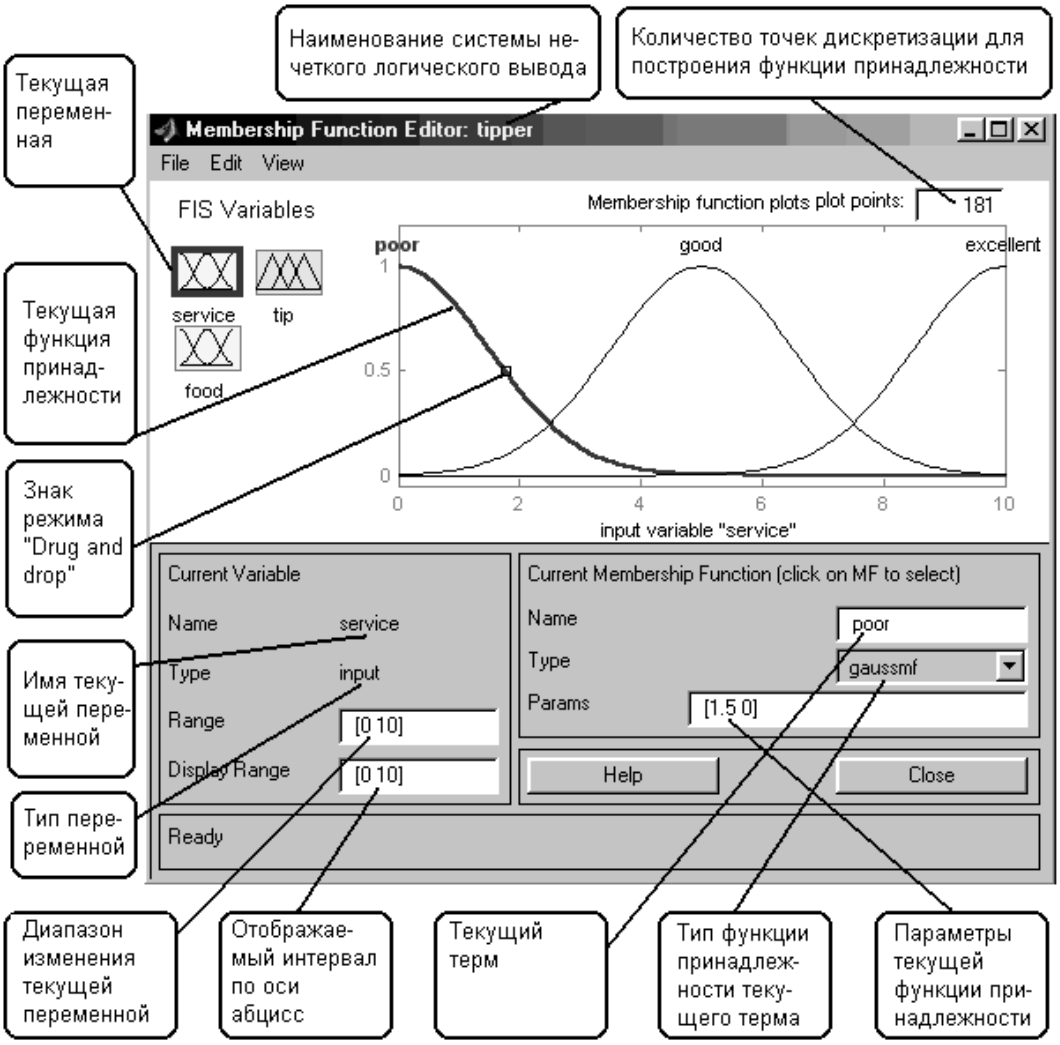


Рис. 9.9 Редактор функций принадлежности

Меню **File** и **View** одинаковые для всех GUI-модулей используемых с системами нечеткого логического вывода. Они подробно описаны в подразделе 9.3.1.

Общий вид меню **Edit** приведен на рис. 9.10.

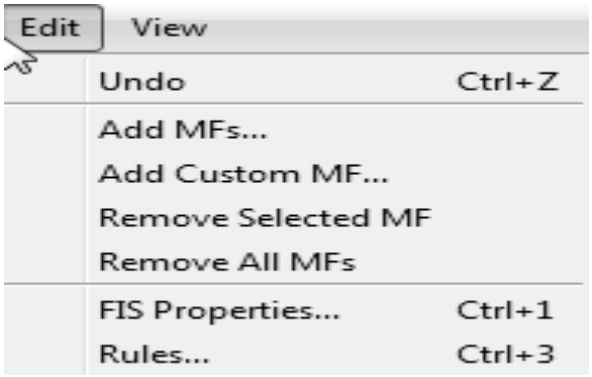


Рис. 9.10 Меню Edit

Команда **Undo** отменяет ранее совершенное действие. Выполняется также по нажатию Ctrl+Z.

Команда **Add MFs…** позволяет добавить термы в терм-множество, используемое для лингвистической оценки текущей переменной. При выборе этой команды появится диалоговое окно (рис. 9.11), в котором необходимо выбрать тип функции принадлежности и количество термов. Значения параметров функций принадлежности будут установлены автоматически таким образом, чтобы равномерно покрыть область определения переменной, заданной в окне Range. При изменении области определения в окне Range параметры функций принадлежности будут промасштабированы.

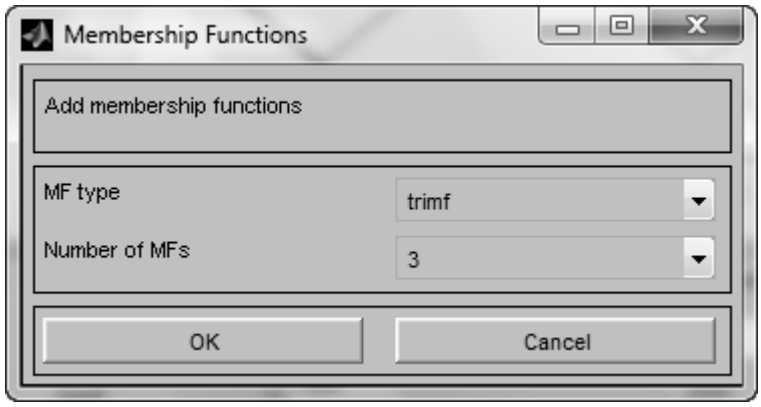


Рис. 9.11 Выбор количества термов
и типа функций принадлежности

Команда **Add Custom MF…** позволяет добавить один лингвистический терм. После выбора этой команды появляется графическое окно (рис. 9.12), в котором необходимо напечатать лингвистически терм (поле MF name), имя функции принадлежности (поле M-File function name) и параметры функции принадлежности (поле Parameter list).

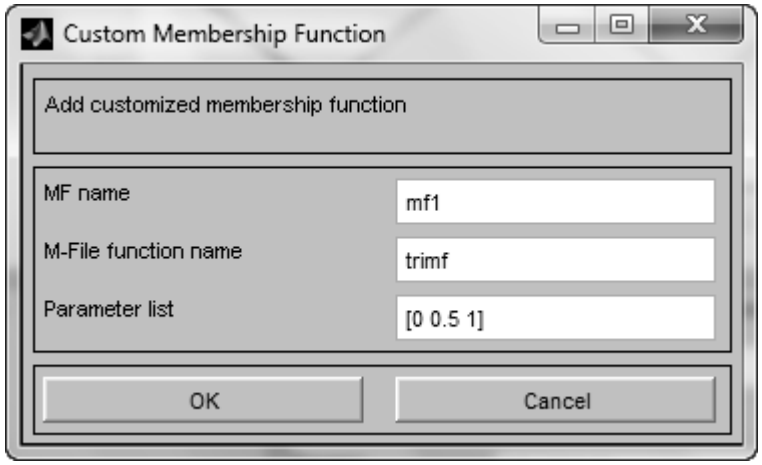


Рис. 9.12 Задание лингвистического
терма с невстроенной ФП

Команда **Remove Selected MF** удаляет текущий терм из терм-множества текущей переменной. Признаком текущей переменной является красная

окантовка ее прямоугольника. Признаком текущего терма является красный цвет его функции принадлежности. Для выбора текущего терма необходимо провести позиционирования курсора мыши на графике функции принадлежности и сделать щелчок левой кнопкой мыши.

Команда **Remove All MFs** удаляет все термы из терм-множества текущей переменной.

Команда **FIS Properties…** открывает FIS-редактор. Эта команда может быть также выполнена нажатием **Ctrl+1**.

Команда **Rules…** открывает редактор базы знаний. Эта команда может быть также выполнена нажатием **Ctrl+3**.

Меню **Type** позволяет установить тип функций принадлежности термов, используемых для лингвистической оценки текущей переменной. На рис. 9.13 приведено меню Type, в котором указаны возможные типы функций принадлежности.

Рис. 9.13 Меню Type

9.3.3 Редактор базы знаний

Редактор базы знаний (**Rule Editor**) предназначен для формирования и модификации нечетких правил. Редактор базы знаний может быть вызван из любого GUI-модуля, используемого с системами нечеткого логического вывода, командой **Rules…** меню Edit или нажатием клавиш **Ctrl+3**. В FIS-редакторе открыть редактор базы знаний можно также двойным щелчком левой кнопкой мыши по прямоугольнику с названием системы нечеткого логического вывода, расположенного в центре графического окна [25].

Общий вид редактора базы знаний с указанием функционального назначения основных полей графического окна приведен на рис. 9.14. В нижней части графического окна расположены кнопки Help и Close, которые позволяют вызвать окно справки и закрыть редактор, соответственно.

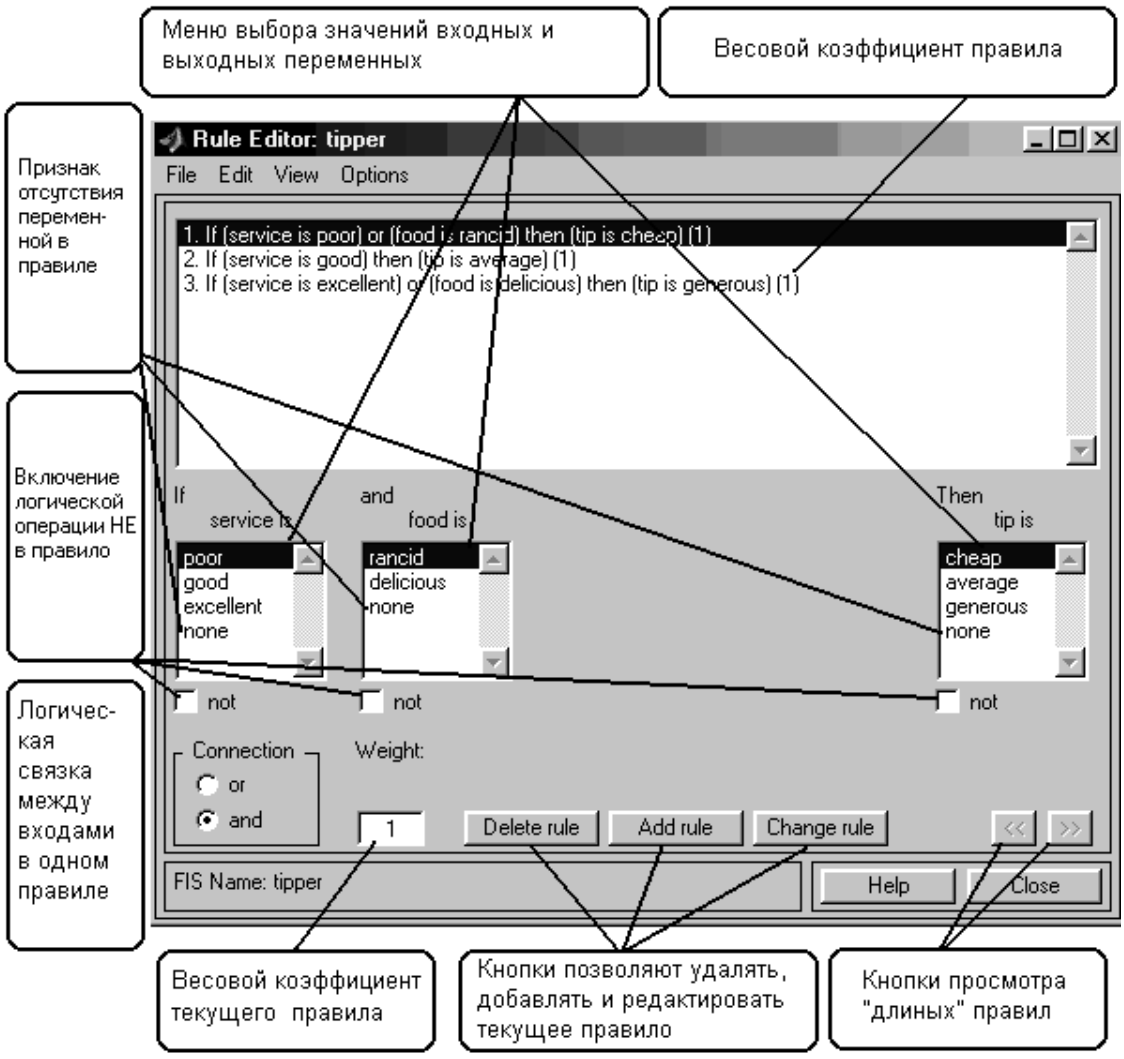


Рис. 9.14 Редактор базы знаний

Редактор базы знаний содержит четыре системных меню **File, Edit, View, Options**, меню выбора термов входных и выходных переменных, поля установки логических операций И, ИЛИ, НЕ и весов правил, а также кнопки редактирования и просмотра правил.

Для ввода нового правила в базу знаний необходимо с помощью мыши выбрать соответствующую комбинацию лингвистических термов входных и выходных переменных, установить тип логической связки (И или ИЛИ) между переменными внутри правила, установить наличие или отсутствие логической операции НЕ для каждой лингвистической переменной, ввести значение весового коэффициента правила и нажать кнопку **Add Rule**. По умолчанию установлены следующие параметры:

- логическая связка переменных внутри правила – И;
- логическая операция НЕ – отсутствует;

– значение весового коэффициента правила – 1.

Возможны случаи, когда истинность правила не изменяется при произвольной значении некоторой входной переменной, т.е. это переменная не влияет на результат нечеткого логического вывода в данной области факторного пространства. Тогда в качестве лингвистического значения этой переменной необходимо установить none.

Для удаления правила из базы знаний необходимо сделать однократный щелчок левой кнопкой мыши по этому правилу и нажать кнопку Delete Rule.

Для модификации правила необходимо сделать однократный щелчок левой кнопкой мыши по этому правилу, затем установить необходимые параметры правила и нажать кнопку **Edit Rule**.

Общий вид меню **Edit** приведен на рис. 9.15.

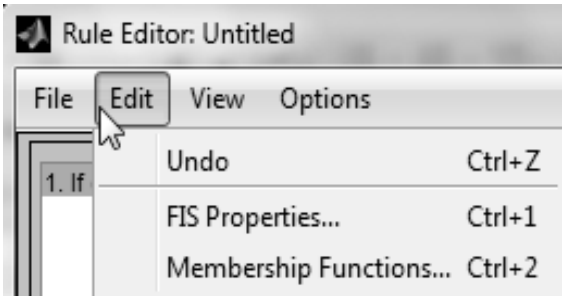


Рис. 9.15 Меню Edit

Команда **Undo** отменяет ранее совершенное действие. Выполняется также по нажатию Ctrl+Z. Команда **FIS Properties…** открывает FIS-редактор. Эта команда может быть также выполнена нажатием Ctrl+1. Команда **Membership Function…** открывает редактор функций принадлежностей. Эта команда может быть также выполнена нажатием **Ctrl+2**.

Меню **Options** позволяет установить язык и формат правил базы знаний (рис. 9.16). При выборе команды **Language** появится список языков English (Английский), Deutsch (Немецкий), Francais (Французский), из которого необходимо выбрать один.

Рис. 9.16 Меню Option

При выборе команды **Format** появится список возможных форматов правил базы знаний: Verbose – лингвистический; Symbolic – логический; Indexed – индексированный.

9.3.4 Просмотр правил

Модуль «Просмотр правил» (Rule Viewer) позволяет проиллюстрировать ход логического вывода по каждому правилу, получение результирующего нечеткого множества и выполнение процедуры дефаззификации [25]. Rule Viewer может быть вызван из любого GUI-модуля, командой **View rules …** меню **View** или нажатием клавиш **Ctrl+4**. Вид **Rule Viewer** для системы логического вывода tipper с указанием функционального назначения основных полей графического окна приведен на рис. 9.17.

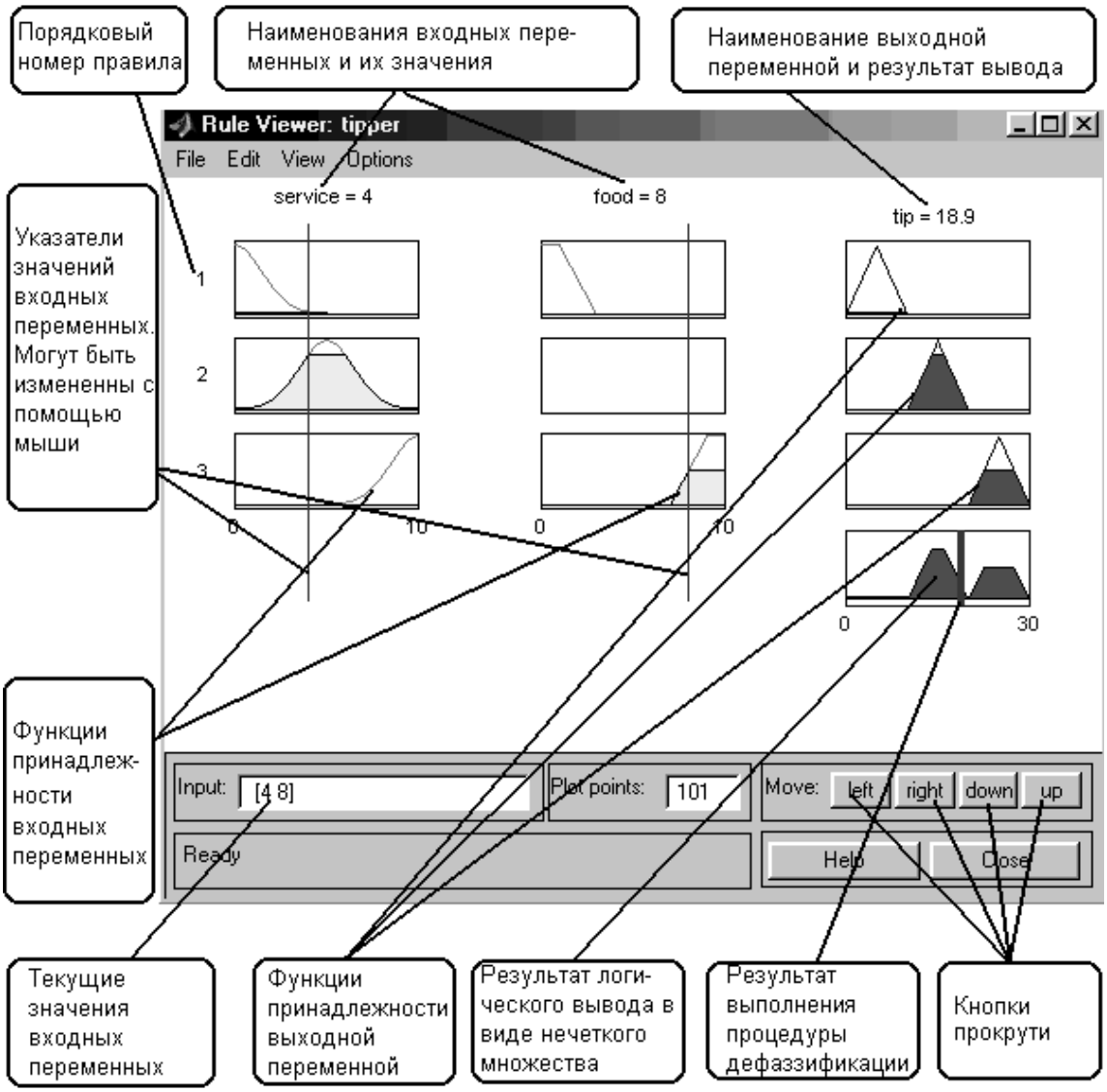


Рис. 9.17 Визуализация логического вывода для системы tipper с помощью Rule Viewer

Rule Viewer содержит четыре меню – **File, Edit, View, Options**, два поля ввода ввода информации – **Input** и **Plot points** и кнопки прокрутки изображения влево-вправо (left-right), вверх-вниз (up-down). В нижней части графического

окна расположены также кнопки **Help** и **Close**, которые позволяют вызвать окно справки и закрыть редактор, соответственно.

Каждое правило базы знаний представляется в виде последовательности горизонтально расположенных прямоугольников. При этом первые два прямоугольника – слева и по центру (рис. 9.17) отображают функции принадлежностей термов посылки правила (**ЕСЛИ** – часть правила), а последний третий прямоугольник – справа соответствует функции принадлежности терма-следствия выходной переменной (**ТО** – часть правила). Пустой прямоугольник в визуализации второго правила означает, что в этом правиле посылка по переменной food отсутствует (food is none). Темная заливка графиков функций принадлежностей входных переменных указывает насколько значения входов, соответствуют термам данного правила. Для вывода правила в формате Rule Editor необходимо сделать однократный щелчок левой кнопки мыши по номеру соответствующего правила. В этом случае указанное правило будет выведено в нижней части графического окна.

Темная заливка графика функции принадлежности выходной переменной представляет собой результат логического вывода в виде нечеткого множества по данному правилу.

Результирующее нечеткое множество, соответствующее логическому выводу по всем правилам показано в нижнем прямоугольнике последнего столбца графического окна. В этом же прямоугольнике красная вертикальная линия соответствует четкому значению логического вывода, полученного в результате дефаззификации.

Ввод значений входных переменных может осуществляться двумя способами:

- путем ввода численных значений в поле **Input**;
- с помощью мыши, путем перемещения линий-указателей красного цвета.

В последнем случае необходимо позиционировать курсор мыши на красной вертикальной линии, нажать на левую кнопку мыши и не отпуская ее переместить указатель на нужную позицию. Новое численное значения соответствующей входной переменной будет пересчитано автоматически и выведено в окно **Input**.

В поле **Plot points** задается количество точек дискретизации для построения графиков функций принадлежности. Значение по умолчанию – 101.

Общий вид меню Edit приведен на рис. 9.18.

Команда **FIS Properties…** открывает FIS-редактор. Эта команда может быть также выполнена нажатием **Ctrl+1**.

Команда **Membership Functions…** открывает редактор функций принадлежностей. Эта команда может быть также выполнена нажатием **Ctrl+2**.

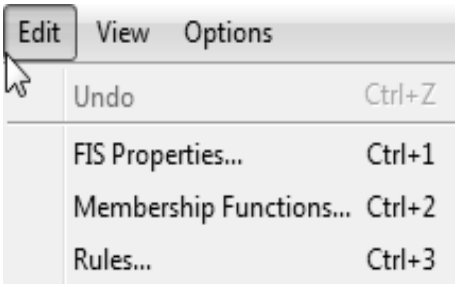


Рис. 9.18 Меню Edit

Команда **Rules…** открывает редактор базы знаний. Эта команда может быть также выполнена нажатием **Ctrl+3**.

Меню **Options** содержит только одну команду **Format**, которая позволяет установить один из следующих форматов вывода выбранного правило в нижней части графического окна: Verbose – лингвистический; Symbolic – логический; Indexed – индексированный.

9.3.5 Модуль просмотра поверхности (пространства управления)

Модуль просмотра поверхности (пространства управления) – Surface Viewer осуществляет визуализацию поверхности «входы-выход». Он позволяет вывести графическое изображение зависимости значения любой выходной переменной от произвольных двух (или одной) входных переменных [25]. Surface Viewer может быть вызван из любого GUI-модуля, используемого с системами нечеткого логического вывода, командой **View surface …** меню **View** или нажатием клавиш **Ctrl+4**. Общий вид модуля Surface Viewer с указанием функционального назначения основных полей графического окна приведен на рис. 9.19.

Surface Viewer содержит четыре верхних системных меню – **File, Edit, View, Options**, три меню выбора координатных осей – **X (input), Y (input), Z (output)**, три поля ввода ввода информации – **X girds, Y girds, Ref. Input** и кнопку **Evaluate** для построения поверхности при новых параметрах. В нижней части графического окна расположены также кнопки **Help** и **Close**, которые позволяют вызвать окно справки и закрыть редактор, соответственно.

Surface Viewer позволяет вращать поверхность «входы-выход» с помощью мыши. Для этого необходимо позиционировать курсор мыши на поверхности «входы-выход», нажать на левую кнопку мыши и не отпуская ее повернуть графическое изображение на требуемый угол.

Поля **X girds** и **Y girds** предназначены для задания количества точек дискретизации по осям X и Y, для построения поверхности «входы-выход». По умолчанию количество дискрет по каждой оси равно 15. Для изменения этого

значения необходимо установить маркер на поле X girds (Y girds) и ввести новое значение.

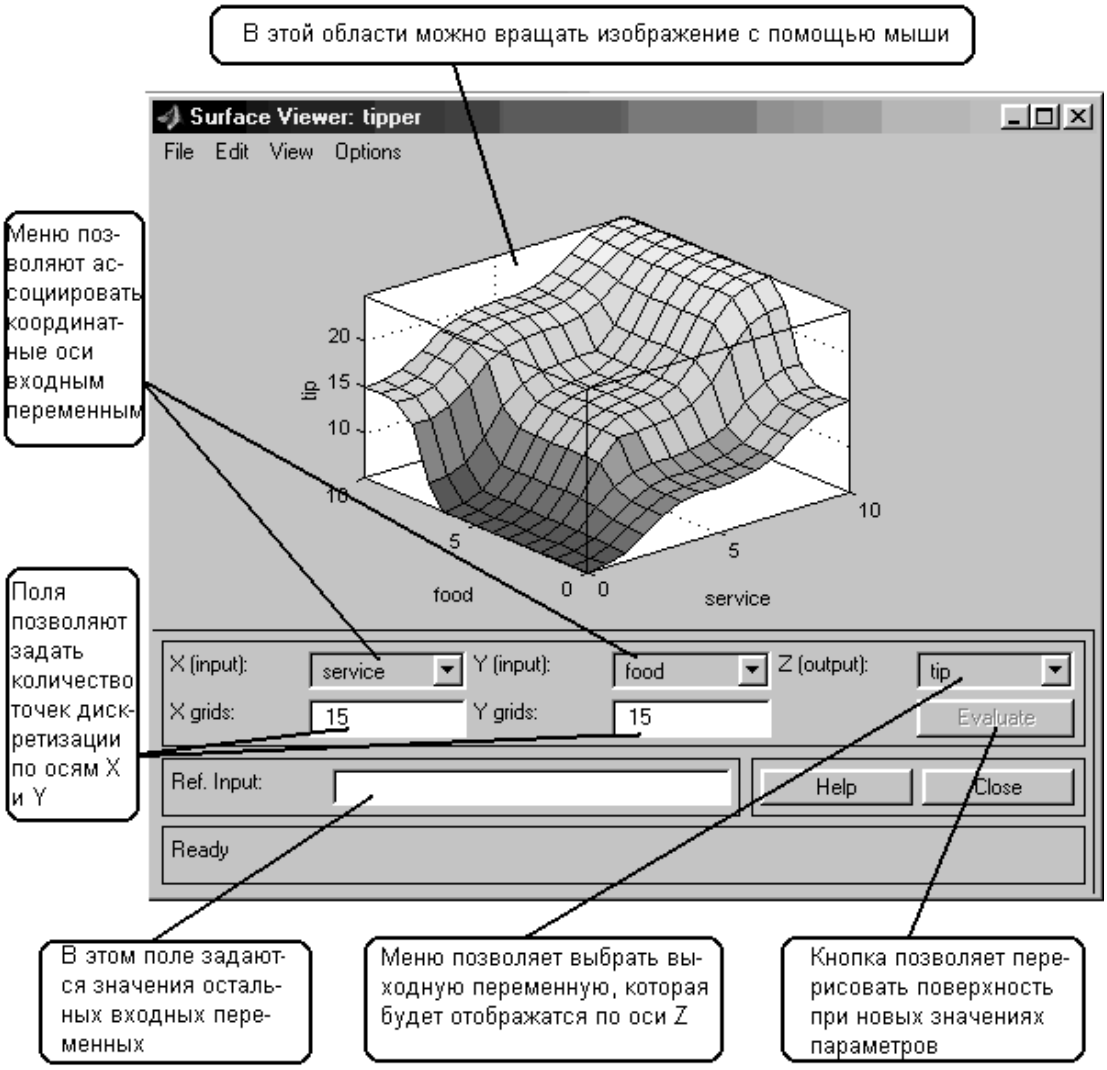


Рис. 9.19 Визуализация поверхности «входы-выход» для системы tipper с помощью Surface Viewer

Поле **Ref. Input** предназначено для задания значений входных переменных, кроме тех, которые ассоциированы с координатными осями. По умолчанию это значения середины интервалов изменения переменных. Для изменения этого значения необходимо установить маркер на поле **Ref. Input** и ввести новые значение.

Меню координатных осей. Меню **X (input), Y (input), Z (output)** позволяют поставить в соответствие осям координат входные и выходные переменные. При этом входные переменные могут отображаться только по осям X и Y, а выходные переменные только по оси Z. В Surface Viewer предусмотрена возможность построения однофакторных зависимостей «вход-выход». Для

этого в меню второй координатной оси (X (input) или Y (input)) необходимо выбрать none.

Меню **Edit**. Общий вид меню приведен на рис. 9.18. Назначения команд меню описано в подразделе 9.3.4.

Меню **Options** изображено на рис. 9.20. Оно содержит команды **Plot, Color Map** и **Always evaluate**.

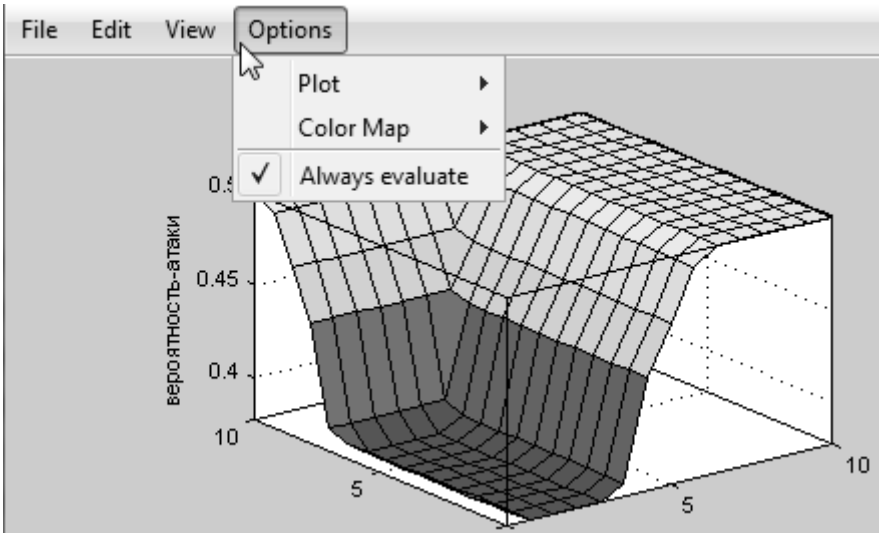


Рис. 9.20 Меню Options

Команда Plot позволяет управлять форматом вывода поверхности «входы-выход». При выборе этой команды появляется меню (рис. 9.21) в котором необходимо выбрать формат вывода поверхности.

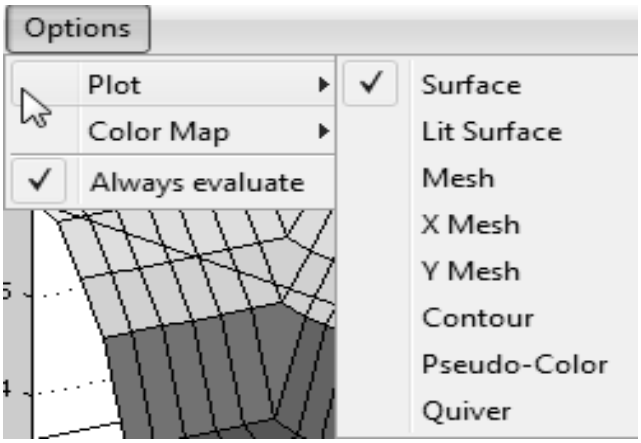


Рис. 9.21 Меню Plot

Команда **Color Map** позволяет управлять палитрой цветов при выводе поверхности «входы-выход». При выборе этой команды появляется меню (рис. 9.22), в котором необходимо выбрать одну из палитр:

- **default** – использовать палитру, установленную по умолчанию;
- **blue** – холодная сине-голубая палитра;

– **hot** – теплая палитра, состоящая из черного, красного, желтого и белого цветов;
– **HSV** – палитра насыщенных цветов: красный, желтый, зеленый, циан, голубой, мажента, красный.

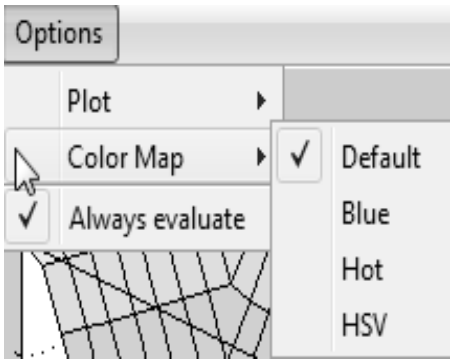


Рис. 9.22 Меню Color Map

Команда **Always evaluate** позволяет установить (отменить) режим автоматического, т. е. без нажатия кнопки **Evaluate**, перерисовывания поверхности «входы-выход» при любом изменении параметров.

9.4 Пример построения системы нечеткого вывода

Для лучшего уяснения возможностей программы Matlab рассмотрим пример создания модели системы поддержки принятия решения (СППР) на основе математического аппарата нечеткого вывода для выявления вероятности программых воздействий (ПВ) в АСУ. В качестве входных данных будем использовать шесть входных функций принадлежности (ФП), а в качестве выходных – одну ФП (по аналогии с подразделом 5.3 пособия).

Для наблюдаемых состояний защищаемой АСУ функциями принадлежности входных переменных являются:

1. «**Audit 1hour»** – количество попыток неудачного входа в систему за последний час.
2. «**Audit 24hour»** – количество попыток неудачного входа в систему за последние 24 часа.
3. «**Network activity»** – средняя нагрузка на сетевой интерфейс (в % от максимальной нагрузки за последнюю минуту).
4. **«Time of day»** – соответствие текущего времени суток рабочему.

5. **«The started processes»** – соответствие имен выполняемых процессов, пути к файлам и их контрольных сумм «эталону».

6. **«Open ports»** – соответствие номеров открытых портов «эталону».

Функцией принадлежности выходной переменной «**вероятность атаки**» является – степень уверенности СППР в наличии ПВ в АСУ.

Шаг 1. Укажите: тип системы **type** – **'Mamdani'**; **andMethod** – **'min'**; **orMethod** – **'max'**; **defuzzMethod** (для систем типа Мамдани) – **'centroid'**.

Шаг 2. В качестве основной ФП (входной и выходной) будем использовать обобщенную колоколообразную ФП (см. рис. 5.2), поэтому устанавливаем параметр **input.mf.type – gbellmf, output.mf.type** – **gbellmf**.

Шаг 3. Переименуем первую входную переменную. Для этого сделаем один щелчок левой кнопкой мыши на блоке input1, введем новое обозначение «**Audit 1hour»** в поле редактирования имени текущей переменной (рис. 9.23) и нажмем <Enter>.

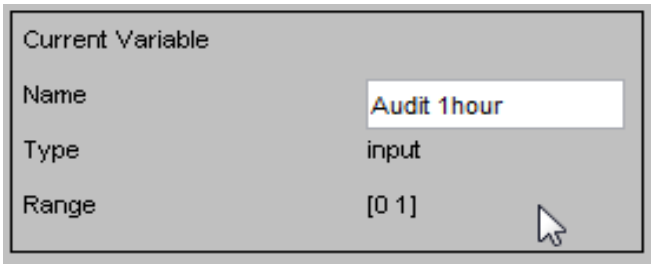


Рис. 9.23 Изменение названия входной переменной в окне Current Variable

После изменение названия входной переменной вы увидите изменения в FIS-редакторе (рис. 9.24)

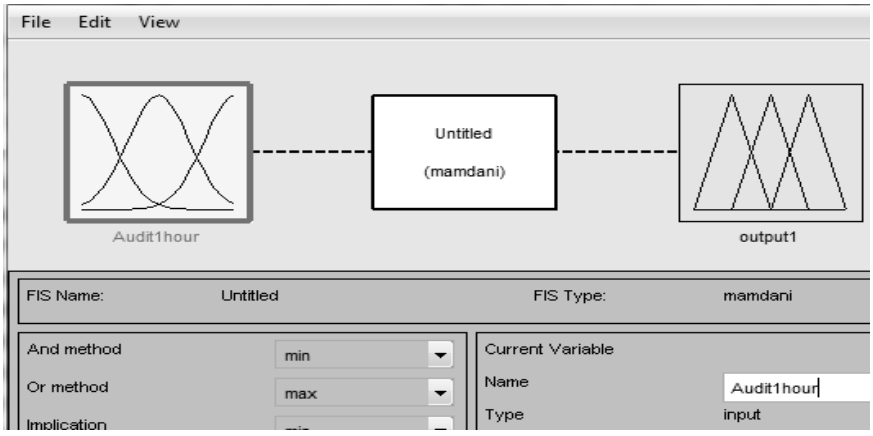


Рис. 9.24 FIS-редактор после изменения названия входной переменной

Через меню **Edit** с помощью команды **Membership Function…** открыть редактор функций принадлежностей и внесите изменения в информацию о

терм-множествах переменной **Audit-1hour** с учетом мнений экспертов. В результате получился следующая ФП (рис. 9.25).

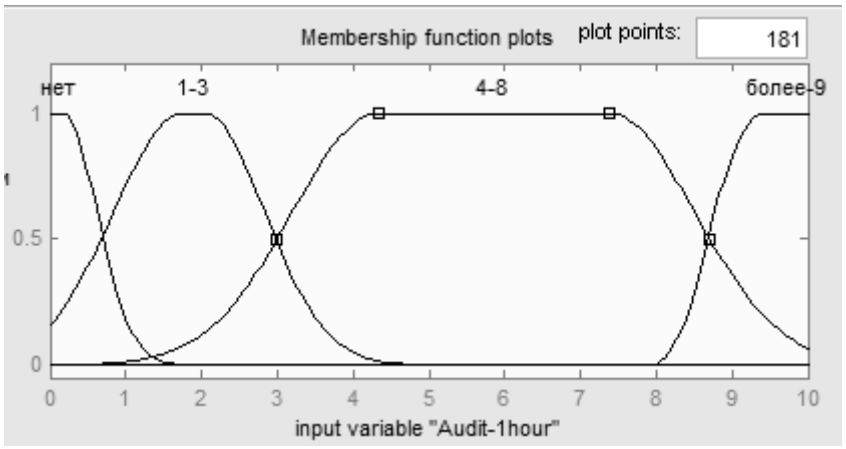


Рис. 9.25 ФП входной переменной Audit-1hour

Шаг 4. Добавим вторую входную переменную. Для этого в меню Edit выбираем команду **Add Variable…** и далее **input.**

Шаг 5. Переименуем вторую входную переменную. Для этого сделаем один щелчок левой кнопкой мыши на блоке input2, введем новое обозначение «**Audit-24hour»** в поле редактирования имени текущей переменной и нажмем <Enter>.

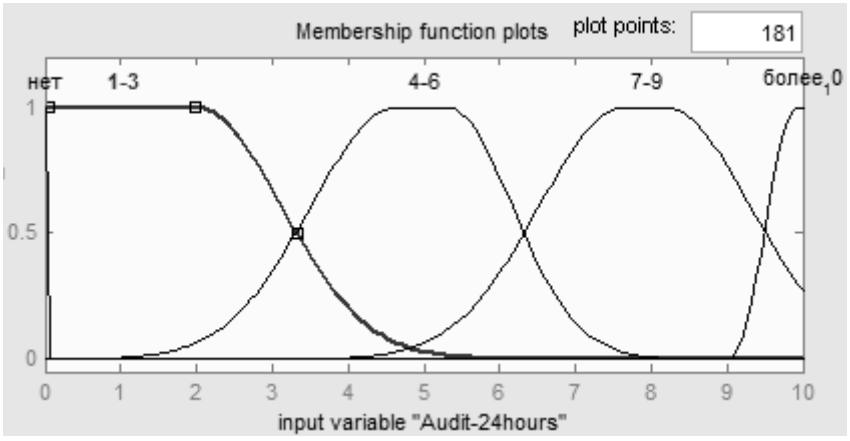


Рис. 9.26 ФП входной переменной Audit-24hour

Шаг 6. Аналогично шагам 2-3, последовательно, добавим и переименуем 3–6 входные переменные в следующие переменные: 3 – «**Network-activity»**, 4 – **«Time-of-days»**, 5 – **«The-started-processes»**, 6 – **«Open-ports»** (рис. 9.27–9.29).

Шаг 7. Переименуем выходную переменную. Для этого сделаем один щелчок левой кнопкой мыши на блоке output1, введем новое обозначение у в поле редактирования имени текущей переменной – «**вероятность-атаки**» и нажмем <Enter>.

Шаг 8. Зададим имя системы. Для этого в меню File выбираем в подменю Export команду **To disk** и вводим имя файла, например, «База правил нечеткого вывода СОА».

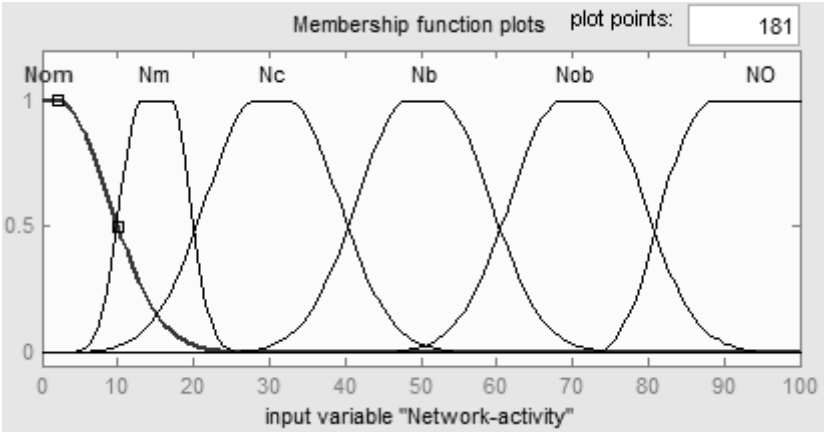


Рис. 9.27 ФП входной переменной Network-activity

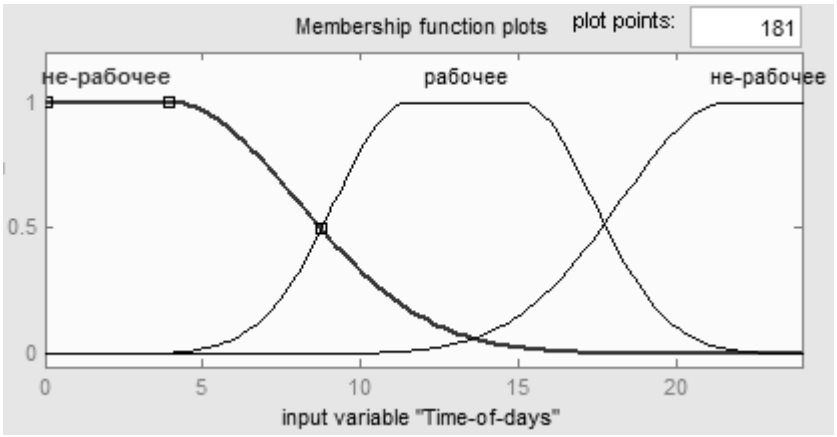


Рис. 9.28 ФП входной переменной Time-of-days

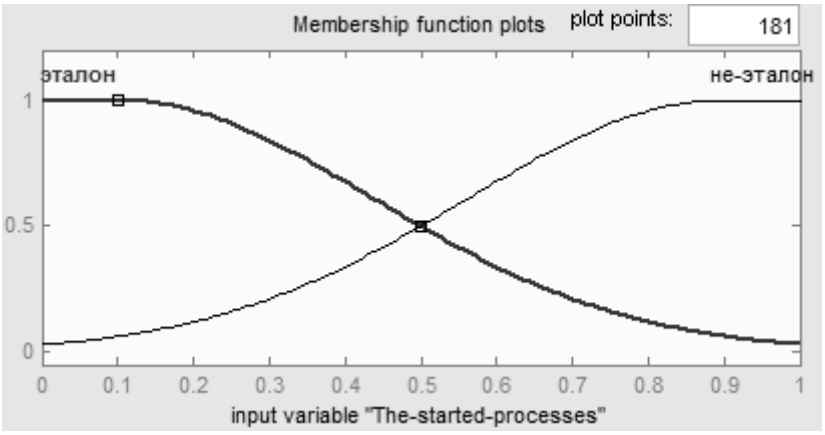


Рис. 9.29 ФП входной переменной The-started-processes

В результате этих шагов СППР в FIS-редакторе примет вид, показанный на рис. 9.30.

Шаг 9. В соответствии с правилами, разработанными экспертами, заполним базу знаний модели СППР вводя правила в «Редактор базы знаний». Для ввода правил войдите в Меню **Edit** откройте редактор базы знаний командой **Rules…** (рис. 9.31).

Вводимое правило может иметь следующий вид: If (Audit-1hour is 1-3) and (Audit-24hours is 4-6) and (Network-activity is Nc) and (Time-of-days is рабочее) and (The-started-processes is эталон) and (Open-ports is эталон) then (вероятность атаки is низкая). Введите все необходимые правила в базу знаний.

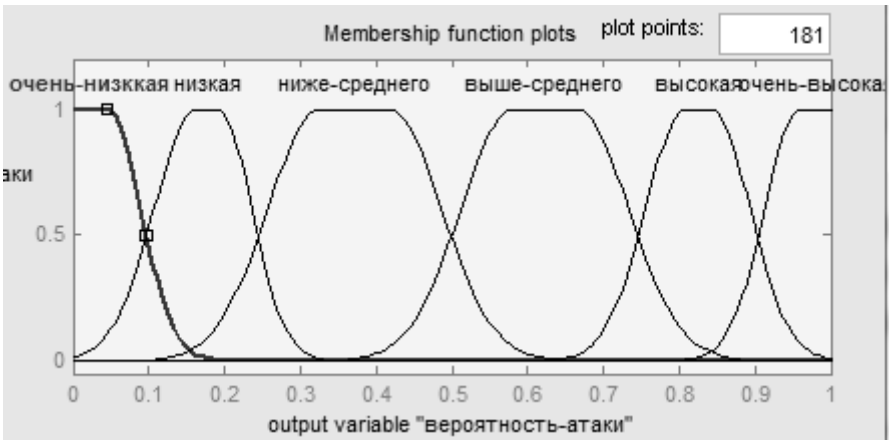


Рис. 9.30 ФП выходной переменной

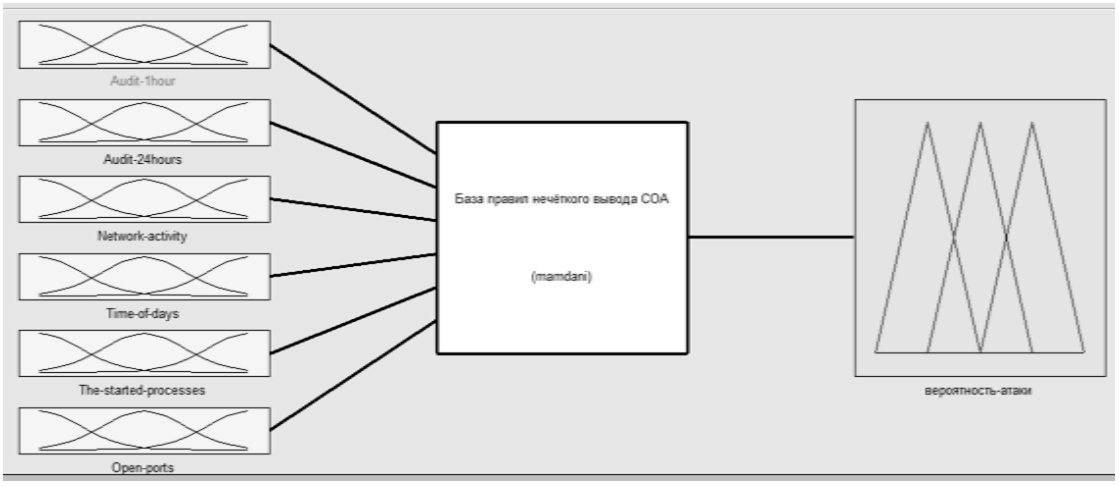


Рис. 9.31 Окончательный вид СППР в FIS-редакторе

Шаг 10. Для просмотра результатов работы модели СППР воспользуемся модулем «Просмотра правил» (см. п. 9.3.4). Введем в качестве вводных данных значения указанные в верхней части рис. 9.32.

Для результата визуализации будем использовать только три правила из базы знаний. Результатом нечеткого логического вывода будет степень уверенности СППР в наличии программых воздействий в АСУ. В данном случае (рис. 9.32) она будет равняться 0,135 или «низкая».

Шаг 11. Теперь для результата визуализации будем использовать все 42 правила из базы знаний. В данном случае (рис. 9.33) степень уверенности СППР в наличии программых воздействий в АСУ будет равняться 0,182 или «низкая».

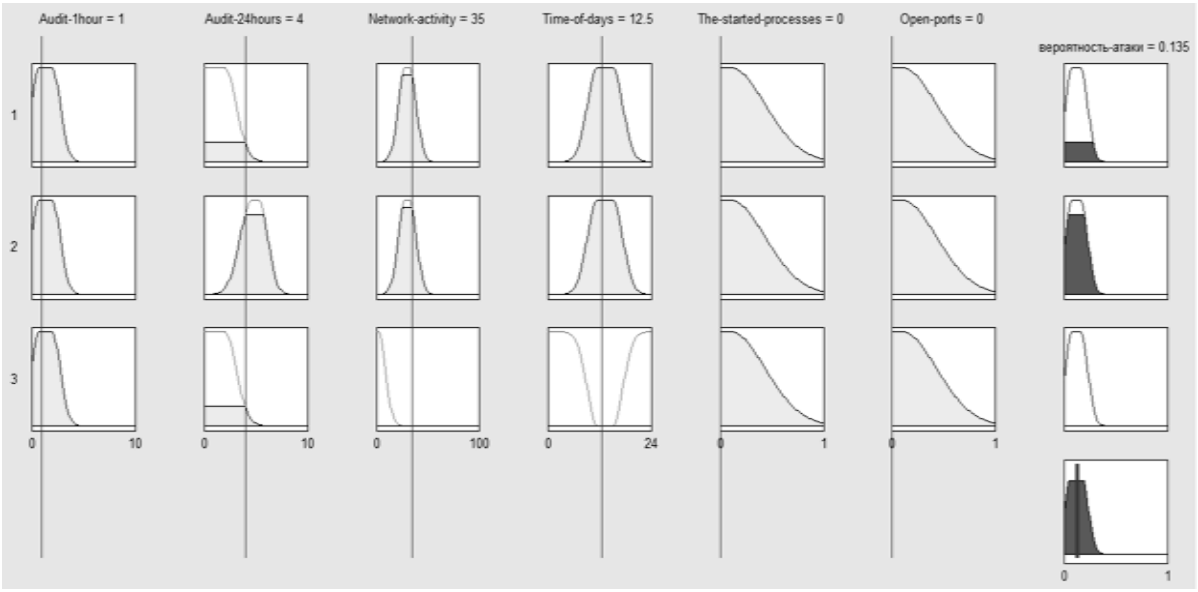


Рис. 9.32 Визуализация логического вывода для трех правил

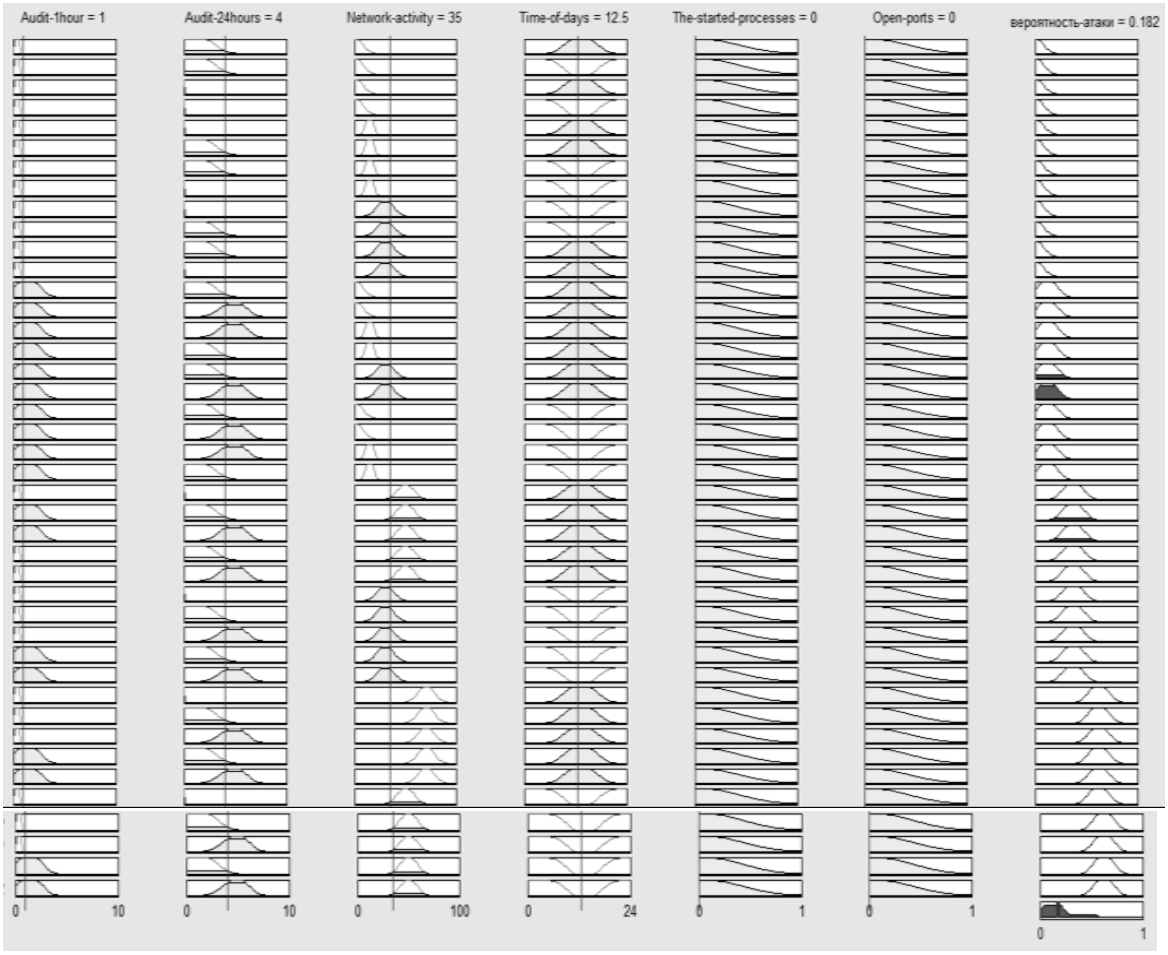


Рис. 9.33 Визуализация логического вывода для 42-х правил

10 РАБОТА В СИСТЕМЕ LEONARDO

10.1 Правила и объекты в системе LEONARDO

10.1.1 Основные сведения

Структура системы LEONARDO представлена на рис. 10.1. Основными элементами системы являются: диалоговый интерфейс, редактор текстов, система объяснений, машина логического вывода, система программирования и отладки, а также компилятор.

Особенностью системы является то, что один и тот же диалоговый интерфейс используется в периоды выполнения (пользователи) и разработки (программисты, инженеры знаний, эксперты). Число возможных баз знаний ограничено лишь объемом директория (каталога), в котором обязательно совместно должны быть размещены система LEONARDO и базы знаний.

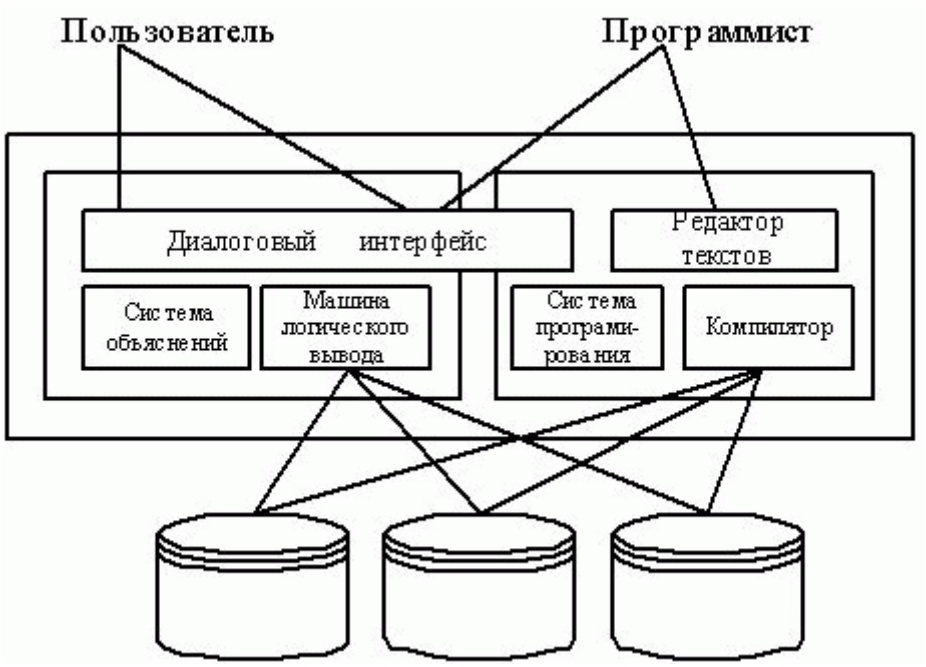


Рис.10.1 Структура системы LEONARDO

База знаний включает правила, простые и сложные объекты (фреймы), содержащие помимо основной также и дополнительную информацию, такую, например, как разметка экрана, процедуры и пр.

Редактор системы предоставляет полный набор возможностей для создания и редактирования правил базы знаний, экранных форм, процедур и фреймов объектов.

Система программирования и отладки включает собственный процедурный язык для построения ЭС, в которых может понадобиться доступ к внешним базам данных, выполнение расчетов, распечатка сложных отчетов и т. п. Кроме того, в системе программирования и отладки реализуются функции проверки и исполнения базы знаний.

Компилятор преобразует исходную базу знаний, сформированную с помощью редактора, в исполняемый формат, и только после этого ЭС может функционировать в режиме консультации. Таким образом, LEONARDO относится к разряду компилирующих систем.

10.1.2 Состав базы знаний, простые и сложные объекты

Знания в системе представляются в виде правил, простых объектов, сложных объектов (с фреймами), содержащих информацию, процедуры, входные и выходные формы и т. п.

Состав базы знаний схематически представлен на рис. 10.2.

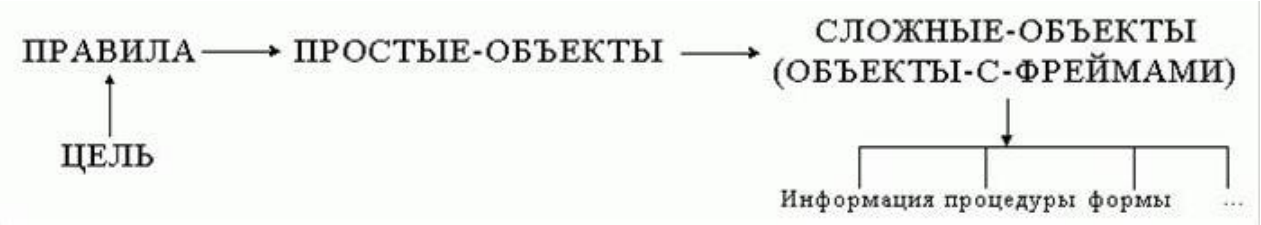


Рис. 10.2 Состав базы знаний системы LEONARDO

Правила являются условными выражениями следующего общего вида:

if УСЛОВИЕ **then** ДЕЙСТВИЕ ,

где УСЛОВИЕ – это некоторое утверждение, истинность которого может быть проверена. Если УСЛОВИЕ выполнено, то выполняется ДЕЙСТВИЕ. ДЕЙСТВИЕ может означать следующее:

1) вывод информации на экран;

2) присваивание значений объектам, которые затем могут быть использованы в других правилах;

3) вычисление значений с помощью процедур, написанных на языке системы LEONARDO;

4) вычисление значений с использованием внешних процедур;

5) вызов внешних процедур обработки данных.

Например, пусть правило имеет следующий вид:

if ‘сфера деятельности’ **is** ‘служебная’ (условие)
then ‘собака’ **is** ‘ротвейлер’ (действие)

Это правило включает в себя логическое выражение (if... then...) и объекты, к которым это логическое выражение применяется. Объекты могут быть числового, текстового и спискового типов. Все объекты имеют, по крайней мере, имя и значение.

В приведенном примере есть два объекта с именами **‘сфера деятельности’** и **‘собака’** со своими значениями, соответственно, *служебная* и *ротвейлер*. Это – простые объекты, так как они имеют только имена и значения. Система распознает их как объекты потому, что они используются в правиле. При этом нет необходимости определять эти простые объекты отдельно от правил.

При компиляции базы знаний система автоматически идентифицирует и устанавливает все простые объекты и определяет их тип на основе контекста их применения в правиле. В приведенном примере объекты имеют текстовой тип.

Другой формат правил позволяет определять тип объектов как числовой. Любой объект, используемый совместно с символами '=', ' <= ', '<', ' > ', ' >= ', '<>', будет идентифицироваться системой как числовой.

Объекты могут иметь гораздо больше атрибутов, чем только имя и значение. Эти дополнительные атрибуты хранятся во фрейме объекта, и такие объекты называются ***объектами - с - фреймами***.

Фреймы – одна из наиболее важных структур базы знаний в системе LEONARDO. Они имеют множество различных применений: например, процедуры хранятся во фреймах. Содержание фреймов определяется разработчиком, создающим прикладную ЭС.

10.1.3 Обработка правил с простыми объектами

В систему LEONARDO встроены стратегии поиска, которые используют ***прямой*** и ***обратный*** вывод.

Обратный вывод иногда называют поиском от цели, так как он начинается от возможного результата и выполняется в обратную сторону, проверяя условия, которые позволили бы подтвердить результат. Каждое условие, в свою очередь, может вести к другому правилу. Таким образом, строится ***дерево правил***, каждое из которых должно быть выполнено для того, чтобы достичь цели. Приведенный ниже пример обработки правил с простыми объектами иллюстрирует стратегию обратного вывода. Прямой вывод выполняется в прямом направлении от условий правил. Когда условие выполнено, правило срабатывает и выполняется его заключение.

В системе LEONARDO используются обе стратегии. При этом происходит наиболее рациональное использование информации, поступающей в систему. Такая смешанная стратегия заключается в том, что сначала начинает работать механизм обратного вывода, а по мере поступления информации в систему, может подключаться механизм прямого вывода, во время которого эта новая информация используется максимально возможным числом правил.

Наиболее простая база знаний в системе LEONARDO состоит только из правил и простых объектов. Например:

seek Собака
if ‘сфера деятельности’ **is** ‘*служебная*’
then Собака **is** ‘*ротвейлер*’
if ‘сфера деятельности’ **is** ‘*пастушья*’
then Собака **is** ‘*кавказская овчарка*’ .

Обработка этих правил осуществляется следующим образом:

1) система находит директиву **seek** и устанавливает цель. В нашем случае целью является присваивание некоторого значения объекту Собака;

2) определяется список возможных значений объекта Собака. В нашем случае этот список состоит только из двух элементов – *ротвейлер* и *кавказская овчарка*;

3) выбирается первый элемент списка и система осуществляет вывод для того, чтобы подтвердить или опровергнуть это значение целевого объекта. В нашем случае система выбирает в качестве возможной цели *ротвейлер*, просматривает правила и находит клаузу **if** ‘сфера деятельности’ **is** служебная;

4) система ищет в базе знаний правило, которое содержало бы в правой части утверждение ‘сфера деятельности’ **is** служебная, чтобы проверить его. В нашем случае в базе знаний такого правила нет, поэтому система спрашивает у пользователя значение объекта ‘сфера деятельности’. Предположим, что пользователь ответил – пастушья. Теперь система знает, что истинное значение утверждения ‘сфера деятельности’ is служебная – ложь и, следовательно, рассматриваемое правило не срабатывает, т. е. *ротвейлер* не является значением объекта Собака;

5) система выбирает из списка следующее возможное значение цели – кавказская овчарка и вновь просматривает базу знаний от цели. На этот раз система находит правило с условием “if ‘сфера деятельности’ is пастушья” и ищет информацию для проверки этого утверждения. Так как пользователь ранее уже ввел значение объекта ‘сфера деятельности’ = пастушья, то правило

if ‘сфера деятельности’ **is** ‘*пастушья*’
then Собака **is** ‘*кавказская овчарка*’

срабатывает;

6) система выводит на экран свое заключение, так как целевому объекту присвоено значение, и директива ***seek*** выполнена.

10.1.4 База правил

База правил (главный набор правил) в системе LEONARDO представляет собой список правил, записанных в свободном формате. В тексте базы правил они обязательно отделяются друг от друга, по крайней мере, одной пустой строкой. При этом правила могут располагаться в базе правил в любой последовательности.

Синтаксис правил в системе LEONARDO следующий:

[**Rule:** 'имя правила']

if {<ЧислОбъект> <= | = | > | >= | < > | <значение>}|
{ (<объект> <оператор> <объект>} [<**and**> | <**or**>
<ТекстОбъект> <**is** | **is not** <значение>...]
[< and | or> <СписокОбъект> <**includes** | **excludes**>
<значение>…]

then <ЧислОбъект> = <значение> [;
<ТекстОбъект> **is** <значение>] [;
<СписокОбъект> <**inclndes** | **excludes**> <значение>] [;

run <ПроцОбъект> <СписокПарам>] [;
<ТекстОбъект> **is** значение

Базовые правила всегда начинаются с ключевого слова **if**. Кроме базовых правил, существуют также специальные правила, в которых используются ключевые слова **seek**, **ask**, **say**.

Ключевое слово **if** маркирует начало нового правила и его антецедент, а ключевое слово then маркирует вторую часть правила (консеквент).

С помощью дополнительных ключевых слов **and** и **or** можно строить более сложные правила, например:

if антецедент-1 **or** антецедент-2
then консеквент-1

или

if антецедент-1 **and** антецедент-2
then консеквент-3 .

На количество клауз антецедента в правилах ограничений не накладывается. Кроме того, связки **and** и **or** могут одновременно использоваться в антецеденте одного правила.

Машина логического вывода системы LEONARDO проверяет истинность антецедента правил по следующему алгоритму:

если все клаузы антецедента связаны через **'and'** и найдено, что одна из клауз принимает значение FALSE, то правило маркируется как несработавшее;

если все клаузы антецедента соединены связкой **‘or’** и найдено, что хотя бы одна клауза истинна – TRUE, то правило маркируется как сработавшее.

Антецедент правила включает в себя объект и значение, связанные оператором сравнения. Имена объектов могут быть любой длины, но не должны содержать символов +, -, *, /, =, (,), <, >, а также символа пробела, который интерпретируется как конец имени объекта. Если в имени объекта необходим символ пробела или другие упомянутые выше специальные символы, то имя объекта должно заключаться в одинарные кавычки.

Объекты могут быть числовые или текстовые, и они должны принимать, соответственно, числовые и текстовые значения. Тип объекта устанавливается системой автоматически в зависимости от контекста его первого применения в правиле.

Для текстовых объектов LEONARDO использует оператор is или его синоним are: *объект are значение*. Текстовое значение представляет собой цепочку символов, не содержащую пробелов. При наличии пробелов текстовое значение необходимо заключать в одинарные кавычки: ‘пример текстового значения с пробелом’. В том случае, если предполагатся, что значением текстового объекта являются символы цифр, то их также необходимо заключать в одинарные кавычки. Например, цена is ‘150’.

Числовые значения являются действительными числами. Для сопоставления с числовыми объектами могут использоваться следующие операторы сравнения:

=	равно;
<=	не больше, чем;
<	меньше, чем;
>=	не меньше, чем;
>	больше, чем;
<>	не равно.

Консеквент правил также состоит из клауз, которые присваиваются объектам заключения в случае, если условия антецедента удовлетворены.

Для текстовых объектов основной формат консеквента имеет следующий вид:

объект **is** значение,

а для числовых

объект = значение / выражение.

Выражения, значения которых присваиваются числовым объектам, состоят из имен объектов, числовых значений и операторов + - * /. Причем в консеквенте могут использоваться только простые выражения (без скобок).

Более сложные вычисления и соответствующие им выражения должны быть реализованы с помощью фреймов-процедур.

Консеквент правила может состоять из любого количества клауз. В этом случае клаузы разделяются при помощи точки с запятой, имеющей смысл конъюнкции.

В консеквенте правила могут использоваться вызовы процедур (**run**) и безусловные правила **ask** и **say.**

При вызове процедуры в консеквенте правила необходимо написать ключевое слово **run**, далее должно следовать имя процедуры и в скобках – аргументы процедуры. Наличие аргументов является обязательным. В том случае, если не предполагается передавать (принимать) какие-либо параметры, то достаточно написать, например,... run proc(0), При этом в любом из слотов фрейма – процедуры, обеспечивающих прием/возврат параметров, достаточно указать произвольное имя фиктивного объекта.

В системе LEONARDO предусмотрена жесткая последовательность размещения аргументов процедуры: сначала перечисляются имена передаваемых параметров (входных для процедуры), а затем – имена выходных параметров (возвращаемых процедурой). При этом в каждой из упомянутых групп параметров (входных и возвращаемых) они обязательно должны быть перечислены в такой последовательности:

− числовые объекты;

− текстовые объекты;

− списковые объекты.

Данная последовательность должна быть сохранена даже в том случае, если какие-либо из типов объектов в каждой из групп параметров отсутствуют.

Вызов процедуры не может быть последней клаузой консеквента правила, так как механизм обратного вывода не работает непосредственно с параметрами процедуры. Поэтому за каждым вызовом процедуры обязательно должна следовать, клауза с объектом.

Процедуры могут выполнять следующие возможные функции:

1) вычисление значения числового объекта;

2) исполнение внешней вычислительной процедуры;

3) взаимодействие с базой данных или коммуникационным устройством;

4) выполнение операций управления при работе с базой знаний.

Безусловные правила **ask, say** и **seek**, являются специальной категорией правил системы LEONARDO.

Безусловные правила состоят из утвердительной части и не содержат условия их срабатывания. При исполнении базы знаний безусловные правила выполняют функции управления.

В каждом из безусловных правил после ключевого слова должно следовать имя объекта, которое является **обязательным.**

При обработке правила ask система запрашивает у пользователя значение соответствующего объекта, например:

ask вопрос

При обработке правила **say** система выводит на экран значение соответствующего объекта, если он уже означен, что весьма удобно при тестировании базы знаний.

При обработке директивы (правила) ask система автоматически формирует вопрос, а при обработке директивы say – генерирует формат выходного сообщения. Однако существует возможность явного задания формата вопроса и выходного сообщения путем заполнения соответствующих слотов фрейма объекта.

Правило seek используется для декларирования целевого объекта, т. е. объекта, значение которого должно быть найдено при обработке всей базы правил. Это правило может располагаться в любом месте базы знаний. Правило seek всегда является самостоятельным, и его нельзя использовать в контексте другого правила.

В системе LEONARDO существует еще один тип объектов – списковые. Объект типа списка содержит в качестве своего значения множество элементов в виде списка. Например:

if ‘время года’ **is** весна

then ‘весенние месяцы’ **include** ‘март, апрель, май’

или

if осадки **is** дождь

then ВремяГода **include** ‘весна, лето, осень’.

LEONARDO распознает списковые объекты по контексту, в котором они встречаются. Для сопоставления списковых объектов используются следующие операторы:

includes (include);

excludes (exclude);

does not include (do not include);

does not exclude (do not exclude).

Синонимы операторов, которые указаны в скобках, введены только для улучшения читаемости правил и выполняют одинаковые функции.

В правилах могут быть так называемые «шумовые слова», которые используются для того, чтобы сделать правила более читаемыми, если весь текст базы знаний разрабатывается на английском языке. К числу «шумовых

слов» относятся: a, an, its, the, do, does. Эти слова наряду с ключевыми словами правил не могут использоваться в качестве идентификаторов (имен) объектов.

10.1.5 Объекты с фреймами

Фрейм – это структура данных (т.е. декларативное представление), предназначенная для представления некоторой стандартной ситуации. В общем случае фрейм объекта может быть представлен в виде следующей конструкции:

$$f = [(r_1 v_1),(r_2 v_2),\ldots,(r_n v_n)],$$

где f – имя фрейма, r_i – имя слота, а v_i – значение слота.

Каждый фрейм объекта автоматически именуется системой с помощью имени, указанного для этого объекта в базе правил. Эти имена могут быть изменены системой только после того, когда разработчик заменит в базе правил существующее имя каким-либо другим.

Фрейм объекта в своих слотах содержит дополнительную информацию, связанную с объектом, которая описывает какое-либо свойство объекта. Название слота должно заканчиваться символом «:».

Доступ к фрейму объекта осуществляется только после ввода правил, их компиляции и выполнения опции OBJECTS.

Слоты фрейма могут быть стандартными и дополнительными, защищенными и незащищенными. Все дополнительные слоты являются незащищенными. Защищенные слоты и их значения выделены на экране красным цветом. При попытке отредактировать защищенный слот возникает звуковой сигнал и ничего не меняется.

Стандартные слоты объекта с фреймом. Каждый из слотов фрейма объекта имеет свое уникальное имя и назначение. Фрейм, созданный системой для нового объекта, например, ‘Сфера деятельности’, имеет следующий вид (стандартные слоты):

1: **Name:** ‘Сфера деятельности’
2: **LongName:**
3: **Type:**
4: **Value:**
5: **Certainty:**
6: **DerivedFrom:**
7: **DefaultValue:**
8: **FixedValue:**
9: **AllowedValue:**

10: **ComputeValue:**

11: **OnError:**

12: **QueryPrompt:**

13: **QueryPreface:**

14: **Expansion:**

15: **Commentary:**

16: **In reduction:**

17: **Conclusion:**

Дадим характеристику стандартным слотам:

Name: (защищенный слот)

Слот содержит имя, введенное при создании объекта либо косвенно, при компиляции правил, либо явно, при добавлении нового объекта к существующему списку. Имя может быть любой длины, но выводится только первые 24 символа.

LongName:

Слот содержит длинное имя объекта, не имеющее ограничение на длину. Используется системой в качестве синонима имени объекта для того, чтобы диалог выглядел более естественно. Однако внутри правил может использоваться только имя, находящееся в слоте «Name:».

Type: (защищенный слот)

В этом слоте хранится идентификатор типа объекта. Система автоматически распознает тип объекта и присваивает одно из следующих названий: Real, Text, List, Procedure.

Value: (защищенный слот)

Слот содержит текущее значение, присвоенное объекту в ходе консультации. Если база знаний не исполнялась, то слот содержит специальное нулевое значение(null). Все значения инициализируются в начале очередного исполнения базы знаний.

Certainty: (защищенный слот)

В этом слоте размещается число в диапазоне от 0 до 1, означающее степень уверенности в истинности этого значения. Если в базе правил факторы уверенности не используются, то значение этого числа равно 1.0.

DerivedFrom: (защищенный слот)

В этом слоте фиксируется источник присвоения объекту значения (источник означивания объекта) в ходе исполнения базы знаний. Возможен один из следующих четырех таких источников:

1) введено вами (ввод с терминала);

2) <правило> (выведено из правила);

3) умолчание (по умолчанию, см. слот DefaultValue);

4) фиксировано (если это объект с фиксированным значением, см. слот FixedValue).

Информация данного слота может быть весьма полезной при тестировании и отладке базы знаний.

DefaultValue:

В данном слоте может быть задано значение, которое используется системой для означивания объекта по умолчанию, в тех случаях, когда значение не может быть выведено из правил, пользователь не может ответить на вопрос о значении объекта и вводит ключевое слово **unknown** (неизвестно), см слоты AllowedValue и ForbidUnk.

FixedValue:

Слот может содержать значение объекта, которое всегда истинно. Если данный слот заполнен и объект используется в консеквенте какого-либо правила, то объекту будет присвоено это заданное фиксированное значение, независимо от значения антецедента.

AllowedValue:

Слот содержит разрешенные значения, которые может принимать объект. Используется системой для формирования меню и выдачи его пользователю в режиме консультации. Если объект текстовой, то разрешенные значения записываются в виде списка значений, разделенных запятыми. Максимальное число элементов списка равно 22. Например, если на вопрос, задаваемый пользователю в ходе консультации перечень возможных ответов включает 'да' или 'нет', то правильная запись данного слота выглядит следующим образом:

9: AllowedValue: да,нет

Для числовых объектов разрешенные значения задаются в виде логического выражения, состоящего из чисел, операторов сравнения, а также связок and и or. Например:

>= 10 and <= 20

5 or 7 or 10 > 100.

Если объект имеет тип Real и в слоте AllowedValue фрейма объекта указано выражение для разрешенных значений, то система автоматически формирует на экране окно для ввода этого значения. В том случае, если введенное число окажется вне разрешенных границ, то система самостоятельно сформирует сообщение об ошибке. Специальное значение unknown (неизвестно) всегда выводится в меню возможных ответов как для текстовых, так и для числовых объектов. Для того чтобы исключить его появление в меню, к стандартным слотам должен быть подключен дополнительный слот (см. слот ForbidUnk:).

ComputeValue:

Это особый случай исполнения процедур LEONARDO. Слот содержит имя процедуры со списком параметров. Процедура предназначена для вычисления значения объекта. Заметим, что процедура выполняется только в том случае, если требуется значение объекта, оно неизвестно и не может быть выведено из правил, а значения всех аргументов известны.

OnError:

В этом слоте может задаваться сообщение, которое система выводит на экран, если пользователь ввел значение объекта, не предусмотренное в слоте AilowedValue: (список разрешенных значений / логическое выражение).

QueryPrompt:

Слот используется системой для выдачи в ходе консультации пользователю вопроса о возможных значениях объекта. Вопрос выдается пользователю тогда, когда при исполнении базы знаний значение объекта в антецеденте правила не означено, например, **if type is not unknown** . Слот может содержать не более одной строки произвольного текста.

Слот может принимать специальное зарезервированное значение **NEVER.** В этом случае пользователь никогда не запрашивается о значении объекта. Заметим, что это не то же самое, как в случае использования слота FixedValue:, Применение слота QueryPrompt: со значением NEVER может использоваться для сокращения числа правил в главном наборе правил. Например, предположим, что необходимо найти значение объекта РЕЗУЛЬТАТ, который обычно принимает значение **дружелюбный**, но иногда значение **агрессивный**. Тогда, если у объекта РЕЗУЛЬТАТ указать в слоте QueryPrompt: значение NEVER, а в слоте DefaultValue: − **дружелюбный,** то достаточно иметь в главном наборе правил только правила для значения **агрессивный.**

QueryPreface:

В этом слоте может задаваться многострочный текст, который система выдает в режиме консультации в качестве пояснения к вопросу, указанному в слоте QueryPrompt:.

Expansion:

В данном слоте может содержаться текст неограниченного размера, в котором размещается расширенное описание объекта. Текст расширения может быть многостраничным, его можно листать с помощью ключа PgDn (PgUp). Этот текст система выводит на экран в режиме консультации после того, как пользователь нажмет ключ F7.

Commentary:

Слот разработчика прикладной ЭС, который предоставляет ему возможность аннотировать фрейм объекта, занося любую информацию, которая может понадобиться во время разработки или сопровождения ЭС.

Introduction:

В этом слоте может задаваться текст заставки, появляющийся в начале консультации с прикладной ЭС. Размер экрана заставки – 24 строки, в каждой строке – 80 символов. В нижней строке экрана система выводит сообщение: «press any key». Слот используется для объекта в команде seek.

Conclusion:

Слот позволяет задавать формат выходного сообщения, выдаваемого системой пользователю после означивания целевого объекта, т. е. в конце консультации. Слот может также использоваться для задания информации, высвечиваемой при обработке безусловного правила say.

Значения объектов могут высвечиваться путем заключения имен объектов в квадратные скобки [...], в которых может также задаваться шаблон вывода значений. Для объектов типа Real шаблон вывода задается строкой символов (любых), длина которой и положение десятичной точки, если они заданы, определяют формат вывода.

Для текстовых объектов шаблон задается с помощью символов «r» и «l», за которыми следует число. Это число означает ширину поля, в которое будет выведено значение объекта. При этом «r» означает выравнивание вправо, а «l» – выравнивание влево внутри поля. Условное управление выводом текста выполняется с помощью выражений, аналогичных антецеденту правил, заключенных в фигурные скобки {...}. Антецедент оценивается и, если он удовлетворен, то выводятся следующие за ним строки текста заключения. В противном случае – текст пропускается до следующих фигурных скобок.

Управлять экраном можно с помощью ключевого слова SCREEN, заключенного в фигурные скобки. Это слово прерывает вывод заключения до нажатия любой клавиши. В текст вывода может включаться информация, хранящаяся во внешних файлах DOS, путем указания ключевого слова **DOS** и реквизитов файла, например: {DOS a:\file.txt}. Частью заключения может быть использование процедуры, например, {**run** имя-процедуры (аргументы)}. При этом после предложения вызова процедуры не должно быть никаких других предложений, в том числе и пустых строк.

Дополнительные слоты объекта с фреймом

Помимо стандартного набора слотов во фрейм объекта разработчик прикладной ЭС может включить следующие **дополнительные слоты:**

BoxWidth:
ForbidUnk:
Form:
IntroAttrib:
Layout:
RuleSet:
AVExpansion:

При этом наличие двоеточия в конце имени слотов является обязательным.

BoxWidth:

По умолчанию для ввода значения объекта, не содержащего множества допустимых значений, отводится окно длиной 60 символов для текстового объекта и 10 символов – для числового объекта. Включая в любое место фрейма этот слот, можно задавать собственную длину окна ответа. Например: BoxWidth: 26.

ForbidUnk:

В этом слоте его значение не указывается. Если этот слот добавлен во фрейм объекта, то он отключает механизм вывода в меню возможных ответов пользователя значение unknown.

Form:

Наличие во фрейме объекта слота Form: позволяет описать форму вывода для выдачи на печать. Объект, во фрейм которого помещен этот слот, должен быть текстового типа. Слот должен быть последним во фрейме, и далее должно следовать определение отчета, которое может содержать:

1) свободный текст, который будет располагаться в печатном документе также, как он выглядит на экране при описании формы отчета;

2) имена объектов с использованием шаблонов печати или без них. Для вывода на печать текущего значения объекта его имя заключается в квадратные скобки, например: [наименование-объекта]. Квадратные скобки сами по себе не должны использоваться в описании отчета. По умолчанию значение текстового объекта будет занимать столько символов, сколько содержит его текущее значение, а в случае числового объекта – девять символов для целой части, два символа – для дробной части. Форматы можно переопределять с помощью спецификаций шаблона печати внутри квадратных скобок. Существуют следующие шаблоны:

lnn – печать в поле размером nn символов с левым выравниванием;

rnn – аналогично, но с правым выравниванием;

xx.x – печать значений числовых объектов с соответствующим числом символов в целой и дробной частях;

3) символы генерации рамок, которые используются для указания углов рамки в соответствующих местах описания формы отчета. Символы вводятся

в виде их ASCII кодов путем набора трехзначного числа при нажатой клавише Alt. Например: Alt + 201 – верхний левый угол; Alt + 187 – верхний правый угол; Alt + 200 – нижний левый угол; Alt + 188 – нижний правый угол. Форма, определенная в слоте Form: , распечатывается с помощью команды процедурного языка **Formprint**(объект), которая содержит имя фрейма объекта, фрейм которого включает слот с описанием формы вывода на печать.

IntroAttrib:

Данный слот и его значение, взятое из таблицы разрешенных цветовых атрибутов, используется для задания собственных цветов фона и символов заставки, открывающей сеанс консультации с прикладной ЭС.

Layout:

Данный слот должен быть последним во фрейме текстового объекта. Он определяет объект как структурированную запись и используется вместе с командами процедурного языка LEONARDO. Подробно этот слот будет рассмотрен при изложении вопросов чтения и записи файлов на диск

RuleSet:

Любой объект может содержать набор правил, которые используются для логического вывода его значений.

Основной принцип работы системы LEONARDO состоит в следующем:

1) когда для оценки антецедента правила в базе правил требуется значение объекта, то система просматривает текущий набор правил и ищет правило, которое могло бы вывести значение объекта;

2) если такое правило не найдено и фрейм объекта не содержит слот RuleSet: , то система обращается к фрейму объекта, так как возможно, что фрейм объекта содержит вопрос для выдачи его пользователю или процедуру для его вычисления (определения);

3) если такое правило не найдено и фрейм объекта содержит слот RuleSet: , то активность текущего набора правил в базе правил прерывается и система начинает обрабатывать набор правил в слоте RuleSet: . Как только в наборе правил этого слота значение объекта будет установлено, осуществляется возврат к набору, из которого был активизирован текущий набор и обработка (логический вывод) продолжается.

AVExpansion:

Данный слот позволяет связать произвольный текст с каждым элементом списка разрешенных ответов. Этот текст будет выводиться в окне размером в девять строк. Длина окна текста зависит от длины максимального элемента списка разрешенных значений ответов, так как окно меню формируется первым и только после этого на оставшейся незанятой площади окна создается окно расширения. Строки, оказавшиеся длиннее, обрезаются справа.

Для разделения элементов произвольного текста и их связи с разрешенными значениями ответов используется знак минус «-». Этот знак необходимо использовать в качестве маркера в первой и последней строках слота (в первой позиции каждой строки). Если для какого-либо элемента из списка разрешенных значений не задается никакого текста, то необходимо сделать пустой параграф, т. е. два знака минус в последовательных строках.

10.1.6 Редактирование объектов

При выборе OBJECTS в главном меню, высвечивается электронная таблица объектов, в которой указатель текущего объекта (строчный курсор) установлен на первом объекте.

Указатель может перемещаться по таблице объектов с помощью стрелок, а для постраничного перелистывания – с использованием ключей PgUp, PgDn.

В том случае, если не был определен ни один объект, то таблица будет пустой.

Объекты могут определяться двумя способами:

1) неявно, путем ввода и последующей компиляции правил с помощью опции CHECK;

2) явно, путем задания их в опции OBJECTS, при этом при следующей компиляции значение может «обнулено».

Операций, которые реализованы с помощью функциональных команд:

F1 – HELP, вызов окна помощи, содержащего справочную информацию, релевантную текущему состоянию диалога;

F2 – Quit, возврат в головное меню;

F3 – FRAME, доступ к фрейму текущего объекта. Этот ключ обеспечивает автоматическую компиляцию отдельного фрейма-процедуры;

F4, F5 – обеспечивает вывод всех правил, связанных с текущим объектом. Ключ F4 выводит правила, содержащие объект в консеквенте, а F5 – правила, содержащие объект в антецеденте;

F8 – NAME, используется для создания нового объекта и задания его имени;

F9 – CHECK, компилирует выделенную процедуру или набор правил по шагам;

F10 – GOTO, устанавливает указатель (строчный курсор) на объект, заданный с помощью имени или номера или, в зависимости от параметра, выполняет переход к следующему объекту указанного типа.

Команда F10 может выполняться со следующими параметрами:

/p – переход к следующему объекту процедуре;

/r – переход к следующему числовому объекту;

/t – переход к следующему текстовому объекту;

/l – переход к следующему списковому объекту;

/o – вызов типов объектов.

10.1.7 Классы и наследование свойств

Системы, основанные на правилах – продукциях, предоставляют, как правило, достаточно мощные средства для работы с так называемыми каузальными знаниями (причина-следствие). Этот тип знаний позволяет выполнить логический вывод на основе некоторых уже установленных фактов или выполнять некоторые действия, если сложилась определенная ситуация. Однако для представления объектных знаний (знаний о свойствах объектов и связей между ними) каузальные зависимости подходят в меньшей степени.

Представление объектных знаний в виде правил обычно влечет за собой необходимость включения в главный набор всех правил для каждого свойства каждого объекта и для каждой связи объекта с другими объектами. Легко видеть, что в этом случае размер главного набора правил может быть значительным, а при увеличении числа значений объектов – быстро увеличиваться, и с такой базой знаний будет трудно работать. По сути дела необходимо написать столько правил, сколько сочетаний значений объектов имеется.

Более подходящий способ заключается в понимании того, что объекты могут быть сущностями некоторого общего класса объектов. В этом случае атрибуты каждого объекта могут храниться во фрейме соответствующей сущности, а для поиска во всем классе объектов в главном наборе правил достаточно иметь единственное правило. В целом, фреймы, объекты-классы и объекты-члены обеспечивают более компактный и обозримый метод обработки объектных знаний.

Определение объектов типа КЛАСС

Для того чтобы создать объект с типом КЛАСС, необходимо в режиме OBJECTS нажать ключ F8, указать имя этого объекта, а затем выбрать соответствующий пункт в меню, предлагаемом системой.

В объекте типа КЛАСС предусмотрены следующие слоты: Members: и Member Slots:.

Members:

Указываются имена всех членов класса, разделенные символом запятая «,».

MemberSlots:

Указываются атрибуты класса как новые слоты, имена которых определяет разработчик прикладной ЭС. Каждое имя слота должно заканчиваться символом двоеточие «:». При желании в слот может быть записано значение. Эти значения будут наследоваться членами класса, если их собственные фреймы не означены.

После окончания редактирования фрейма-класса необходимо последовательно нажать ключи F2 и F9, после чего система выполнит генерацию (или модификацию) фреймов членов данного класса. Объекты и фреймы создаются автоматически, и в окне на экране высвечивается процесс

их генерации. Далее объекты, члены класса, могут редактироваться обычным образом.

Фреймы членов класса содержат слот **IsA:**, который работает как указатель на фрейм класса, из которого соответствующий член класса был сгенерирован, а также копии всех слотов, следующих за MemberSlots: во фрейме класса, но без значений. При необходимости разработчик может ввести нужные значения, но если никакие значения не введены, то они будут унаследованы от объекта-класса, когда это будет необходимо.

Использование объектов классов в главном наборе правил

Если в базе правил выполняется обращение к объекту типа класс, то оно должно начинаться фразой:

for all <класс> или **for some** <класс>,

Квантификатор for all генерирует поиск по всем членам класса.

Квантификатор for some заканчивает поиск после того, как один из членов класса удовлетворяет клаузы правила.

В квалифицированных правилах, в тех местах, где в нормальном синтаксисе разрешено использовать имя объекта, разрешено использовать фразу:

<ИмяСлота:> **of** <ОбъектКласс>.

Двоеточие в конце имени слота является неотъемлемой частью синтаксиса и всегда должно присутствовать. Как и в случае обычных объектов, референты слотов могут быть типа Real, Text или List, и при ссылке на них в правилах может использоваться соответствующий синтаксис.

Фреймы объектов должны содержать в слотах QueryPrompt: и AllowedValue: текст вопроса о значении слотов и возможные варианты ответов.

10.1.8 Создание и запуск прикладных экспертных систем в системе LEONARDO

Рекомендуемый порядок работы с системой совпадает с последовательностью пунктов главного меню. Сначала производится первичный набор правил (или считывание из файла), затем компиляция (для создания простых объектов) и потом редактирование фреймов и процедур.

При создании или редактировании базы знаний составляющие ее правила, объекты и процедуры находятся в текстовом формате и не могут исполняться. Прежде чем появится возможность исполнять базу знаний, необходимо выполнить опцию CHECK, которая реализует две функции:

1) преобразует базу знаний из исходного текстового формата в исполняемый код;

2) выполняет проверку синтаксиса правил, процедур и объектов.

При попытке исполнить базу знаний, которая не была скомпилирована или редактировалась после компиляции, система выдаст сообщение об ошибке. В том случае, если база знаний была скомпилирована и сохранена в файле, то для ее исполнения повторная компиляция не требуется.

Система LEONARDO компилирует и проверяет синтаксис базы знаний в следующей последовательности. Во-первых, компилируется главный набор правил и осуществляется идентификация объектов. Во-вторых, компилируются все объекты с фреймами, содержащими наборы правил.

В последнюю очередь осуществляется компиляция процедур. Система при компиляции всегда придерживается указанной выше последовательности.

Существенно, что фреймы объектов, не содержащие наборы правил в слоте RuleSet:, система не компилирует. Поэтому при их модификации выполнение опции CHECK не требуется.

Чаще всего при компиляции базы знаний применяется опция CHECK-SUMMARY. Это наиболее быстрый метод, так как при этом не возникает остановок, за исключением случаев, когда система обнаруживает синтаксические ошибки. В этом режиме на экране постоянно высвечивается информация о том, что именно компилирует система в текущий момент – правила, объекты или процедуры. При этом высвечиваются номера предложений в компилируемой процедуре.

Если система находит синтаксическую ошибку, то компиляция прерывается и на экран выводится окно с сообщением об ошибке. Для продолжения компиляции необходимо нажать клавишу Esc, после чего система выводит текст того предложения, в котором обнаружена ошибка. Разработчик прикладной ЭС должен прервать компиляцию, внести исправление и повторно выполнить опцию CHECK-SUMMARY. В противном случае могут возникнуть так называемые «наведенные ошибки», что обычно и бывает в большинстве существующих компиляторов.

Компиляция в системе LEONARDO аналогична компиляции обычных (традиционных) программ и поэтому не исключает появления в базе знаний логических ошибок.

В процессе исполнения базы знаний существует возможность вмешательства со стороны разработчика (пользователя) для получения или ввода дополнительной информации. Эти возможности реализованы в виде следующих функциональных клавиш:

F1 – Help	F6 – BackUp
F2 – Quit	F7 – Expand
F3 – Why?	F8 – Review
F4 – How?	F9 – Log
F5 – Volunteer	

Функции вмешательства являются контекстно-зависимыми, поэтому в любой момент система выводит на экран в виде динамической подсказки только те из них, которые релевантны текущему состоянию диалога.

Дадим характеристику этим возможностям.

F1 – выдает обычную подсказку.

F2 – прерывает процесс консультирования и возвращает в главное меню.

F3 – эта функция вмешательства позволяет пользователю спросить у прикладной ЭС: «Почему задан данный вопрос?». Система задает пользователю вопрос, когда необходимо означить объект, значение которого не может быть выведено. На экране появляется сообщение следующего вида:

Применено правило

if **объект1** is ...
than **объект2** is ...
для выяснения значения **объект1**, а
значение объекта **объект2**
не известно, вывести его не удается,

где объект1 – это объект, значение которого запрашивается у пользователя.

Если активизировать функцию Why? (F3) повторно, то система выполнит трассировку пути рассуждения от текущего вопроса в направлении к цели. Последовательное использование клавиши F3 в итоге приведет к концу бэктрекинга и на экране в ответ на вопрос Why? появится сообщение:

«Это текущая цель базы знаний».

Функция Why? доступна в тот момент, когда система задает пользователю вопрос во время консультации.

F4 – эта функция позволяет объяснить How? было получено значение объекта, т. е., как был получен этот результат.

На экране появляется следующее сообщение (такого вида):

Для вывода того, что **Объект** был **ЗначениеОбъекта**
я использовала правило
if ...
then **Объект** is **ЗначениеОбъекта**.

Возврат к консультации выполняется с помощью клавиши F2. Клавиша F4 может быть нажата повторно, в результате чего на экран будет выведен в виде меню список объектов, упомянутых в антецеденте правила и их текущие значения. Выбор объекта осуществляется с использованием клавиш управления курсором и ENTER. После этого на экране высвечивается правило, применявшееся для вывода значения выбранного объекта. Путь рассуждения может быть отслежен повторно с помощью клавиши PgDn, что дает возможность исследовать другую ветвь рассуждения. Функция How? доступна в тот момент, когда в процессе консультации получено заключение.

F5 – эта функция позволяет пользователю прервать естественный процесс консультирования (вопрос ЭС – ответ пользователя) и ввести значения объектов по собственному усмотрению, что дает возможность избежать обратного вывода значений некоторых объектов, который в противном случае выполнялся бы. Информацию, введенную таким образом, обрабатывает механизм прямого вывода. При этом в появляющееся окно выводятся все объекты, известные системе, вместе с их текущими значениями, если они уже есть. Для выбора нужного объекта используются клавиши курсора и ENTER. В том случае, если в соответствующем слоте фрейма объекта зарезервирован вопрос, то он выводится в стандартном виде и пользователь выбирает или

вводит значение объекта. В противном случае система (по умолчанию) генерирует свой собственный вопрос. При одном использовании команды F5 могут быть введены значения нескольких объектов. Возврат к стандартному процессу консультирования осуществляется с использованием клавиши F2.

F6 – функция позволяет изменить ответ на один из ранее заданных системой вопросов в режиме консультирования. При этом верхнее (диалоговое) окно расширяется на весь экран, а последний вопрос выделен. Далее необходимо с помощью курсорных клавиш переместить курсор на тот объект, значение которого необходимо изменить, и нажать ENTER. После ввода нового значения система работает также, как и в режиме Volunteer (F5).

F7 – функция высвечивает информацию, содержащуюся в слоте Expansion: фрейма текущего объекта. Существенно, что функция работает в любом случае, даже когда упомянутый слот фрейма текущего объекта не заполнен. При этом выдается стандартное сообщение.

F8 – эта функция позволяет вывести на экран и просмотреть протокол всего диалога текущей консультации. Если протокол не умещается на весь экран, то возможен скроллинг экрана.

F9 – эта функция позволяет сохранить в файле протокол текущей консультации. При этом система запрашивает имя DOS-файла, включая драйвер, путь и расширение, в котором должен быть сохранен протокол, а также предлагает отредактировать и внести в протокол необходимые комментарии.

При создании базы правил может понадобиться проверка логики ее работы. Система LEONARDO предоставляет два средства трассировки исполнения базы правил – в **прямом** и **обратном** направлениях.

Трассировка базы правил в **прямом** направлении осуществляется посредством активизации опции EXECUTE-FORWARDS Этот режим работы предназначен для исследования в базе правил отдельных цепочек прямого вывода. Для работы в таком режиме необходимо, чтобы в базе правил было хотя бы одно правило типа **ask** <**объект**>. После ввода значения объекта база правил исполняется в прямом направлении до тех пор, пока не сработают все правила. После этого используется следующее **ask**-правило, и прямой вывод повторяется до тех пор, пока не будут заданы все вопросы и не сработают все правила. При работе в таком режиме разработчик прикладной ЭС может исследовать значения всех объектов и то, как они были получены. Часто при тестировании в прямом направлении полезно помещать в начале базы правил фиктивное **ask**-правило для того, чтобы инициализировать диалог. Далее, в режиме НАЗНАЧИТЬ (F5), можно ввести различные значения объектов, а также проследить, выполняется ли требуемый ход рассуждения. Для получения дополнительной информации в ходе тестирования могут быть использованы другие функции вмешательства.

Трассировка базы правил в **обратном** направлении предназначена для исследования и проверки отдельных цепочек в базе правил. Система предлагает обозначить цель в терминах имени и значения объекта. После этого

база правил отрабатывается в обратном направлении для того, чтобы проверить заданную цель. При этом любые процедуры, существующие в базе объектов или базе правил, выполняются как обычно. Для отслеживания трассы могут применяться имеющиеся в системе LEONARDO команды вмешательства: Why?(F3), How?(F4) и т. д.

10.2 Процедурные фреймы

Синтаксис главного набора правил ограничивает возможности разработчика прикладной ЭС как в части организации интерфейса с пользователем, так и в представлении знаний. Например, в антецеденте правила нельзя использовать скобки для представления необходимых логических выражений, а в консеквенте правила предусмотрена возможность применения лишь конъюнктивных связок. Указанные ограничения не являются абсолютными, поскольку процедурный язык системы LEONARDO предоставляет большие возможности не только по реализации сложных логических выражений, но и ряд других функций.

Процедуры являются модулями исполняемого кода, записанного на языке программирования системы LEONARDO. Они распознаются компилятором по ключевому слову run, стоящему перед ними. Объект типа процедура создается системой LEONARDO при компиляции главного набора правил. В процессе компиляции выполняется проверка этой процедуры, система, естественно, ничего, кроме имени, в ней не находит и поэтому выдает сообщение об ошибке. После этого разработчик прикладной ЭС должен войти в режим OBJECTS, выбрать объект, соответствующий процедуре, и отредактировать его фрейм, т. е. заполнить слоты.

Процедура вызывается из правил с помощью ключевого слова run. Например, правило:

if ‘Собака будет служить для’ **is** ‘охрана помещений’
and ‘Вы предпочитаете чтобы собака была’ **is** ‘злая’
and height > 0
then Собака **is** ‘Доберман-пинчер’;
height_this_dog = 1500;
run *check* (height, height_this_dog, Собака);
Result **is** done

вызывает выполнение процедуры *check*.

Каждый параметр процедуры – полноправный объект базы знаний. При этом параметры типизированы (Real, Text, List). Если параметры уже употреблялись где-нибудь в базе правил, то они будут сохранять свой прежний тип. Если же это первое появление объекта в базе правил, то он будет создан по умолчанию как объект типа undefined. Напомним еще раз, что вызов процедуры не может быть последним консеквентом в правиле. В нашем случае – это клауза «Result is done», отделенная от предыдущей клаузы символом «;». Правила, в которых выполняются вызовы процедур, не отличаются от любых других правил. Антецедент правила должен быть удовлетворен прежде, чем сработает его консеквент.

Процедуры не могут быть активизированы и исполняться самостоятельно, без антецедента в правиле. Исключение составляет специальный случай присваивания объекту вычисляемого значения в стандартном слоте ComputeValue: фрейма объекта.

10.2.1 Слоты процедурного фрейма

Фрейм объекта типа процедуры (или процедурный фрейм) имеет слоты, которые отличаются от слотов фреймов объектов.

Процедурный фрейм, созданный системой LEONARDO для объекта типа процедура, имеет следующий вид:

1: **Name:**
2: **LongName:**
3: **Type:**
4: **AcceptsReal:**
5: **AcceptsText:**
6: **AcceptsList:**
7: **ReturnsReal:**
8: **ReturnsText:**
9: **RetumsList:**
10: **LocalReal:**
11: **LocalText:**
12: **LocalList:**
13: **Externals:**
14: **Body:**

Name: (защищенный слот)

Слот содержит имя процедуры, указанное разработчиком прикладной ЭС при создании главного набора правил базы после ключевого слова **run.**

LongName:

Слот содержит длинное имя объекта.

Type: (защищенный слот)

Слот определяет тип объекта, т. е. Procedure.

AcceptsReal:

AcceptsText:

AcceptsList:

Эти слоты предназначены для определения входных параметров процедуры типа Real, Text и List соответственно. Значения переменных в этих слотах в процедуре модифицироваться не могут.

ReturnsReal:

RetumsText:

ReturnsList:

Эти слоты предназначены для описания выходных параметров процедуры типа Real, Text и List соответственно.

LocalReal:

LocalText:

LocalList:

Эти три слота предназначены для определения локальных по отношению к процедуре объектов.

Externals:

Слот позволяет определять внешние процедуры, которые могут быть вызваны из данной процедуры.

Body:

Слот маркирует начало тела процедуры.

Таким образом, слоты процедурного фрейма предназначены для описания входных и выходных параметров процедуры, а также для определения локальных объектов.

Слоты параметров и передача параметров

Система LEONARDO должна «понимать», какие из передаваемых процедуре параметров могут быть изменены внутри процедуры, а какие только передают в процедуру информацию. Причина этого кроется в следующем. Когда при консультации с прикладной ЭС используются средства **бэктрекинга** (F6), то объекты, означенные в процессе логического вывода, должны быть восстановлены со своими первоначальными значениями. Когда в цепочке логического вывода происходит вызов процедур, системе необходимо знать,

какие объекты могут измениться в результате выполнения процедуры, для того чтобы восстановить их значения при бэктрекинге.

Эту возможность обеспечивает структура списка параметров фрейма процедуры. Объекты-параметры, находящиеся в слотах «AcceptsXxxx:», не могут модифицироваться в процедуре (они передаются с помощью механизма «передача по значению»), тогда как параметры в слотах типа «ReturnsXxxx:» могут быть изменены в теле процедуры (они передаются с помощью механизма «передача с помощью ссылки»).

Таким образом, при написании правила, которое будет вызывать процедуру, необходимо отчетливо понимать тип и статус объектов, передаваемых в виде параметров.

Структура списка параметров при обращении к процедуре должна быть следующей: первыми в списке параметров должны обязательно стоять те объекты, которые не могут модифицироваться в процедуре; вторыми – которые могут модифицироваться. Внутри каждой из этих групп объекты типа Real должны стоять первыми, за ними следуют объекты типа Text и, наконец, объекты типа List.

Передача параметров в процедуру может быть выполнена с использованием команды **GLOBAL**, которая делает глобальные объекты (объекты главного набора правил) доступными в процедуре только для чтения. При этом имена объектов в главном наборе правил и процедуре должны совпадать. Такая техника позволяет исключить необходимость заполнения слотов типа Acceptsxxxx: во фрейме-процедуре.

Если команда GLOBAL помещена в начале процедуры (после слота Body:), то процедуре становятся доступными значения всех объектов, но при этом они не могут модифицироваться. Естественно, что с использованием оператора присваивания локальным объектам соответствующего типа глобальные объекты станут доступными для модификации, но в главном наборе правил изменения значений объектов не произойдет. Например, с использованием команды GLOBAL в процедуре можно вывести на экран (печать) некоторую информацию, не передавая объекты через список параметров. В том случае, если в процедуре имеется необходимость модификации всех объектов, указанных в главном наборе правил, то в любом месте главного набора правил необходимо написать команду **CONTROL COMMON**. Такая техника, естественно, не требует указания команды GLOBAL во фрейме-процедуре.

Локальные переменные

Переменные, которые используются только внутри процедуры, называются локальными и определяются в слотах LocalXxxx. Эти переменные не появляются в глобальном списке объектов (опция ОБЪЕКТЫ) и существуют

только во время исполнения процедуры. Между вызовами процедуры значения локальных переменных не сохраняются.

Локальные переменные задаются в виде списка их имен, разделенных запятыми. Список может занимать столько строк, сколько необходимо разработчику прикладной ЭС. Если при декларировании локальных переменных используется более одной строки, то каждая из них, за исключением последней, должна заканчиваться запятой. Например:

10: **LocalReal:** al, b2
11: **LocalText:** поле, Name
12: **LocalList:** список

Числовые и текстовые локальные переменные могут быть декларированы в виде массивов произвольной размерности, которые определяются с помощью индексов. Размерность массивов может достигать пяти (5). Так, например, слоты:

10: **LocalReal:** MAC (5,30), Vector (l00)
11: **LocalText:** Таблица (2,105)

определяют двухмерный числовой массив MAC с размерностью пять на шесть; одномерную переменную Vector на сто чисел; двухмерную текстовую переменную Таблица, размером два на сто пять элементов. Заметим, что каждый элемент текстового массива имеет произвольную длину.

Обращение к элементам массива выполняется традиционно, как и в других языках программирования, с помощью индексов. В примере:

MAC (i,j)= Vector (K-l)*(5 + n)

элементу массива MAC с индексами i, j будет присвоено (к-1) значение массива Vector, умноженное на (5+n).

Текст процедуры записывается на процедурном языке системы LEONARDO в следующей строке после слота Body: .

Существенно, что суммарное число локальных переменных не должно превышать (с учетом элементов массивов) одну тысячу.

10.2.2 Процедурный язык

Процедурный язык системы LEONARDO обладает возможностями, аналогичными алгоритмическим языкам программирования высокого уровня. Предложения на процедурном языке системы записываются в любом формате, но по умолчанию предполагается, что каждое из них будет занимать одну строку. Если предложение процедуры не помещается в одной строке, то оно может быть записано в любом количестве строк. При этом для индикации продолжения строки используется символ «&» в качестве последнего отличного от пробела символа «переполняющейся строки».

Для улучшения читаемости текста можно использовать пустые строки или комментарии, которые должны начинаться с комбинации символов «/*».

Числовые переменные обрабатываются также, как и в большинстве процедурных языков. Имеются следующие стандартные операции:

+ сложение;

- вычитание или унарный минус;

* умножение;

/ деление;

^ возведение в степень.

Стандартное старшинство операций имеет следующий порядок: ^; *; /; +; -. Для изменения стандартного старшинства операций могут применяться скобки. Примеры типичных предложений присваивания, в том числе и с вычислительными выражениями:

TextPerem = ‘произвольный текст’

Выручка = ОбъемПродажи * (Цена - Скидка)

Объем = (4 / 3) * Пи * Радиус^3

В предложениях процедурного языка системы LEONARDO могут модифицироваться только объекты типа LocalXxxx и ReturasXxxx. Попытки модифицировать объекты типа AcceptsXxxx при компиляции выявлены не будут, а при исполнении приведут к ошибке.

Обработка текстовых строк

В системе LEONARDO обеспечена возможность манипулирования текстовыми объектами и текстовыми константами. Текстовые константы определяются с помощью одинарных кавычек:

ТекстПерем = ‘строка текста’

Конкатенация выполняется с помощью оператора “ + ”:

FullName = name (1,1)+name (2,1)

Собака = ‘кавказская’ + ‘ ’ + ‘овчарка’ (кавказская овчарка)

Далее перечислены функции, обеспечивающие дополнительные возможности обработки текстовых строк.

Len (текстовой объект)

Эта функция с одним аргументом возвращает число символов текстового объекта. Например, совокупность предложений:

...

ТекстОбъект = ‘ПГУПС’

ДлинаИмени = Len (ТекстОбъект)

...

присвоит объекту ДлинаИмени значение, равное числу пять.

Location (подстрока, строка)

Эта функция с двумя аргументами типа Text возвращает позицию начала подстроки в строке или «0», если строка не содержит подстроки. Предположим, что Позиция – это объект типа Real, а Текст – текстовой объект, содержащий значение: «абвгдежзиклмнопрст...». Тогда предложение

Позиция = Location (‘где’, Текст)

присвоит переменной Позиция значение четыре,

Позиция = Location (‘гдж’, Текст)

присвоит переменной Позиция значение 0.

mid (ИсхСтрока, НачалоПодстроки, ДлинаПодстроки)

Это функция с одним текстовым (ИсхСтрока) и двумя числовыми (НачалоПодстроки, ДлинаПодстроки) аргументами возвращает подстроку длиной ДлинаПодстроки, начинающуюся с позиции НачалоПодстроки в переменной ИсхСтрока. Например:

dddd = ‘31-Mar-98 11: 55’	(исходная строка)
day = mid (dddd, 1, 2)	(31)
month = mid (dddd, 3, 5)	(-Mar-)
year = mid (dddd, 8, 2)	(98)
time = mid (dddd, 11, 5)	(11: 55)

Uppercase(строка)

Это процедура с одним текстовым аргументом. Она преобразовывает значение аргумента, содержащего только **латинские буквы**, в верхний регистр. Например:

Образец = ‘abcde123’

Uppercase (Образец)

установит значение объекта Образец, равное ABCDE123.

Lowercase (строка)

Возвращает значение аргумента, содержащего только латинские символы, в нижнем регистре. Используется также, как и uppercase.

CHAR (ЧислОбъект)

Преобразование числа в символ. Данная функция преобразует число в диапазоне 0... 255 в односимвольную строку. Если аргумент больше 255, то преобразуются только 8 наиболее значащих битов.

CONVRT (символ, дескриптор)

Данная функция преобразует символ в число в диапазоне 0... 255. Дескриптор = I1, I2, I3, I4 для бинарного целого длины 1, 2, 3 или 4, и R4 для IEEE четырех байтового числа с плавающей запятой. Например:

Число = CONVRT (‘1’, R4)

ICHAR (символ)

Эта функция конвертирует односимвольную текстовую строку в число в диапазоне 0... 255. Если в параметре задана строка, содержащая более одного символа, то преобразуется только первый символ.

INT (ЧисловойОбъект)

Эта функция возвращает целую часть числа. Например,

ЧисловойОбъект = 10. 25

ЧисОБ = INT (ЧисловойОбъект)

значение переменной ЧисОб будет равно 10.

LTRIM (ТекстОбъект)

Эта функция с одним параметром отсекает ведущие пробелы в символьной строке. Например,

ТекстОбъект = ‘ текст’,

ТекстОбъект1 = LTRIM (ТекстОбъект)

значение ТекстОбъект1 будет равно ‘текст’

RTRIM (ТекстОБ)

Это функция с одним параметром отсекает завершающие пробелы в символьной строке.

STRIP (ТекстОБ)

Функция с одним параметром отсекает ведущие и завершающие пробелы в символьной строке.

Обработка списковых объектов

CONCAT (аргум1, аргум2)

Эта функция с двумя аргументами выполняет их конкатенацию, причем первый аргумент обязательно должен быть списковым объектом, а второй аргумент может быть либо списковым объектом, либо текстовым объектом. Например:

СтекОбъект = CONCAT (Список1, Список2)

дает возможность получить СтекОбъект, значениями которого будут все значения списковых объектов Список1 и Список2, даже если у этих объектов имеются повторяющиеся значения.

ELEM (ЧислОб, СписокОб)

Функция с двумя аргументами, возвращающая элемент списка с номером, заданным первым аргументом. Пусть списковый объект СписокОб имеет значения: лето, зима, осень. Тогда

ТекстОб =ELEM (3,СписокОБ) присвоит объекту ТекстОБ значение ‘осень’.

FIRST (СписокОБ)

Эта функция с одним аргументом позволяет извлечь первый элемент списка.

NELEM (СписокОб)

Функция с одним аргументом, позволяющая определить число элементов в списке.

REST (СписокОб)

Функция обеспечивает возврат списка с удаленным первым элементом.

SET (< функциональный код>, список1, список2)

Эта функция выполняет над списками операции как над множествами. Значение функционального кода определяет операцию, выполняемую над двумя списковыми аргументами:

1 – объединение множеств;

2 – пересечение множеств;

3 – вычитание множеств (список2/список1);

4 – возвращает множество, не содержащееся в список1 и список2

Предложения управления вводом и выводом

Вывод на экран и ввод с клавиатуры выполняются с помощью перечисленных ниже процедур и функций.

AS

Форматирование чисел. Эта функция задает формат преобразования числа в символьное представление. Например:

ТекстовойОбъект = ЧисловойОбъект **as** дескриптор

или

ТекстовойОбъект = ЧисловойОбъект **as** ‘xx...x. x...x’,

где дескриптор = I1, I2,I3, I4 для бинарного целого длины 1, 2, 3 или 4, и R4 – для IEEE четырехбайтового числа с плавающей запятой; ЧисловойОбъект – число или числовой объект; ‘xx...x. x...x’ – формат шаблона для вывода.

Предположим, что числовой объект ЧисловойОбъект содержит значение 123,45. Тогда выражение ТекстовойОбъект = Числовой Объект as ‘xxx.x’ присвоит текстовому объекту строку длиной пять символов 123.4 .

FINPUTC

Эта функция читает вход с клавиатуры и возвращает код терминатора, используемого для завершения ввода:

КодЗавершения = FINPUTC (объект, длина),

где объект – числовой или текстовой объект, которому должно быть присвоено значение; длина – максимальное число вводимых символов; КодЗавершения – числовой объект, в который будет помещено значение числового кода завершающего символа.

Функция осуществляет проверку ввода. Если задан числовой объект, то нечисловые символы (исключая символ ‘.’) отвергаются и система выдает сообщение об ошибке. Попытка ввести больше символов, чем задано параметром «длина», вызовет звуковой сигнал.

Нажатие **ENTER** без ввода каких-либо символов, будет означать ввод символьной строки нулевой длины для текстовых объектов и 0.0 – для числовых объектов. Однако если объект уже содержит какое-либо значение, то оно будет сохранено.

AT

Процедура выводит на экран список выражений в заданной позиции:

AT (строка,столбец,<список выражений>),

где строка – номер строки (числовой объект), в которой будет начинаться вывод на экран <список выражений>; столбец – аналогично для номера столбца; <список выражений> – переменное число аргументов любого типа.

Предположим, что объект ПородаСобаки имеет значение ‘Овчарка’ и РостСобаки – значение 120.

Тогда:

AT (10, 3, ‘Порода собаки’, ПородаСобаки, ‘и ее рост’ &, РостСобаки as ‘xx’) выведет на экран, начиная с 3-й позиции 10-й строки, следующее сообщение:

Порода собаки Овчарка и ее рост 120.

Аналогично:

AT (10, 3, ‘Сумма 4 и 7 равна’,4+7 as ‘xx.xx’)

будет выведено в следующем виде:

Сумма 4 и 7 равна 11.00.

ATR

Процедура обеспечивает вывод на экран списка выражений в заданной позиции без перехода к новой строке. Описание параметров аналогично AT.

formprint (ТекстОбъект)

Процедура с одним текстовым аргументом, которая обеспечивает вывод на принтер заданного отчета. Фрейм объекта ТекстОбъект, заданного в аргументе, должен включать слот Form:, содержащий форму отчета.

iprint (список выражений)

Процедура с переменным числом аргументов любого типа печатает список аргументов на принтере.

channel (номер принтера)

Процедура устанавливает канал вывода на принтер для использования в процедурах **formprint** и **iprint**. Разрешенные значения аргумента – 0 (по умолчанию) для LPT1, 1 – для LPT2 и т. д.

Кроме рассмотренных выше средств ввода/вывода в системе LEONARDO имеются следующие процедуры и функции:

INPUT

Это псевдопеременная, которая при исполнении замещается на сообщение, введенное с клавиатуры. Например, выражение **answer = INPUT** присвоит переменной **answer** цепочку символов, введенную с клавиатуры. В том случае,

если объект **answer** числовой, то символы будут проверяться на соответствие числовому типу.

FINPUT (объект, длина)

Процедура с двумя параметрами, где <объект> – объект, которому должно быть присвоено значение, <длина> – максимальное количество вводимых символов. Процедура FINPUT получает отклик пользователя с клавиатуры и выполняет проверку ввода. Если задан числовой объект, то нечисловые символы отвергаются и процедура будет вновь запрашивать значение объекта до тех пор, пока не будет введено число. Попытки ввести больше символов, чем указано в параметре <длина>, вызовут звуковой сигнал. Ключ Esc обновляет поле, а F2 – приводит к прерыванию консультации пользователя с экспертной системой. Нулевой отклик реализуется аналогично функции **FINPUTC**.

PRINT (список-выражений)

Это процедура с произвольным количеством аргументов любого типа. Она выводит данные на экран, начиная с текущего положения курсора. Повторное обращение к процедуре приводит к автоматическому переходу на новую строку. При этом изображение на экране смещается вверх аналогично тому, как это происходит с командами MS DOS при работе в этой системе.

PROMPT (список-выражений)

Эта процедура выводит на экран данные без перехода к новой строке.

Процедурные средства работы с экраном.

В процедурном языке системы LEONARDO существует ряд средств, которые дополнительно могут использоваться для создания адаптированных к пользователю экранов. Эти средства могут применяться для построения модулей, обеспечивающих ввод информации в виде «заполненных форм». При этом пользователь может вводить одновременно несколько связанных элементов информации.

Упомянутые средства делятся на три группы:

1) определение атрибутов всего экрана;

2) определение атрибутов для секций экрана;

3) формирование рамок вокруг областей экрана.

box (ВерхняяСтрока, ЛеваяКолонка, НижняяСтрока, ПраваяКолонка, Атрибут [, ФлагЗаполнения])

Процедура с пятью числовыми аргументами (числами или числовыми объектами) и шестым необязательным аргументом рисует на экране прямоугольную рамку. Первые четыре аргумента – координаты углов рамки (25*80). Пятый аргумент – цветовой атрибут рамки, значение которого находится в диапазоне 0 ... 255. Шестой аргумент – флаг, указывающий, будет ли заполняться рамка или нет. Он необязательный. Если флаг заполнения равен

0, то внутренняя область, ограниченная рамкой, остается без изменения. Если флаг равен 1, то область заполняется пробелами цвета, заданного в цветовом атрибуте. Наиболее простой путь выбора атрибута – с помощью опции UTILITES-ATTRIBUTE.

colour (ВерхняяСтрока, ЛеваяКолонка, НижняяСтрока, ПраваяКолонка, Атрибут)

Процедура с пятью числовыми аргументами. Устанавливает цветовой атрибут прямоугольной зоны экрана. Аргументы имеют такой же смысл, как и в процедуре box.

curloc (строка, столбец)

Процедура с двумя числовыми аргументами (числовыми объектами) устанавливает курсор в указанную позицию экрана.

cursor (ТипКурсора)

Процедура с одним числовым аргументом. Она устанавливает тип курсора, который может принимать следующие значения:

0 – погасить курсор (убрать с экрана);

1 – стандартный курсор (подчеркивание);

2 – курсор размером в половину символьной ячейки экрана;

3 – курсор размером в полную символьную ячейку экрана.

hold

Процедура без аргументов. Она используется для фиксации изображения на экране до тех пор, пока пользователь не нажмет какую-либо клавишу. При этом в 25-й строке экрана появится сообщение «Press any key».

screen (атрибут)

Процедура с одним числовым аргументом. Устанавливает цветовой атрибут всего экрана.

date

Это псевдопеременная, возвращающая текущее значение даты и времени в формате DOS. dd-mmm-yy hh:mm. Например, ТекстОб = date. Значение ТекстОб будет равно, скажем, 1-Apr-12 13:09.

Предложения передачи управления, итераций и условные предложения

В ходе выполнения процедуры управление может передаваться условно или безусловно с использованием предложения **goto** метка:. Любое предложение в процедуре может быть помечено меткой, за исключением особо оговариваемых случаев. Метка – это цепочка символов, заканчивающихся двоеточием. Например, **goto начало:**.

Циклы программируются с помощью предложения **repeat**, которое имеет три альтернативные формы:

1) **repeat while** <условие>

...

endrep

2) **repeat until** <условие>

...

endrep

3) **repeat** <индекс> (<начало>, <окончание>, [<шаг>])

...

endrep

Первые две формы логически эквивалентны. Любая из них может использоваться для выражения одного и того же условия окончания цикла. Выбор формы зависит от того, какой из двух синтаксисов лучше подходит в конкретном случае.

Для формата **repeat while** <условие>, если условие выполнено (true), то управление передается следующему предложению в теле цикла. В противном случае, если условие не выполнено (false), то осуществляется выход из цикла, т. е. управление передается предложению, следующему за **endrep**.

Для формата **repeat until** <условие>, если условие не выполнено, то управление передается следующему предложению в теле цикла. В противном случае – предложению, следующему за **endrep**.

Для формата **repeat** <индекс> (<начало>, <окончание>[,<шаг>]) имеет место следующее:

индекс – числовой объект (типа LocalReal) – параметр цикла;

начало и окончание – произвольные числовые выражения, представляющие начальное и конечное значения индекса (параметра) цикла;

шаг – произвольное необязательное выражение, оцениваемое как число и определяющее шаг изменения значения индекса на каждой итерации выполнения тела цикла. Значение шага может быть как положительным, так и отрицательным. По умолчанию значение шага равно 1.

Цикл начинается со значения индекса, равного значению <начало> и повторяется до тех пор, пока <индекс>, итеративно изменяющийся на величину <шаг>, не станет больше (или меньше), чем значение <окончание>. Каждое предложение цикла любого из трех типов должно заканчиваться словом endrep. **repeat и endrep не могут быть помечены в процедуре метками.**

Для условных предложений в системе LEONARDO принят следующий основной синтаксис:

if <условие> **then** ...

...

else ...

...

endif

Клауза else должна быть размещена обособленно в отдельной строке.

В условных предложениях допускаются следующие операторы сравнения:

eq равно;

ne не равно;

gt больше, чем;

ge не меньше;

lt меньше, чем;

le не больше.

Условные клаузы могут быть связаны логическими операторами:

and конъюнкция (и);

or дизъюнкция (или).

Примеры условных предложений:

```
if x gt 30 then y = ( x - 20 )/4
```

или

```
if x gt 0 and y lt 40 then x1 = 1
   x2 = 2
   else
   x1 = 2
   x2 = l
   End if
```

Любая процедура может вызывать другую процедуру с использованием предложения: run <имя-процедуры> (<список-параметров>). Предложение run может быть правой частью условного предложения, например:

if x gt y than run PROC (x, y, z).

Правила передачи параметров при вызове вложенных процедур такие же, как и правила вызова процедур в наборе правил слота RulSet.

Все процедуры обязательно должны заканчиваться предложением **return.** Причем это предложение может быть записано ровно столько раз, сколько необходимо разработчику прикладной ЭС.

Таким образом, структура фрейма-процедуры имеет следующий вид:

< слоты процедурного фрейма >

Body: <маркер начала процедуры>

...

< операторы, предложения процедурного языка >
...
return < маркер конца процедуры >

10.3 Работа с внешними файлами

Для накопления промежуточных данных, хранения наборов нормативных показателей в системе LEONARDO предусмотрены средства для работы с файлами последовательного, прямого доступа и байтовыми файлами. Внешние файлы должны открываться для чтения или записи с помощью одной из команд openseq, opendir, openbst и обязательно закрываться с помощью команды close(unit), где unit – соответствующая числовая переменная или константа.

10.3.1 Последовательные файлы

Для работы с последовательными файлами используются следующие операторы (команды): **openseq, readseq, writeseq**.

openseq (устройство, имя, доступ)

Эта команда открывает файл для последовательного доступа. Устройство – числовая переменная или константа: 1, 2, 3,... , идентифицирующая номер файла для последующего использования в команде close. Имя – имя открываемого файла, записанное в кавычках. Доступ: 1 – для чтения; 2 – для записи. Например предложения:

openseq (1, 'a:\dir\file.seq', 1)
...
close (1)

открывают и закрывают файл последовательного доступа с номером 1 и именем file.seq для чтения на устройстве a: в директории \dir . В том случае, если файл размещен в том же каталоге, что система LEONARDO, то достаточно указать только имя файла без указания пути.

readseq (устройство, буфер)

Процедура последовательного чтения из файла. Устройство – числовая переменная или константа, указанная в команде openseq. Буфер – текстовая переменная, в которую считывается запись.

writrseq (устройство, буфер)

Процедура последовательной записи в файл. Устройство – числовая переменная или константа, указанная в команде openseq. Буфер – текстовая переменная, в которую считывается запись.

Обработка метки конца файла в последовательных файлах осуществляется следующим образом. При попытке чтения записи за пределами файла в буфер помещается строка вида **<<<EOF>>>**, наличие которой должно всегда проверяться сразу же после завершения работы оператора readseq. Например:

```
   ...
openseq( 1 ,'demo.seq', 1)
repeat i (1,30)
        readseq (1, buffer)
        if buffer eq '<<<EOF>>>' that goto метка:
        lengths=len (buffer)
        array(i)=mid (buffer, 1 lengths)
endrep
метка:
   ...
```

Предложения

```
lengths = len (buffer)
array (i) = mid (buffer, 1 ,lengths)
```

могут быть заменены на одно

```
array (i) = mid (buffer,l, len(buffer)).
```

Байтовые файлы

Байтовые файлы в системе LEONARDO являются неструктурированными и могут рассматриваться как последовательность байтов. Доступ к таким файлам осуществляется с помощью операторов (команд): **openbst, readbst, writebst** и **position**.

openbst (устойство, имя, доступ)

Открывает файл как байтовый. Смысл аргументов аналогичен процедуре openseq.

readbst (Устройство, Буфф, КолБайт)

Читает фрагмент длиной КолБайт из файла, открытого на Устройстве, начиная с текущего указателя байтов и помещает его в Буфер.

writebst (Устройство, Буфер, КолБайт)

Пишет из Буфера КолБайт в файл, открытый на Устройстве, начиная с текущего указателя байтов файла.

position (Устройство, Позиция)

Используется для обработки файлов, открытых как байтовая цепочка. Устанавливает указатель байтов в заданную во втором аргументе позицию файла, открытого на заданном в первом аргументе устройстве. После выполнения операции чтения или записи указатель байтов файла перемещается на количество пересылаемых байтов.

10.3.2 Файлы прямого доступа

Работа с файлами прямого доступа в системе LEONARDO поддерживается следующими процедурами: **opendir, readdir** и **writedir**. Эти процедуры используются совместно с функциями **pack, unpack** и слотом **Layout:** фрейма объекта.

opendir (Устройство, Имя, ДлинаЗаписи)

Процедура открывает файл для прямого доступа. Устройство – числовая переменная или константа. Имя – имя открываемого файла. Длина Записи – размер записи в байтах.

readdir (Устройство, Буфер, НомерЗаписи)

Процедура читает запись в режиме прямого доступа. Устройство – числовая переменная или константа. Буфер – текстовая переменная (имя фрейма объекта, в котором в слоте Layout: содержится формат записи), в которую считывается запись. НомерЗаписи – номер читаемой записи (числовая переменная или константа).

writedir (Устройство, Буфер, НомерЗаписи)

Процедура записывает запись в режиме прямого доступа. Смысл параметров аналогичен команде readdir.

pack (ТекстовойОбъект)

Процедура с одним текстовым аргументом, которая используется совместно с командой writedir, но может применяться и самостоятельно. Выполняет конкатенацию объектов и помещает результат в свой аргумент. При работе совместно с командой writedir список конкатенируемых объектов и их форматы размещаются в слоте Layout: фрейма текстового объекта, имя которого является аргументом процедуры pack. Все объекты, перечисленные в слоте Layout:, должны быть **глобальными**, т. е. должны появиться в списке параметров, передаваемых из главного набора правил (базы правил) во фрейм-процедуру с помощью оператора run.

unpack (ТекстовойОбъект)

Процедура с одним текстовым аргументом, которая используется совместно с командой readdir, но может применяться самостоятельно. Распаковывает содержимое аргумента в виде значений ряда других объектов. Используется совместно со слотом Layout: фрейма объекта, имя которого является

аргументом функции unpack. Этот слот определяет список объектов, в которые будет выполняться распаковка. Все объекты в слоте Layout: должны быть глобальными.

Layout:

Этот слот фрейма текстового объекта использует следующие дескрипторы: I1, I2, I3, I4 и R4, которые интерпретируются как бинарные целые длины 1, 2, 3, 4 и IEEE четырехбайтовое число с плавающей запятой, что обеспечивает доступ к бинарным файлам. Например, приведенный ниже слот Layout: описывает запись с бинарным целым, занимающим 2 байта, 10 байтами текста и IEEE числом с плавающей запятой:

Layout:

numl	I2
text1	10
num2	R4

Пример работы с файлами прямого доступа.

Пусть во фрейме-процедуре предполагается работа с файлом прямого доступа, размещенным на жестком диске в директории, в котором находится система LEONARDO. Имя файла NAME.TXT. Структура записи имеет вид:

	ТЕКСТ (10 символов)	ЧИСЛО (IEEE)	ТЕКСТ (10 символов)	ЧИСЛО (IEEE)
Идентификаторы полей	m1	m2	m3	m4

Представим базу правил (главный набор правил) в виде:

1: **if** work **is** go

2: **then run** proc(m2, m4,ml,m3,A); fin **is** s

3: **seek** fin

Запись в строке 1: позволяет сразу же активизировать процедуру, если в слоте Fixed-Value: фрейма объекта work будет записано: FixedValue: go. В противном случае система запросит значение объекта work.

Заполнение слотов фрейма объекта A должно быть следующим:

1: Name: A (формирует система)

...

18: Layout:

ml 10

m2 R4 (записывает разработчик ЭС)

m3 10

m4 R4

Во фрейме-процедуре должны быть заполнены следующие слоты:

1: Name: proc (формирует система)

...

```
7: ReturnsReal: m2,m4
8: ReturnsText: ml,m3,A
...
10: LocalReal: n(2), k
11: LocalText:nn(2)
...
14: Body:
15:   k=l
16:   opendir (1,'NAME.TXT',28)
17:   /* Чтение в буфер первой записи файла
18:   readdir(l, A, k)
19:   /*Распаковка записи
20:   unpack (A)
21:   /*Объекты записи стали доступными для обработки
22:   nn(l) = ml
23:   n(1) = m2
24:   nn(2) = m3
25:   n(2) = m4
26:   ...
27:   /*Операции над элементами записи файла
28:   ...
29:   /*Подготовка к записи в файл
30:   m1-nn(l)
31:   m2=n(l)
32:   m3=nn(2)
33:   m4=n(2)
34:   /* Упаковка
35:   pack(A)
36:   /*Запись в файл
37:   writedir(1,A,k)
38:   ...
39:   close(l)
40:   return
```

Если работа с файлом в процедуре-фрейме начинается с оператора readdir, то файл должен быть создан на диске в директории, в котором размещается система LEONARDO, с помощью какого-либо текстового редактора. Если файл находится в другом каталоге, то к нему должен быть указан соответствующий путь. При этом необходимо весь файл заполнить символом ноль “0”. Объем файла определяется разработчиком прикладной ЭС исходя из потребностей по хранению данных. В том случае, если работа начинается

с оператора writedir, то система LEONARDO сама создает файл в соответствии с именем файла в команде opendir.

10.4 Представление и обработка неточной информации

В системе LEONARDO реализовано два различных метода представления и обработки неопределенной информации, один из которых реализован на основе теоремы Байеса, а другой базируется на факторах уверенности.

10.4.1 Источники и типы неопределенности

Для выбора метода представления и обработки неопределенных данных необходимо, прежде всего, определить тип неопределенности и соотнести результаты анализа с возможностями системы LEONARDO.

Накопленный опыт свидетельствует, что во многих практических случаях нет необходимости использовать механизмы факторов уверенности или байесовской логики. Зачастую вполне достаточным является применение простых шкал с вербальными значениями, например, такими как «маленький», «нормальный», «большой» и т. п.

Истинная неопределенность возникает тогда, когда некоторое событие произошло, но у нас нет возможности убедиться в том, насколько велики шансы неприятных последствий этого события, хотя мы и не уверены в их проявлениях. Это как раз тот случай, когда нужны специальные методы представления и обработки неопределенности.

Байесовская логика базируется на предположении, что всегда существует хотя бы небольшая вероятность совершения того или иного события. В системе LEONARDO существует механизм неформального представления байесовской вероятности, позволяющий изменить шансы истинности финального заключения, в зависимости от истинности промежуточных заключений.

Другой вид неопределенности возникает, когда мы имеем дело

с формулировкой правила для прогнозирования некоторого заключения.

Предположим, что если утверждение A истинно, то утверждение B также должно быть истинным, т. е.

if A **then** B

Поскольку A и B являются утверждениями, то правило можно переписать

в следующем виде:

if A **is** X **then** B **is** Y

Часто мы не полностью уверены, что это правило выполняется. Например, известно, что иногда B **is** Z даже тогда, когда A **is** X.

Для выражения данного эффекта вводится понятие так называемого фактора уверенности, значение которого связано с различными значениями B при условии, что A **is** X. Это утверждение, например, может быть записано

в следующем виде:

if A **is** X **then** B **is** Y {суверенностью 0.7}

and B **is** Z {с уверенностью 0.2}

Смысл этой записи заключается в том, что если A равно X, то B будет равно Y, скажем, в 70 % времени, равно Z – в 20 % времени, а в 10 % времени B равно еще какому-то значению.

При выборе метода представления неопределенности важно понимать, в чем заключается источник этой неопределенности. Если неопределенность связана с выбором детальности описания, то целесообразно использовать список подходящих вербальных значений параметров в слоте AllowedValue: с последующим применением полностью детерминированных правил. В свою очередь, если неопределенность характеризуется множественными логическими линиями рассуждения, связанными с множественными заключениями различной уверенности от одного и того же антецедента, то наиболее подходящими являются факторы уверенности.

С другой стороны, если неопределенность характеризуется факторами, оказывающими положительные и отрицательные эффекты на вероятность истинности заключений, то целесообразно использовать механизм байесовского вывода.

В системе LEONARDO имеется несколько управляющих, типа **CONTROL,** предложений, связанных с обработкой неопределенности. К числу операторов выбора логики обработки неопределенности относятся:

1) **CONTROL NOUNC** – нет неопределенности, используется по умолчанию;

2) **CONTROL CF** – использование факторов уверенности;

3) **CONTROL BAYES** – использование байесовской логики.

Установка порога (критерия окончания некоторых аспектов логического вывода) осуществляется с помощью следующего выражения:

CONTROL ‘**thereshold**’ <число>,

где <число> – число в диапазоне 0 ... 1.

Эти управляющие предложения могут появиться в любом месте в наборе правил. Если используются конфликтующие предложения, то отрабатывается последнее. Эти предложения действуют только в главном наборе правил – базе

правил и поднаборах правил, хранящихся внутри фреймов, что дает возможность смешивать методы, если это понадобится.

10.4.2 Факторы уверенности

Для обработки неопределенности в системе LEONARDO используются так называемые **многозначные объекты**. Заметим, что это самостоятельный класс объектов, ничего общего не имеющий со списковыми объектами (типа List). Дело в том, что в объектах типа List элемент либо принадлежит, либо не принадлежит списковому значению и неопределенности здесь нет.

В свою очередь, многозначные объекты являются специальными случаями текстовых объектов, где каждая реализация имеет свое собственное значение и связанное с ним значение фактора уверенности. Заметим, что числовые и списковые объекты в системе LEONARDO не могут быть многозначными, а для того, чтобы использовать факторы уверенности, в набор правил необходимо включить предложение CONTROL CF.

Текстовый объект становится многозначным путем добавления разработчиком прикладной ЭС в его фрейм специального слота **MultiValue:** без указания значения слота.

В том случае, если во фрейм текстового объекта добавлен слот MultiValue: , то правило может иметь, например, следующий вид:

if план **is** ‘не выполнен’

then прогноз **is** банкротство {cf.6};

прогноз **is** внешнее управление {cf.1},

где {cf.x} – идентификатор слова ‘уверенность’.

Если предположить, что до исполнения правила объект прогноз не был означен, то после срабатывания правила он будет иметь два воплощения – одно со значением «банкротство», другое – «внешнее управление».

Заметим, что если фрейм объекта не содержит слот MultiValue:, то присваивание нового значения объекту будет вновь устанавливать его фактор уверенности и его новое значение. При этом если в наборе правил используется управляющая опция CONTROL STRICT, то система LEONARDO будет выдавать сообщение, информирующее о присваивании объекту нового значения.

Таким образом, синтаксис правил с факторами уверенности такой же, как и у обычных правил, за исключением того, что разработчик прикладной ЭС может приписать в каждом присваивание консеквента правила фразу:

{cf <число>},

где <число> – число в диапазоне 0...1, которое означает уверенность, связанную с данной частью консеквента правила, или может интерпретироваться как вероятность наступления события, указанного в консеквенте. При этом если уверенность составляет, скажем, 72% , то необходимо написать – {cf.72}.

Процедурная поддержка многозначных объектов осуществляется следующим образом. Если определенная в главном наборе правил цель является многозначной, то во время консультации система в своем заключении упомянет только одно значение цели с наибольшим фактором уверенности. Аналогично, при замещении объекта его значением, в соответствии с определением слота Conclusion: (синтаксис – [ИмяОбьекта]), будет использоваться только одно значение с наибольшим фактором уверенности.

Процедурные возможности системы LEONARDO обеспечивают полную гибкость при работе с многозначными объектами, появляющимися в слоте Conclusion:. При этом использование в слоте Conclusion: строки

{run procedure (обьект1, обьект2, ... , обьектN}

обеспечивает вызов процедуры procedure с соответствующими объектами, представленными как параметры.

Для доступа ко всем значениям и факторам уверенности многозначных объектов во фреймах-процедурах могут использоваться следующие функции:

MULTIPLISITY (объект) – возвращает кратность объекта;

MVCERTAINTY (объект, n) – возвращает фактор уверенности n-го значения объекта;

MEMBER (объект, n) – возвращает n-е значение объекта;

CFLIST (объект) – возвращает список всех факторов уверенности объекта;

CERTAINTY (объект) – возвращает фактор уверенности простого объекта;

MEMLIST (объект) – возвращает список всех реализаций многозначного объекта, причем список упорядочен по убыванию фактора уверенности.

Заметим, что многозначные текстовые объекты могут использоваться как альтернатива списковым объектам, которые могут иметь только одно значение фактора уверенности для всего объекта (0 или 1). В этом случае текстовой объект становится многозначным с помощью задания слота MultiValue:. Процедурная поддержка таких объектов осуществляется с использованием перечисленных выше функций.

Для организации ввода пользователем значения фактора уверенности объекта наряду со слотом **Multi Value:** используется дополнительно вводимый разработчиком прикладной ЭС слот **Ask Certainty:** . Если этот слот отсутствует, то значение фактора уверенности, по умолчанию, становится равным 1.0 – полная уверенность.

Опции, задаваемые в слоте **Ask Certainty:**, включают.

BAR – высвечивается горизонтальное окно со сторонами LOW и HIGH. Курсор внутри окна перемещается с помощью стрелок, а также с использованием ключей HOME и END;

SCALE – высвечивается горизонтальное окно со шкалой 0 ... 1.0;

NUMERIC – фактор уверенности должен быть введен пользователем в числовом виде в диапазоне 0 ... 1.0 включительно.

Результирующий фактор уверенности консеквента получается путем умножения фактора уверенности антецедента на фактор уверенности консеквента. Таким образом, если имеется правило:

if прогноз **is** дождь **or** прогноз **is** снег

then действие **is** ОставатьсяДома {cf.5}

и текущая уверенность в том, что будет снег равна 0.2, а будет дождь – 0.6, то уверенность в истинности антецедента будет равна max (0.2 ; 0.6) = 0.6, а уверенность в истинности заключения – равна 0.5 * 0.6, т. е 0.3.

10.4.3 Байесовские наборы правил

Набору правил должно предшествовать предложение **CONTROL BAYES**. Синтаксис байесовских правил такой же, как и обычных правил, за исключением того, что разработчик прикладной ЭС может включить непосредственно перед ключевым словом **then** (в антецеденте правил) необязательную фразу:

{Ln <число1> Ls <число2>},

где Ls – логическая достаточность клаузы; Ln – логическая необходимость клаузы; <число1>, <число2> – число в соответствующем диапазоне.

Разработчик может также добавить к любому консеквенту фразу:

{Prior <число>},

где Prior – априорная вероятность исхода; <число> – число в диапазоне 0 ... 1.0.

Например:

if индикатор **is** красный

and станок **is** греется {**Ls** 10 **Ln** 0.01}

then действие **is** ‘выключить станок’ {**Prior** 0.25}

Рассмотрим семантику понятий **логическая необходимость** (Ln) и **логическая достаточность** (Ls).

Логическая необходимость антецедента (Ln) – это фактор, лежащий в диапазоне 0.001 ... 1.0. Он работает следующим образом:

если **антецедент ложный** (false), то Ln уменьшает шансы того, что консеквент окажется истинным. Значение 1.0 принимается по умолчанию и не оказывает никакого влияния. Значение 0.0 для Ln запрещено, так как в этом случае шансы стали бы равными 1/бесконечность. Пусть в приведенном примере антецедент ложный, тогда шансы того, что станок ‘необходимо выключить’, уменьшаются приблизительно в 100 раз.

Логическая достаточность антецедента (Ls) – это фактор, лежащий в диапазоне 1.0 ... 1000, который работает следующим образом: если **антецедент истинный** (true), то шансы того, что консеквент окажется истинным, возрастают. Значение Ls по умолчанию равно 1.0. Допустим, что в приведенном примере антецедент оказался истинным, тогда необходимость выключения станка увеличится приблизительно в 10 раз.

Таким образом, фактор логической необходимости (Ln) «срабатывает» тогда, когда антецедент **ложный**, а фактор логической достаточности (Ls) – когда антецедент **истинный.**

Априорная вероятность (Prior) означает априорные шансы консеквента оказаться истинным. В приведенном примере значение 0.25 означает, что шансы выключения станка могут находиться в диапазоне от 1 до 3. Это может быть реалистично, если переход к той части базы знаний, в которой записано это правило, выполняется только в том случае, когда станок находится в помещении, где вентиляция отсутствует, и станок находится на грани своего ресурса.

Система LEONARDO преобразует априорную вероятность консеквента в шансы консеквента по формуле

$$\text{ШАНСЫ}(k)=\frac{\text{ВЕРОЯТНОСТЬ}(k)}{1-\text{ВЕРОЯТНОСТЬ}(k)},$$

где ВЕРОЯТНОСТЬ – <число>, содержащееся во фразе {Prior<число>}; (к) – признак консеквента.

Далее вычисления выполняются по схеме

$$\text{ШАНСЫ}(k)=\begin{cases}\text{ШАНСЫ}(k)*L_S, \text{ если клауза истинна;}\\ \text{ШАНСЫ}(k)*L_N, \text{ если клауза ложная;}\\ \text{ШАНСЫ}(k), \text{ если истинность клаузы не определена.}\end{cases}$$

После этого вероятность консеквента вычисляется с помощью шансов по формуле

$$\text{ВЕРОЯТНОСТЬ}(k)=\frac{\text{ШАНСЫ}(k)}{1+\text{ШАНСЫ}(k)}.$$

В том случае, если значение антецедента само было выведено с помощью какого-либо правила, то его вероятность подсчитывается аналогичным образом.

Вычисленная вероятность используется для модификации факторов логической достаточности (Ls) или логической необходимости (Ln) по следующей схеме:

− новаяKs = 1 + (стараяLs - 1)*(ВероятностьАнтецедента - 0.5)/0.5, если вероятность антецедента >=0.5;

− новаяLn = 1 - (1 - стараяLn)*(0.5 - ВероятностьАнтецедента)/0.5, если вероятность антецедента < 0.5.

Ключевой характеристикой байесовского вывода и вывода с применением факторов уверенности является то, что они оба поддерживают разработку прикладной ЭС со множественными линиями правдоподобных рассуждений. Правила могут присваивать A, B, C и т. д. альтернативные значения с разными уверенностями. Некоторые более или менее разумные эвристические правила могут использоваться для прекращения рассмотрения некоторых альтернатив.

Примерами правил подобного типа могут быть следующие:

1) «если уверенность становится < 0.2, то предполагается, что значение истинности – ложь»;

2) «если уверенность когда-нибудь станет равной 1, то необходимо отсечь остальные альтернативы».

Решение о том, при каком значении уверенности прекращать рассматривать альтернативы, является менее важным, чем способность поддерживать параллельные цепочки рассуждений до тех пор, пока это необходимо и, таким образом, выводить ранжированное множество конечных заключений.

Система LEONARDO прекращает поиски дополнительных свидетельств истинности значения объекта, если определенность достигает значения 1.0 .

Для того чтобы управлять другими аспектами окончания вывода, в набор правил может быть записано квази-правило:

CONTROL ‘thereshold <число>’,

где <число> – числовое значение в диапазоне 0.0 ... 1.0, равное по умолчанию 0.2 .

В том случае, если были собраны все свидетельства того, что объект имеет некоторое значение и уверенность оказывается ниже порога, то уверенность становится равной 0.0 .

Важным является вопрос агрегирования неопределенности. В цепочке вывода может встретиться несколько правил, поставляющих свидетельства для одного и того же объекта. Предположим, что имеются следующие правила:

if B **includes** Y **then** C **is** V {cf.7}

if B **excludes** Z **then** C **is** V {cf.6},

где – B списковый объект, который, например, включает Y и не включает Z.

Вполне разумным представляется следующее рассуждение. Поскольку имеется более одного правила, приводящего к одному и тому же заключению, то свидетельства должны объединяться и увеличивать уверенность заключения. В этих целях в системе LEONARDO используется метод уменьшения интервала между текущей уверенностью консеквента и 1.0 за счет нового фактора уверенности.

Так, в рассматриваемом примере после срабатывания первого правила C станет равным V с уверенностью 0.7, а остающийся интервал будет равен 0.3 . После успешного срабатывания второго правила новое значение фактора уверенности будет равно 0.3 * 0.6 + 0.7, или 0.88 . Заметим, что результат не зависит от порядка правил (как и должно быть), и если фактор уверенности станет когда-нибудь равным 1.0, то результирующая уверенность также будет установлена равной 1.0.

10.5 Рекурсивные процедуры в системе leonardo

Рекурсия – это очень мощный инструмент решения многих задач, однако применять рекурсию следует только тогда, когда природа самой решаемой задачи рекурсивна. Как правило, это различные комбинаторные задачи, решаемые полным или ограниченным перебором. В общем случае рекурсия – это способ определения функции. Слово «**рекурсия**» – производное от термина **recurrence**, что в переводе с английского означает **повторение.**

Классическим примером здесь является факториал – функция f (n) = n! Эту функцию можно определить различными способами, в том числе и рекурсивно, при этом рекурсивность – это не свойство самой функции, а свойство ее описания:

$$n! = \begin{cases} 1 & \text{при } n = 0 \\ n * (n-1)! & \text{при } n = 0 \end{cases}$$

Как видно, здесь n! определяется через (n-1)!, т. е. через эту же самую функцию. Поэтому такое определение и называется рекурсивным. Другим примером могут служить так называемые числа Фибоначчи, которые рекурсивно задаются формулой

$$T(n) = T(n-1) + T(n-2).$$

В том случае, если вызов рекурсивной функции в явном виде содержится в теле описания этой функции, то говорят, что имеет место прямая или явная рекурсия. В свою очередь, если вызов содержится в теле другой процедуры,

к которой производится обращение из данной процедуры, то имеет место неявная или косвенная рекурсия.

В системе LEONARDO поддерживается как прямая, так и косвенная рекурсия.

Применение техники рекурсивного программирования в системе LEONARDO можно рассмотреть на примере решения известной задачи о Ханойской башне с использованием косвенной рекурсии.

В одной из древних легенд говорится следующее: «В храме Бенареса находится бронзовая плита с тремя алмазными стержнями. На одном из стержней Бог при сотворении мира нанизал 64 диска разного размера из чистого золота так, что наибольший диск лежит на бронзовой плите, а остальные образуют пирамиду, сужающуюся кверху. Это башня Брамы. Работая день и ночь, жрецы переносят диски с одного стержня на другой, следуя законам Брамы:

1) диски можно перемещать с одного стержня на другой только по одному;

2) нельзя класть больший диск на меньший.

Когда все 64 диска будут перенесены с одного стержня на другой, и храмы, и башни, и жрецы-брамины превратятся в прах и наступит конец света».

Эта легенда породила задачу о : переместить «m» дисков с одного из трех стержней на другой, соблюдая «законы Брамы».

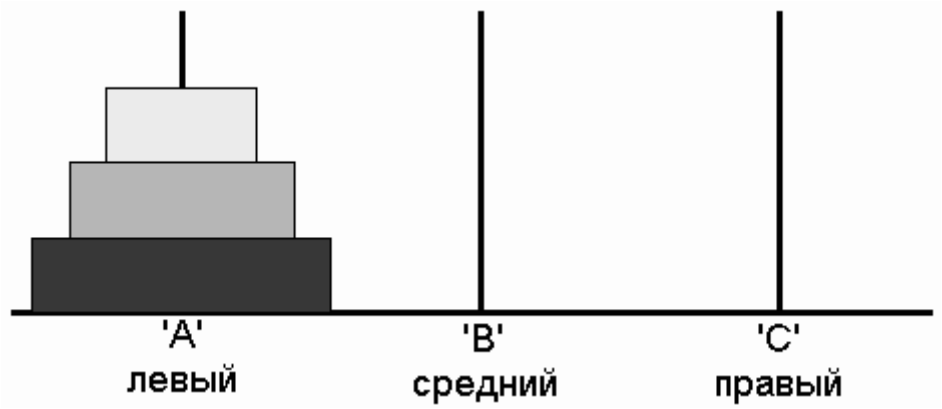


Рис. 10.3 Ханойская башня, состоящая из трех дисков

Число перемещений, которое необходимо сделать, определяется выражением 2^N - 1, где N – это высота башни.

Рекурсивная процедура, позволяющая решить задачу о Ханойской башне, строится следующим образом.

В главном наборе правил базы знаний определяется вызов процедуры, которая, в свою очередь, вызывает рекурсивную процедуру, позволяющую определить порядок перемещения дисков.

/* Главный набор правил для построения Ханойской башни

```
seek goal
if height > 0 then run dohanoi(height);
goal is done
```

Этот набор состоит из двух правил. В первом правиле с помощью безусловного правила seek указывается целевой объект goal. Второе правило обеспечивает ввод значения высоты башни (объект height) и проверку принадлежности введенного значения интервалу [1,11]. Если введенное число соответствует указанному интервалу, то происходит активизация процедуры dohanoi(height), в которой входным параметром является введенное пользователем значение объекта height. После завершения выполнения процедуры фиксируется факт достижения цели – «goal is done», машина логического вывода определяет, что цель достигнута, и экспертная система завершает работу.

После компиляции главного набора правил, система LEONARDO выполнит генерацию следующих объектов, в соответствии с контекстом их применения:

height	Real
dohanoi	Procedure
goal	Text
hanoi	Procedure

Объект hanoi появится в этом списке только после того, как во фрейме-процедуре dohanoi будет указано обращение к процедуре hanoi, а сама процедура dohanoi будет подвергнута компиляции.

Процедура dohanoi выглядит следующим образом:

```
1:  Name: dohanoi
...
4:  AcceptsReal: height
...
10: LocalReal: attribute,k
11: LocalText: from,to,spare
...
14:     Body:
15:
16:
17:     /*Очистка экрана и выдача заголовка
18:
19:     attribute=30
20:     screen(attribute)
21:     print(' ')
22:     print(' Необходимая последовательность перемещений')
```

```
23:       print('     для башни из ',height as 'xx',' дисков:')
24:       print(' ')
25:
26: /* Установка значений параметров рекурсивной процедуры
27:
28:                         from = 'A'
29:                         to='B'
30:                         spare='C'
31:                         k=0
32: /* Активизация рекурсивной процедуры
```

СПИСОК ЛИТЕРАТУРЫ

1. *Адаменко А.Д., Кучуков А.О.* Логическое программирование и Visual Prolog. – СПб.: БХВ-Петербург, 2003. – 992 с.
2. *Борисов В.В., Круглов В.В., Федулов А.С.* Нечеткие модели и сети. – М.: Горячая линия-Телеком, 2007. – 284 с.
3. Военная системотехника и системный анализ. Модели и методы принятия решений в сложных организационно-технических комплексах в условиях неопределенности и многокритериальности: учебник / под ред. проф. Б.В. Соколова. – СПб: ВИКУ имени А. Ф. Можайского, 1999. – 496 с.
4. *Войцеховский С.В. Хомоненко А.Д.* Выявление вредоносных программных воздействий на основе нечеткого вывода // Проблемы информационной безопасности. Компьютерные системы. – 2011. – № 3 – С. 81-91.
5. *Хомоненко А.Д., Войцеховский С.В.* Уточнение нечеткого вывода на основе алгоритма Мамдани в системе обнаружения вторжений // Проблемы информационной безопасности. Компьютерные системы. – 2011. – № 4. – С. 41-48.
6. *Гаврилова Т.А., Хорошевский В.А.* Базы знаний интеллектуальных систем. – СПб.: Питер, 2000. – 384 с.
7. *Герман О. В.* Введение в теорию экспертных систем и обработку знаний. – Мн.: ДизайнПРО, 1995. – 225 с.
8. *Джарратано Д., Райли Г.* Экспертные системы: принципы разработки и программирование: пер. с англ. – 4-е изд.– М.: ООО «И.Д. Вильямс», 2007. – 1152 с.
9. *Джексон П.* Введение в экспертные системы. – М.: Издательский дом «Вильямс», 2001. – 393 с.
10. *Ерофеев А. А., Поляков А. О.* Интеллектуальные системы управления. – СПб.: Изд-во СПбГТУ, 1994. – 264 с.
11. *Змитрович А. И.* Интеллектуальные информационные системы. – Мн.: НТООО «ТетраСистемс», 1997. – 368 с.
12. Искусственный интеллект: В 3 кн. Кн. 1. Системы общения и экспертные системы: справочник / под ред. Э. В. Попова. – М.: Радио и связь, 1990. – 406 с.
13. *Круглов В.В., Борисов В.В.* Искусственные нейронные сети. Теория и практика. – 2-ое изд., стереотип. – М.: Горячая линия-Телеком, 2002. – 382 с.
14. *Круглов В.В., Дли М.И., Голунов Р.Ю*. Нечеткая логика и искусственные нейронные сети: учебное пособие. – М.: Издательство Физико-математической литературы, 2001. – 224 с.
15. *Леоненков А.В.* Нечеткое моделирование в среде MATLAB и fuzzyTECH. – СПб: БХВ-Петербург, 2003. – 736 с.
16. *Марселлус Д.* Программирование экспертных систем на ТУРБО ПРОЛОГЕ: пер. с англ. – М.: Финансы и статистика, 1994. – 256 с.
17. *Микони С.В.* Модели и базы знаний: учебное пособие. – СПб.: ПГУПС, 2000.

18. *Нильсон Н.* Принципы искусственного интеллекта: перев.с англ. – М.: Радио и связь, 1985. – 374 с.

19. Основы современных компьютерных технологий: учебник / Г. А. Брякалов и др.; под ред. проф. А. Д. Хомоненко . – СПб.: КОРОНА принт, 2005. – 672 с.

20. *Рассел С, Норвиг П.* Искусственный интеллект: современный подход / пер. с англ.. – 2-е изд. – М.: Изд. дом «Вильямс», 2006. – 1408 с.

21. *Рыжиков Ю. И.* Руководство к практическим занятиям по информатике: учебное пособие. – СПб: ВИКУ имени А. Ф. Можайского, 1998. – 143 с.

22. *Слэйгл Дж.* Искусственный интеллект. Подход на основе эвристического программирования: пер. с англ. – М.: Мир, 1973. – 320 с.

23. Статические и динамические экспертные системы: учеб.пособие / Э. В. Попов, И.В Фоминых, Е.Б. Кисель и др. – М.: Финансы и статистика, 1996. – 320 с.

24. *Чень Ч., Ли Р.* Математическая логика и автоматическое доказательство теорем: пер. с англ. – М.: Наука. Главная редакция физико-математической литературы, 1983. – 360 с.

25. *Штовба С.Д.* Введение в теорию нечетких множеств и нечеткую логику. – Винница: Изд. Винницкого ГТУ, 2001. – 198 с. – [Электронный ресурс] Режим доступа: matlab.exponenta.ru, свободный.

26. *Штовба С.Д.* Проектирование нечетких систем средствами MATLAB. – М.: Горячая линия – Телеком, 2007. – 288 с.

27. *Яхъяева Г. Э.* Нечеткие множества и нейронные сети: учеб.пособие. – 2-е изд., испр. – М.: Интернет-университет информационных технологий: БИНОМ. Лаборатория знаний, 2008. – 315 с.

28. *Larsen H.L., Yager R.R.* The use of fuzzy relational thesauri for classificatory problem solving in information retrieval and expert systems // IEEE J. on System, Man, and Cybernetics 23(1), 1993. – P. 31-41.

29. *Mamdani E.H., Assilian S.* An experiment in linguistic synthesis with a fuzzy logic controller // International Journal of Man-Machine Studies. – 1975. – № 7. – P. 1-13.

30. *Mamdani E.H.* Application of fuzzy logic to approximate reasoning using linguistic Systems // Fuzzy Sets and Systems. – 1977. – V. 26. – P. 1182-1191.

31. *Sugeno M.* Industrial applications of fuzzy control . Elsevier Science Pub. Co., 1985.

32. *Takagi T., Sugeno M.* Fuzzy Identification of Systems and Its Applications to Modeling and Control // IEEE Trans. Systems, Man, and Cybernetics, 1985. – vol. 15. – № 1. – P. 116-132.

33. *Zhou Shang-Ming, Gan John Q.* Extracting Takagi-Sugeno Fuzzy Rules with Interpretable Submodels via Regularization of Linguistic Modifiers // IEEE Trans on Knowledge and Data Engineering. – August 2009. – Vol. 21. – № 8. – P. 1191-1204.