А. П. Частиков Т. А. Гаврилова Д.
Л.Белов

РАЗРАБОТКА ЭКСПЕРТНЫХ СИСТЕМ.

СРЕДА CLIPS

Санкт-Петербург
«БХВ-Петербург»
2003

УДК 681.3.06
ББК 32.813
Ч-25

Книга является одним из первых российских изданий по практической разработке экспертных систем. Подробно рассмотрены вопросы домашинного этапа разработки – извлечения и структурирования знаний, а также технологические аспекты разработки систем, основанных на знаниях. В качестве среды разработки экспертных систем описана среда CLIPS. Книга содержит достаточное количество справочной информации по CLIPS, что позволяет рекомендовать ее даже опытным программистам, которые в своей практической деятельности занимаются разработкой экспертных систем.

Для студентов вузов, инженеров по знаниям, программистов, проектировщиков эксепртных систем.

Содержание

Введение

Как-то в середине 90-х годов прошедшего столетия состоялась встреча Роберта Меткалфа, изобретателя Ethernet, и знаменитого профессора по искусственному интеллекту Эдварда Фейгенбаума. В состоявшейся дискуссии двух ученых были затронуты вопросы, связанные с состоянием дел в области искусственного интеллекта. Скептически настроенный Меткалф говорил: "Несмотря на все средства, израсходованные с 1969 года (года повального увлечения проблемами искусственного интеллекта) на работы по искусственному интеллекту, компьютер до сих пор не может разобрать примитивной устной речи и прочесть даже крупные буквы моей рукописи". Профессор Фейгенбаум признал, что "в течение долгого времени от искусственного интеллекта ожидалась большая отдача, чем он мог дать, и компании потеряли массу денег, безуспешно пытаясь использовать его в бизнесе". Однако, продолжал он, "в последнее время, как он убежден, искусственный интеллект прекрасно окупает вложенные средства, в основном в виде так называемых экспертных систем и баз знаний, способных принимать и аргументировать логические решения".

Да, в "классическом" или "традиционном" искусственном интеллекте, как сейчас называют его символьное направление, успешно создаются и развиваются экспертные системы или системы, основанные на знаниях, для широкого круга предметных областей.

Эдварда Фейгенбаума называют "отцом экспертных систем", как это значится на обложке одной из его книг "Становление экспертной компании". Он действительно стоял у истоков экспертной индустрии и создал первую экспертную систему DENDRAL в области идентификации органических соединений с помощью анализа масс-спектрограмм. Далее Фейгенбаум вместе с Шортлифом и Букхененом спроектировали первую медицинскую экспертную систему MYCIN, при этом они сделали открытие, которому было суждено существенно расширить сферу создания и использования экспертных систем. Когда они удалили из системы MYCIN базу знаний (конкретную медицинскую информацию), то осталась часть, называемая "машиной логического вывода". Было показано, что базу знаний можно изменять и заменять полностью, не нарушая целостности системы. Так возникла EMYCIN (Empty MYCIN — пустой MYCIN) или первая экспертная оболочка — инструментальная среда для построения экспертных систем различного назначения. С тех пор (с середины 70-х годов XX столетия) появилось большое число подобных инструментальных систем MicroExpert, GURU, G2, JESS и др.

В предлагаемой читателю книге освещаются вопросы теории и практики разработки экспертных систем. В качестве инструментальной среды разработки используется экспертная оболочка CLIPS. Суть технологии CLIPS заключается в том, что язык и среда CLIPS предоставляют пользователям возможность быстро создавать эффективные, компактные и легко управляемые экспертные системы. При этом пользователь применяет множество уже готовых инструментов (встроенный механизм управления базой знаний, механизм логического вывода, менеджеры различных объектов CLIPS и т. д.) и конструкций (упорядоченные факты, шаблоны, правила, функции, родовые функции, классы, модули, ограничения, встроенный язык COOL и т. д.).

Книга состоит из шести частей (часть VI — приложения, среди которых глоссарий) и списка литературы.

В части I описываются основные проблемы, связанные с разработкой экспертных систем: представление знаний, извлечение знаний, структурирование и концептуализация знаний, вывод на знаниях. Фактически это введение в инженерию знаний.

Часть II посвящена истории создания и развития инструментальной среды CLIPS, а также рассмотрению основных возможностей CLIPS.

Часть III содержит описание синтаксиса базовых конструкций языка CLIPS, необходимых для разработки экспертных систем — фактов, правил, глобальных переменных и функций. В конце данной части приводится пример разработки экспертной системы AutoExpert.

Часть IV раскрывает дополнительные возможности CLIPS, такие как использование классов и объектов, родовых функций, модулей и ограничений. Эти возможности значительно упрощают процесс создания экспертных систем. В последней главе данной части приведен пример построения экспертной системы C1OS для оптимизации бинарных таблиц соответствий логических схем, использующий описанные возможности.

Часть V содержит справочную информацию об основных функциях и командах CLIPS, необходимых разработчику, а также примеры их использования.

Часть VI объединяет приложения. В приложении I сосредоточены БНФ-определения всех основных конструкций языка CLIPS. Приложение 2 содержит список основных сообщений об ошибках среды CLIPS. В приложении 3 приводится перечень основных предупреждений среды CLIPS. Приложение 4 содержит все зарезервированные имена среды CLIPS. Приложение 5 представляет собой глоссарий используемых терминов.

В настоящий момент CLIPS является свободно распространяемым программным продуктом, который продолжает успешно развиваться и совершенствоваться. Совсем недавно, 29 марта 2002 г., появилась очередная версия CLIPS — 6.20. Несмотря на достаточно большое число зарубежных публикаций о языке и среде CLIPS, в русскоязычной литературе эта система до сих пор не освещалась. Данная книга призвана восполнить этот пробел.

Книга может использоваться в качестве учебного пособия студентами вузов при изучении дисциплин: "Интеллектуальные системы", "Системы искусственного интеллекта", "Интеллектуальные информационные системы", "Проектирование экспертных систем" и др. Книга содержит большой объем справочной информации по CLIPS, поэтому она может быть рекомендована и опытным пользователям, и программистам, которые в своей практической деятельности занимаются разработкой экспертных систем.

Авторы выражают свою признательность и благодарность Г. Г. Ворошиловой за помощь в работе при подготовке глав 15 и 16 книги к изданию и Дехкановой Марии за помощь в оформлении рисунков к части I книги.

Авторы благодарят рецензентов за ценные замечания, которые способствовали улучшению книги.

Особенную признательность авторы выражают к. т. н. А. И. Адаменко за доброжелательную помощь и поддержку при подготовке рукописи к изданию.

ЧАСТЬ I.Экспертные системы.

ГЛАВА 1. Системы, основанные на знаниях.

1.1. Знания и данные

Если у вас есть проблема или задача, которую нельзя решить самостоятельно — вы обращаетесь к знающим людям, или к экспертам, к тем, кто обладает ЗНАНИЯМИ. Термин "системы, основанные на знаниях" (knowledge-based systems) появился в 1976 году одновременно с первыми системами, аккумулирующими опыт и знания экспертов. Это были экспертные системы (expert systems) MYCIN и DENDRAL [Shortliffe, 1976; Shortliffe Feigenbaum, Buchanan, 1978] для медицины и химии. Они ставили диагноз при инфекционных заболеваниях крови и расшифровывали данные масс-спектрографического анализа.

Экспертные системы появились в рамках исследований по искусственному интеллекту (ИИ) (artificial intelligence) в тот период, когда эта наука переживала серьезный кризис, и требовался существенный прорыв в развитии практических приложений. Этот прорыв произошел, когда на смену поискам универсального алгоритма мышления и решения задач исследователям пришла идея моделировать конкретные знания специалистов-экспертов. Так в США появились первые коммерческие *системы, основанные на знаниях, или экспертные системы (ЭС).* Эти системы по праву стали первыми интеллектуальными системами, и до сих пор единственным критерием интеллектуальности является наличие механизмов работы со знаниями.

Так появился новый подход к решению задач искусственного интеллекта — *представление знаний.*

Подробнее об истории искусственного интеллекта можно почитать в [Поспелов, 1986; Джексон, 2001; Гаврилова, Хорошевский, 2001; Эндрю, 1985|.

При изучении интеллектуальных систем традиционно возникает вопрос — что же такое знания и чем они отличаются от обычных данных, десятилетиями обрабатываемых на компьютерах. Можно предложить несколько рабочих определений, в рамках которых это становится очевидным.

Определение 1.1

***Данные*— это информация, полученная в результате наблюдений или измерений отдельных свойств (атрибутов), характеризующих объекты, процессы и явления предметной области.**

Иначе, данные — это конкретные факты, такие как температура воздуха, высота здания, фамилия сотрудника, адрес сайта и пр.

При обработке на ЭВМ данные трансформируются, условно проходя следующие этапы:

- D1 — данные как результат измерений и наблюдений;
- D2 — данные на материальных носителях информации (таблицы, протоколы, справочники);
- D3 — модели (структуры) данных в виде диаграмм, графиков, функций;
- D4 — данные в компьютере на языке описания данных;
- D5 — базы данных на машинных носителях информации.

Знания же основаны на данных, полученных эмпирическим путем. Они представляют собой результат опыта и мыслительной деятельности человека, направленной на обобщение этого опыта, полученного в результате практической деятельности.

Так, если вооружить человека данными о том, что у него высокая температура (результат наблюдения или измерения), то этот факт не позволит ему решить задачу выздоровления. А если опытный врач поделится знаниями о том, что температуру можно снизить жаропонижающими препаратами и обильным питьем, то это существенно приблизит решение задачи выздоровления, хотя на самом деле нужны дополнительные данные и более глубокие знания.

Определение 1.2

***Знаний*— это связи и закономерности предметной области (принципы, модели, законы), полученные в результате практической деятельности и профессионального опыта, позволяющего специалистам ставить и решать задачи в данной области.**

При обработке на ЭВМ знания трансформируются аналогично данным:

- Z1 — знания в памяти человека как результат анализа опыта и мышления;
- Z2 — материальные носители знаний (специальная литература, учебники, методические пособия);

- Z3 — поле знаний — условное описание основных объектов предметной области, их атрибутов и закономерностей, их связывающих;
- Z4 — знания, описанные на языках представления знаний (продукционные языки, семантические сети, фреймы — *см. далее);*
- Z5 — база знаний на машинных носителях информации. Часто используется и такое определение знаний:

Знания — это хорошо структурированные данные, или данные о данных, или метаданные.

Ключевым этапом при работе со знаниями является формирование поля знаний (третий этап Z3), эта нетривиальная задача включает выявление и определение объектов и понятий предметной области, их свойств и связей между ними, а также представление их в наглядной и интуитивно понятной форме. Этот термин впервые был введен при практической разработке экспертной системы по психодиагностике АВТАНТЕСТ [Гаврилова, 1984] и теперь широко используется разработчиками ЭС.

Без тщательной проработки поля знаний не может быть речи о создании базы знаний.

Существенным для понимания природы знаний являются способы определения понятий. Один из широко применяемых способов основан на идее интенсионала и экстенсионала.

Определение 1.3

***Интенсионал понятия* — это определение его через соотнесение с понятием более высокого уровня абстракции с указанием специфических свойств.**

Например, интенсионал понятия "МЕБЕЛЬ": "предметы, предназначенные для обеспечения комфортного проживания человека и загромождающие дом".

Определение 1.4

Экстенсионал — это определение понятия через перечисление его конкретных примеров, т. е. понятий более низкого уровня абстракции.

Экстенсионал понятия "МЕБЕЛЬ": "Шкаф, диван, стол, стул и т. д.".

Интенсионалы формируют знания об объектах, в то время как экстенсионал объединяет данные. Вместе они формируют элементы поля знаний конкретной предметной области.

Для хранения данных используются базы данных (для них характерны большой объем и относительно небольшая удельная стоимость информации), для хранения знаний — базы знаний (небольшого объема, но исключительно дорогие информационные массивы).

База знаний — основа любой интеллектуальной системы, где знания описаны на некотором языке представления знаний, приближенном к естественному.

Знания можно разделить на:

- глубинные;
- поверхностные.

Поверхностные — знания о видимых взаимосвязях между отдельными событиями и фактами в предметной области.

Глубинные — абстракции, аналогии, схемы, отображающие структуру и природу процессов, протекающих в предметной области. Эти знания объясняют явления и могут использоваться для прогнозирования поведения объектов.

Поверхностные знания

"Если ввести правильный пароль, на экране компьютера появится изображение рабочего стола".

Глубинные знания

"Понимание принципов работы операционной системы и *знания на уровне квалифицированного системного администратора".*

Современные экспертные системы работают, в основном, с поверхностными знаниями. Это связано с тем, что на данный момент нет универсальных методик, позволяющих выявлять глубинные структуры знаний и работать с ними.

Кроме того, в учебниках по ИИ знания традиционно делят на *процедурные* и *декларативные.* Исторически первичными были процедурные знания, т. е. знания, "растворенные" в алгоритмах. Они управляли данными. Для их изменения требовалось изменять текст программ. Однако с развитием информатики и

программного обеспечения все большая часть знаний сосредотачивалась в структурах данных (таблицы, списки, абстрактные типы данных), т. е. увеличивалась роль декларативных знаний.

Рис. 1.1. Пирамида Ньюэлла

Сегодня знания приобрели чисто декларативную форму, т. е. знаниями считаются предложения, записанные на языках представления знаний, приближенных к естественному языку и понятных неспециалистам.

Один из пионеров ИИ Алан Ньюэлл проиллюстрировал эволюцию средств общения человека с компьютером как переход от машинных кодов через символьные языки программирования к языкам представления знаний (рис. 1.1).

1.2. Модели представления знаний

В настоящее время разработаны десятки моделей (или языков) представления знаний для различных предметных областей. Большинство из них может быть сведено к следующим классам:

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

В свою очередь это множество классов можно разбить на две большие группы (рис. 1.2):

- модульные;
- сетевые.

Модульные языки оперируют отдельными (не связанными) элементами знаний, будь то правила или аксиомы предметной области.

Сетевые языки предоставляют возможность связывать эти элементы или фрагменты знаний через отношения в семантические сети или сети фреймов.

Рассмотрим подробнее наиболее популярные у разработчиков языки представления знаний (ЯПЗ).

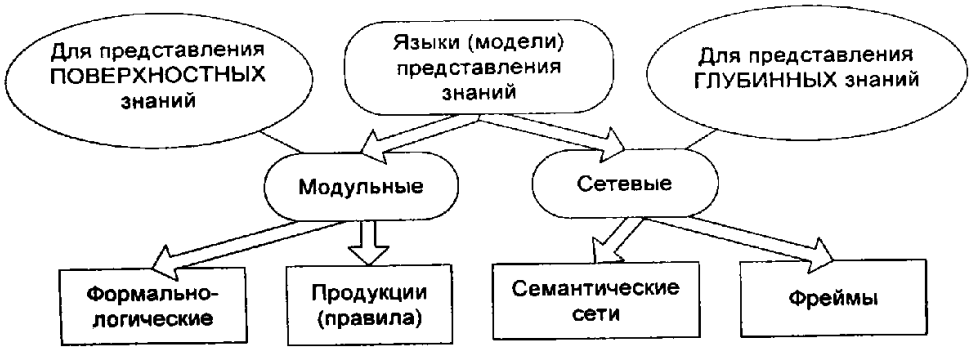

Рис. 1.2. Классификация моделей представления знаний

1.2.1. Продукционная модель

ЯПЗ, основанные на правилах (rule-based), являются наиболее распространенными и более 80% ЭС используют именно их.

Определение 1.5

Продукционная модель или модель, основанная на правилах, позволяет представить знания в виде предложений типа "Если (условие), то (действие)".

Под "условием" (антецедентом) понимается некоторое предложение-образец, по которому осуществляется поиск в базе знаний, а под "действием" (консеквентом) — действия, выполняемые при успешном исходе поиска (они могут быть промежуточными, выступающими далее как условия, и терминальными или целевыми, завершающими работу системы).

Чаще всего вывод на такой базе знаний бывает *прямой* (от данных к поиску цели) или *обратный* (от цели для ее подтверждения — к данным). Данные — это исходные факты, хранящиеся в базе фактов, на основании которых запускается машина вывода или интерпретатор правил, перебирающий правила из продукционной базы знаний *(см. разд. 1.3).*

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

Имеется большое число программных средств, реализующих продукционный подход (например, языки высокого уровня CLIPS и OPS 5; "оболочки" или "пустые" ЭС — EXSYS Professional и Карра, инструментштные системы КЕЕ, ARTS, PIES [Хорошевский, 1993]), а также промышленных ЭС на его основе (например, ЭС, созданных средствами G2 [Попов, 1996]). Подробнее см. [Попов, Фоминых и др., 1996; Хорошевский, 1993; Гаврилова, Хорошевский, 2001; Durkin, 1998].

1.2.2.Семантические сети

Термин *"семантическая" означает* "смысловая", а сама семантика — это наука, устанавливающая отношения между символами и объектами, которые они обозначают, т. е. наука, определяющая смысл знаков. Модель на основе семантических сетей была предложена американским психологом Куиллиа-ном. Основным ее преимуществом является то, что она более других соответствует современным представлениям об организации долговременной памяти человека [Скрэгг, 1983].

Определение 1. 6

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

В качестве понятий обычно выступают абстрактные или конкретные объекты, а *отношения* это связи типа: "это" ("АКО — A-Kind-Of, "is" или "элемент класса"), "имеет частью" ("has part"), "принадлежит", "любит".

Можно предложить несколько классификаций семантических сетей, связанных с типами отношений между понятиями.

-По количеству типов отношений:

- •однородные (с единственным типом отношений);
- •неоднородные (с различными типами отношений).

-По типам отношений:

- •бинарные (в которых отношения связывают два объекта);
- N-арные (в которых есть специальные отношения, связывающие более двух понятий).

Наиболее часто в семантических сетях используются следующие отношения:

•элемент класса (роза *это* цветок);

•атрибутивные связи /иметь свойство (память *имеет свойство* — объем);

•*значение* свойства (цвет *имеет значение* — желтый);

•пример элемента класса (роза, *например* — чайная);

•связи типа *"часть-целое"* (велосипед *включает* руль);

•функциональные связи (определяемые обычно глаголами "производит", "влияет"...);

•количественные *(больше, меньше, равно...);*

•пространственные *(далеко от, близко от, за, под, над...);*

•временные *(раньше, позже, в течение...);*

•логические связи *(и, или, не)* и др.

Минимальный состав отношений в семантической сети таков:

•элемент класса или АКО;

•атрибутивные связи /иметь свойство;

•значение свойства.

Недостатком этой модели является сложность организации процедуры организации вывода на семантической сети.

Эта проблема сводится к нетривиальной задаче поиска фрагмента сети, соответствующего некоторой подсети, отражающей поставленный запрос к базе.

На рис. 1.3 изображен пример семантической сети. В качестве вершин тут выступают понятия "человек", "т. Смирнов", "Audi A4", "автомобиль", "вид транспорта" и "двигатель".

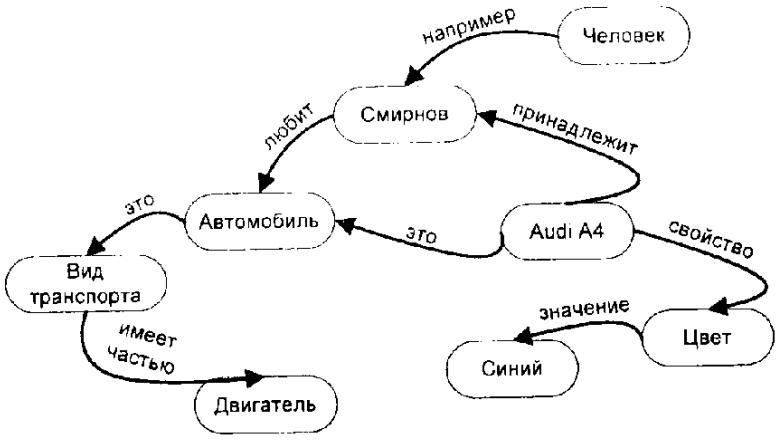

Рис. 1.3. Семантическая сеть

Для реализации семантических сетей существуют специальные сетевые языки, например, NET [Цейтин, 1985], язык реализации систем SIMER + MIR [Осипов, 1997] и др. Широко известны экспертные системы, использующие семантические сети в качестве языка представления знаний — PROSPECTOR, CASNET, TORUS [Хейес-Рот и др., 1987; Durkin, 1998].

1.2.3. Фреймы

Термин *фрейм* (от англ. *frame* — "каркас" или "рамка") был предложен Марвином Минским [Минский, 1979], одним из пионеров ИИ, в 70-е годы для обозначения структуры знаний для восприятия пространственных сцен. Эта модель, как и семантическая сеть, имеет глубокое психологическое обоснование.

Определение 1.7

Фрейм — это абстрактный образ для представления стереотипа объекта, понятия или ситуации.

Интуитивно понятно, что под абстрактным образом понимается некоторая обобщенная и упрощенная модель или структура. Например, произнесение вслух слова "комната" порождает у слушающих образ комнаты: "жилое помещение с четырьмя стенами, полом, потолком, окнами и дверью, площадью 6—20 м^2". Из этого описания ничего нельзя убрать (например, убрав окна, мы получим уже чулан, а не комнату), но в нем есть "дырки" или *"слоты"*— это незаполненные значения некоторых атрибутов — например, количество окон, цвет стен, высота потолка, покрытие пола и др.

В теории фреймов такой образ комнаты называется фреймом комнаты. Фреймом также называется и формализованная модель для отображения образа.

Различают *фреймы-образцы* или *прототипы,* хранящиеся в базе знаний, и *фреймы-экземпляры,* которые создаются для отображения реальных фактических ситуаций на основе поступающих данных. Модель фрейма является достаточно универсальной, поскольку позволяет отобразить все многообразие знаний о мире через:

•*фреймы-структуры,* использующиеся для обозначения объектов и понятий (заем, залог, вексель);

•*фреймы-роли* (менеджер, кассир, клиент);

•*фреймы-сценарии* (банкротство, собрание акционеров, празднование именин);

•*фреймы-ситуации* (тревога, авария, рабочий режим устройства) и др.

Традиционно структура фрейма может быть представлена как список свойств:

(ИМЯ ФРЕЙМА:
(имя 1-го слота: значение 1-го слота),
(имя 2-го слота: значение 2-го слота),
…….
(имя N-го слота: значение N-го слота)).

Ту же запись можно представить в виде таблицы (см. табл. 1.1), дополнив ее двумя столбцами.

Таблица 1.1. *Структура фрейма*

Имя фрейма			
Имя слота	Значение слота	Способ получения значения	Присоединенная процедура

В таблице дополнительные столбцы (3-й и 4-й) предназначены для описания способа получения слотом его значения и возможного присоединения к тому или иному слоту специальных процедур, что допускается в теории фреймов. В качестве значения слота может выступать имя другого фрейма, так образуются сети фреймов.

Существует несколько способов получения слотом значений во фрейме-экземпляре:

•по умолчанию от фрейма-образца (Default-значение);

•через наследование свойств от фрейма, указанного в слоте АКО;

•по формуле, указанной в слоте;

•через присоединенную процедуру;

•явно из диалога с пользователем;

•из базы данных.

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

Например, в сети фреймов на рис. 1.4 понятие "ученик" наследует свойства фреймов "ребенок" и "человек", которые находятся на более высоком уровне иерархии. На вопрос "любят ли ученики сладкое?" следует ответ "да", т. к. этим свойством обладают все дети, что указано во фрейме "ребенок". Наследование свойств может быть частичным: возраст для учеников не наследуется из фрейма "ребенок", поскольку указан явно в своем собственном фрейме.

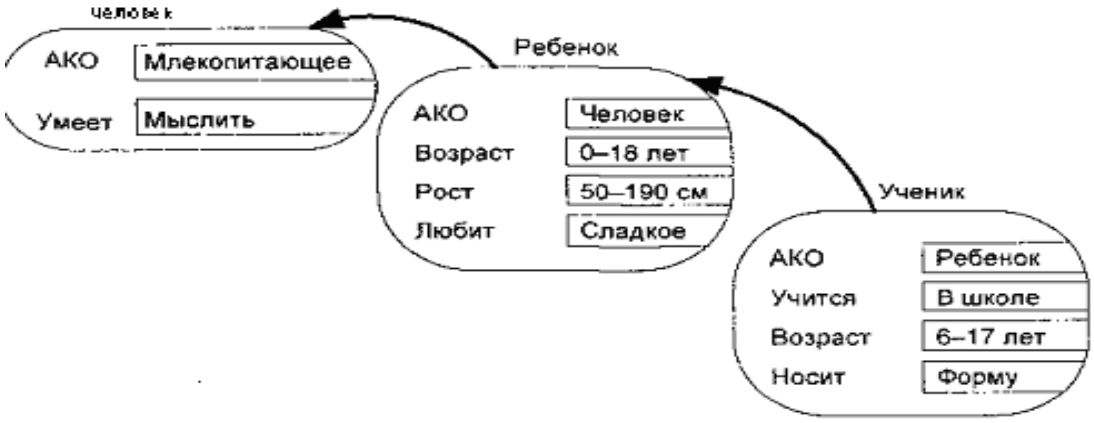

Рис. 1.4. Сеть фреймов

Основным преимуществом фреймов как модели представления знаний является то, что она отражает концептуальную основу организации памяти человека [Шенк, Хантер, 1987], а также ее гибкость и наглядность.

Специальные языки представления знаний в сетях фреймов FRL (Frame Representation Language) [Байдун, Бунин, 1990], KRL (Knowledge Representation Language) [Уотермен, 1989], фреймовая "оболочка" Карра [Стрельников, Борисов, 1997] и другие программные средства позволяют эффективно строить промышленные ЭС. Широко известны такие фрейм-ориентированные экспертные системы, как ANALYST, МОДИС, TRISTAN, ALTERID [Ковригин, Перфильев, 1988; Николов, 1988; Sisodia, Warkentin, 1992].

1.2.4. Формальные логические модели

Традиционно в представлении знаний выделяют *формальные логические модели, основанные на классическом* исчислении предикатов 1-го порядка, когда предметная область или задача описывается в виде набора аксиом. Реально исчисление предикатов 1-го порядка в промышленных экспертных системах практически не используется. Эта логическая модель применима в основном в исследовательских "игрушечных" системах, т. к. предъявляет очень высокие требования и ограничения к предметной области. В промышленных же экспертных системах используются различные ее модификации и расширения, изложение которых выходит за рамки этой книги. См. [Ада-менко, Кучуков, 2003].

1.3. Вывод на знаниях

Как уже сказано в *разд. 1.2,* наибольшее распространение получила продукционная модель представления знаний. При ее использовании база знаний состоит из набора правил, а программа, управляющая перебором правил, называется *машиной вывода.*

Определение 1.8

Машина вывода **(интерпретатор правил) — это программа, имитирующая логический вывод эксперта, пользующегося данной продукционной базой знаний для интерпретации поступивших в систему данных.**

Обычно она выполняет две функции:

- просмотр существующих данных (фактов) из рабочей памяти (базы данных) и правил из базы знаний и добавление (по мере возможности) в рабочую память новых фактов;
- определение порядка просмотра и применения правил. Этот механизм управляет процессом консультации, сохраняя для пользователя информацию о полученных заключениях, и запрашивает у него информацию, когда для срабатывания очередного правила в рабочей памяти оказывается недостаточно данных [Осуга, Саэки, 1990].

В подавляющем большинстве систем, основанных на знаниях, механизм вывода представляет собой небольшую по объему программу и включает двакомпонента — один реализует собственно вывод, другой управляет этим процессом.

Действие *компонента вывода* основано на применении правила, называемого *modus ponens:* "Если известно, что истинно утверждение А, и существует правило вида "ЕСЛИ А, ТО В", тогда утверждение В также истинно".

Таким образом, правила срабатывают, когда находятся факты, удовлетворяющие их левой части: если истинна посылка, то должно быть истинно и заключение.

Компонент вывода должен функционировать даже при недостатке информации. Полученное решение может и не быть точным, однако система не должна останавливаться из-за того, что отсутствует какая-либо часть входной информации.

Управляющий компонент определяет порядок применения правил и выполняет четыре функции:

1.*Сопоставление*— образец правила сопоставляется с имеющимися фактами.

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

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

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

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

В одном цикле может сработать только одно правило. Если несколько правил успешно сопоставлены с фактами, то интерпретатор производит выбор по определенному критерию единственного правила, которое срабатывает в данном цикле. Цикл работы интерпретатора схематически представлен на рис. 1.5.

Информация из рабочей памяти последовательно сопоставляется с посылками правил для выявления успешного сопоставления. Совокупность отобранных правил составляет так называемое *конфликтное множество.* Для разрешения конфликта интерпретатор имеет критерий, с помощью которого он выбирает единственное правило, после чего оно срабатывает. Это выражается в занесении фактов, образующих заключение правила, в рабочую

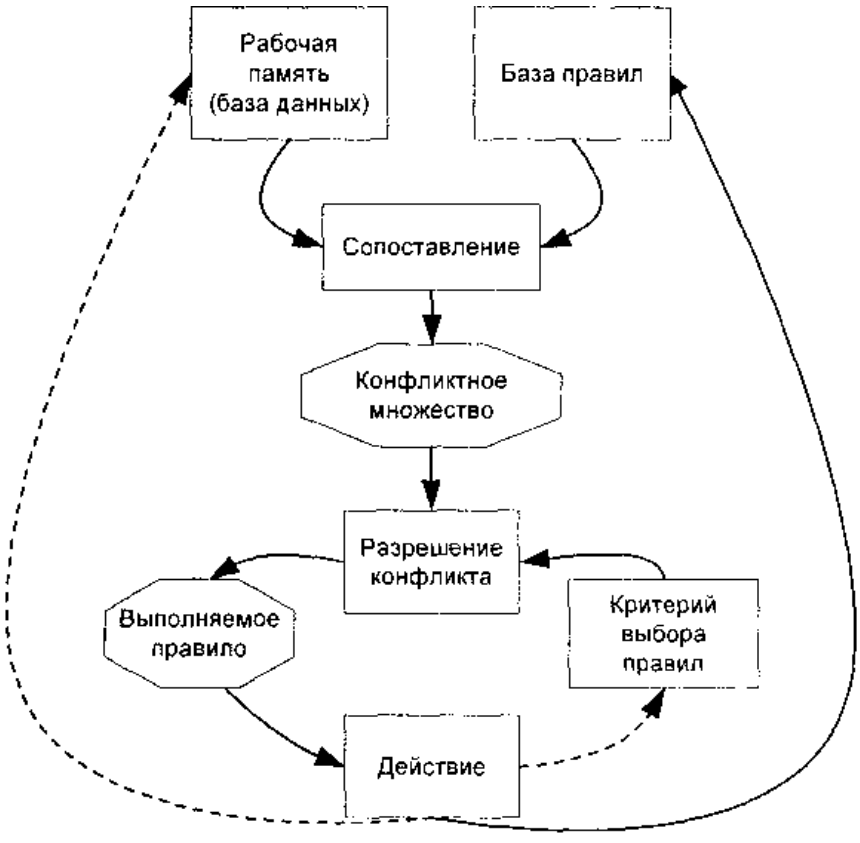

Рис. 1.5. Цикл работы интерпретатора

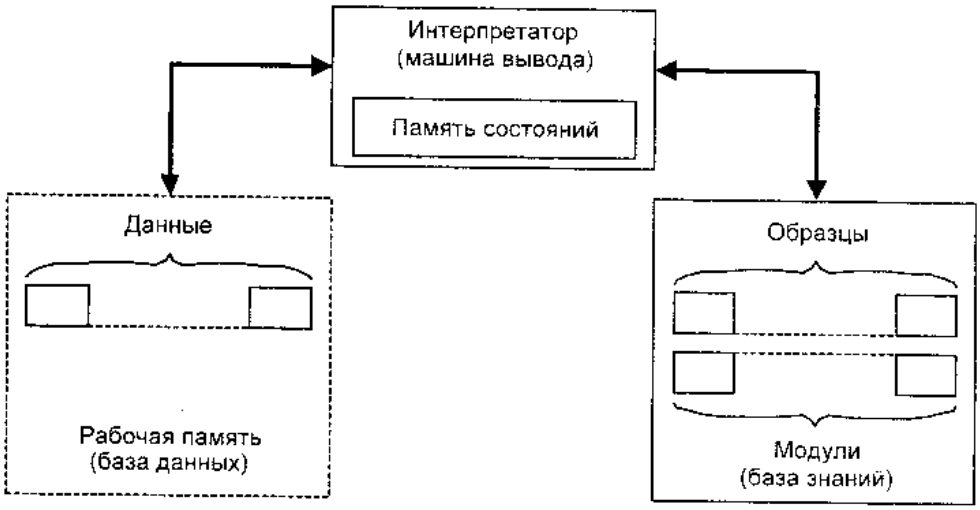

Рис. 1.6. Схема функционирования интерпретатора

память или в изменении критерия выбора конфликтующих правил. Если же в заключение правила входит название какого-нибудь действия, то оно выполняется.

Работа машины вывода зависит только от состояния рабочей памяти и от состава базы знаний. На практике обычно учитывается история работы, т. е. поведение механизма вывода в предшествующих циклах. Информация о поведении механизма вывода запоминается в памяти состояний (рис. 1.6). Обычно память состояний содержит протокол системы.

1.3.1. Управление выводом

От выбранного метода поиска, т. е. стратегии вывода, будет зависеть порядок применения и срабатывания правил. Процедура выбора сводится к определению направления поиска и способа его осуществления. Процедуры, реализующие поиск, обычно "зашиты" в механизм вывода, поэтому в большинстве систем инженеры знаний не имеют к ним доступа и, следовательно, не могут в них ничего изменять по своему желанию.

При разработке стратегии управления выводом важны:

- исходная точка в пространстве состояний. От выбора этой точки зависит и метод осуществления поиска — в прямом или в обратном направлении.
- метод и стратегия перебора — в глубину, в ширину, по подзадачам или иначе.

При *обратном* порядке вывода вначале выдвигается некоторая гипотеза, а затем механизм вывода как бы возвращается назад, переходя к фактам, пытаясь найти те, которые подтверждают гипотезу (рис. 1.7, правая часть). Если она оказалась правильной, то выбирается следующая гипотеза, детализирующая первую и являющаяся по отношению к ней подцелью. Далее отыскиваются факты, подтверждающие истинность подчиненной гипотезы. Вывод такого типа называется управляемым целями, или управляемым консеквентами. Обратный поиск применяется в тех случаях, когда цели известны и их сравнительно немного.

В системах с *прямым выводом* по известным фактам отыскивается заключение, которое из этих фактов следует (см. рис. 1.7, левая часть). Если такое заключение удается найти, то оно заносится в рабочую память. Прямой вывод часто называют выводом, управляемым данными, или выводом, управляемым антецедентами.

Существуют системы, в которых вывод основывается на сочетании упомянутых выше методов — обратного и ограниченного прямого. Такой комбинированный метод получил название циклического.

Пусть имеется фрагмент базы знаний из двух правил:

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

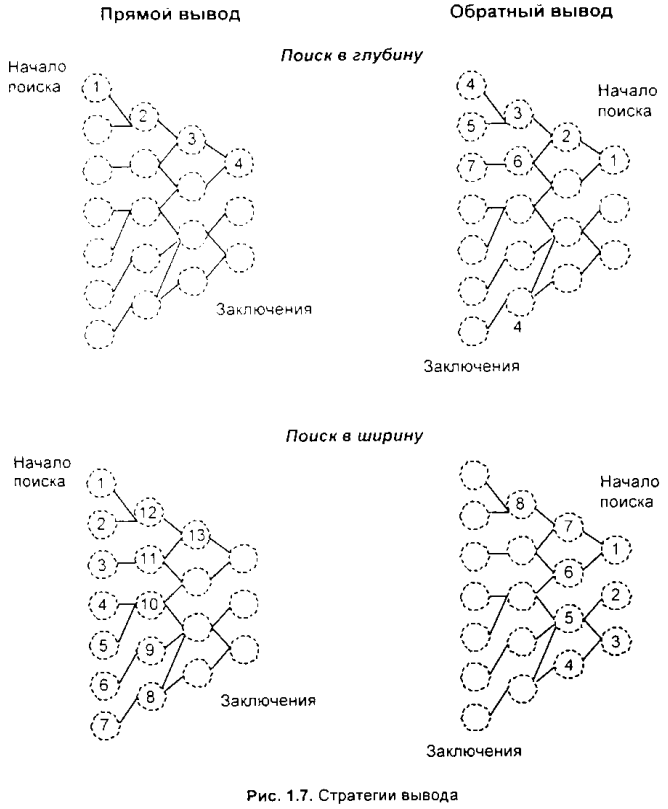

Рис. 1.7. Стратегии вывода

Рис. 1.7. Стратегии вывода

Предположим, в систему поступили факты — "человек активный" и "любит солнце".

ПРЯМОЙ ВЫВОД— исходя из фактических данных, получить рекомендацию.

- 1-й проход.

•*Шаг 1.* Пробуем /7/, не работает (не хватает данных "отдых — летом").

•*Шаг 2.* Пробуем /72, работает, в базу поступает факт "отдых — летом".

- 2-й проход.

- *Шаг 3.* Пробуем Я/, работает, активизируется цель "ехать в горы", которая и выступает как совет, который дает ЭС.

ОБРАТНЫЙ ВЫВОД— подтвердить выбранную цель при помощи имеющихся правил и данных.

- 1-й проход.

- *Шаг 1.* Цель — "ехать в горы": пробуем *П1* — данных "отдых — летом" нет, они становятся новой целью и ищется правило, где она в левой части.
- *Шаг 2.* Цель "отдых — летом": правило *П2* подтверждает цель и активизирует ее.

- 2-й проход.

- *Шаг 3.* Пробуем *П1,* подтверждается искомая цель.

1.3.2. Методы поиска в глубину и в ширину

В системах, база знаний которых насчитывает сотни правил, желательным является использование стратегии управления выводом, позволяющей минимизировать время поиска решения и тем самым повысить эффективность вывода. К числу таких стратегий относятся: поиск в глубину, поиск в ширину, разбиение на подзадачи и альфа-бета-алгоритм [Таунсенд, Фохт, 1991; Уэно, Исидзука, 1989; Справочник по ИИ, 1990].

При *поиске в глубину* в качестве очередной подцели выбирается та, которая соответствует следующему, более детальному уровню описания задачи. Например, диагностирующая система, сделав на основе известных симптомов предположение о наличии определенного заболевания, будет продолжать запрашивать уточняющие признаки и симптомы этой болезни до тех пор, пока полностью не опровергнет выдвинутую гипотезу.

При *поиске в ширину,* напротив, система вначале проанализирует все симптомы, находящиеся на одном уровне пространства состояний, даже если они относятся к разным заболеваниям, и лишь затем перейдет к симптомам следующего уровня детальности.

Разбиение на подзадачи подразумевает выделение подзадач, решение которых рассматривается как достижение промежуточных целей на пути к конечной цели. Примером, подтверждающим эффективность разбиения на подзадачи, является поиск неисправностей в компьютере — сначала выявляется отказавшая подсистема (питание, память и т. д.), что значительно сужает пространство поиска. Если удается правильно понять сущность задачи и оптимально разбить ее на систему иерархически связанных целей-подцелей, то можно добиться того, что путь к ее решению в пространстве поиска будет минимален.

Альфа-бета-алгоритм позволяет уменьшить пространство состояний путем удаления ветвей, не перспективных для успешного поиска. Поэтому просматриваются только те вершины, в которые можно попасть в результате следующего шага, после чего неперспективные направления исключаются. Альфа-бета-алгоритм нашел широкое применение в основном в системах, ориентированных на различные игры, например, в шахматных программах.

1.4. Работа с нечеткостью

При формализации знаний существует проблема, затрудняющая использование традиционного математического аппарата. Это проблема описания понятий, оперирующих качественными характеристиками объектов *(много, мало, сильный, очень сильный* и т. п.). Эти характеристики обычно размыты и не могут быть однозначно интерпретированы, однако содержат важную информацию (например, "одним из возможных признаков гриппа является *высокая* температура").

Кроме того, в задачах, решаемых интеллектуальными системами, часто приходится пользоваться неточными знаниями, которые не могут быть интерпретированы как полностью истинные или ложные (логические true/false или 0/1). Существуют знания, достоверность которых выражается некоторой промежуточной цифрой, например 0,7.

Как, не разрушая свойства размытости и неточности, представлять подобные знания формально? Для разрешения таких проблем в начале 70-х годов XX века американский математик Лотфи Заде предложил формальный аппарат *нечеткой* (fuzzy) алгебры и нечеткой логики [Заде, 1972]. Позднее это направление получило широкое распространение [Орловский, 1981; Аверкин и др., 1986; Яшин, 1990] и положило начало одной из ветвей ИИ под названием *мягкие вычисления* (soft computing).

Л. Заде ввел одно из главных понятий в нечеткой логике — понятие лингвистической переменной.

Лингвистическая переменная (ЛП) — это переменная, значение которой определяется набором вербальных (т. е. словесных) характеристик некоторого свойства.

Например, ЛП "рост" определяется через набор *{карликовый, низкий, средний, высокий, очень высокий}*.

1.4.1. Основы теории нечетких множеств

Значения лингвистической переменной (ЛП) определяются через так называемые *нечеткие множества* (НМ), которые в свою очередь определены на некотором *базовом* наборе значений или базовой числовой шкале, имеющей размерность. Каждое значение ЛП определяется как нечеткое множество (например, НМ "низкий рост").

Нечеткое множество определяется через некоторую базовую шкалу B и функцию принадлежности НМ — $\mu(x)$, $x \in B$ Д принимающую значения на интервале [0; 1]. Таким образом, нечеткое множество B — это совокупность пар вида $(x, \mu(x))$, где $x \in B$. Часто встречается и такая запись:

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

где x_i — i-е значение базовой шкалы.

Функция принадлежности определяет субъективную *степень уверенности* эксперта в том, что данное конкретное значение базовой шкалы соответствует определяемому НМ. Эту функцию не стоит путать с вероятностью, носящей объективный характер и подчиняющейся другим математическим зависимостям.

Например, для двух экспертов определение НМ "высокая" для ЛП "цена автомобиля" в условных единицах может существенно отличаться в зависимости от их социального и финансового положения.

"Высокая_цена_автомобиля_1" = {50000/1 + 25000/0.8 + 10000/0.6 + 5000/0.4}

"Высокая_цена_автомобиля_2" = {25000/1 + 10000/0.8 + 5000/0.7 + 3000/0.4}

Пусть перед нами стоит задача интерпретации значений ЛП "возраст", таких как "молодой" возраст, "преклонный" возраст или "переходный" возраст. Определим "возраст" как ЛП (рис. 1.8). Тогда "молодой", "преклонный", "переходный" будут значениями этой лингвистической переменной. Более полно, базовый набор значений ЛП "возраст" следующий:

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

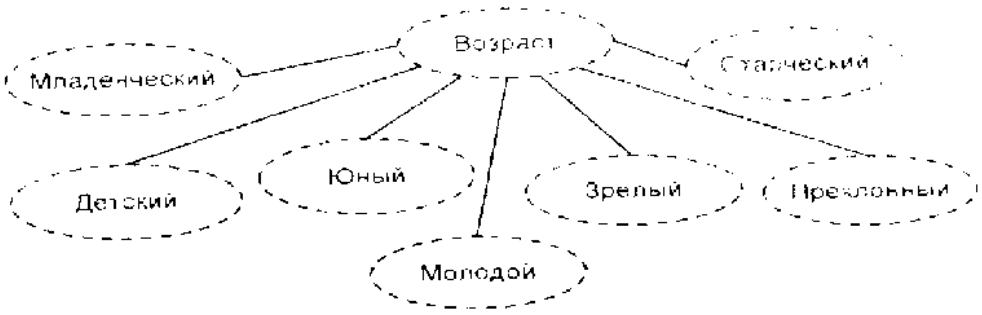

Рис. 1.8. Лингвистическая переменная "возраст" и нечеткие множества, определяющие ее значения

Для ЛП "возраст" базовая шкала — это числовая шкала от 0 до 120, обозначающая количество прожитых лет, а функция принадлежности определяет, насколько мы уверены в том, что данное количество лет можно отнести к данной категории возраста. На рис. 1.9 отражено, как одни и те же значения базовой шкалы могут участвовать в определении различных НМ.

Например, определить значение НМ "младенческий" можно так:

$$\text{"младенческий"} = \left\{\frac{0,5}{1} + \frac{1}{0,9} + \frac{2}{0,8} + \frac{3}{0,7} + \frac{4}{0,6} + \frac{5}{0,3} + \frac{10}{0,1}\right\}$$

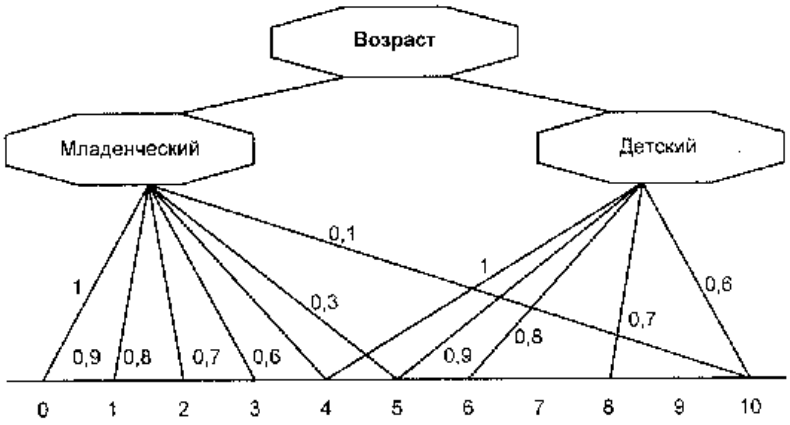

Рис. 1.9. Формирование нечетких множеств

Рис. 1.10. График функции принадлежности нечеткому множеству "младенческий возраст"

Рис. 1.10 иллюстрирует оценку НМ неким усредненным экспертом, который ребенка до полугода с высокой степенью уверенности относит к младенцам ($\mu = 1$). Дети до четырех лет причисляются к младенцам тоже, но с меньшей степенью уверенности ($0,5 < \mu < 0,9$), а в десять лет ребенка называют так только в очень редких случаях — к примеру, для девяностолетней бабушки и 15 лет может считаться младенчеством. Таким образом, нечеткие множества позволяют при определении понятия учитывать субъективные мнения отдельных индивидуумов.

1.4.2. Операции с нечеткими знаниями

Для операций с нечеткими знаниями, выраженными при помощи лингвистических переменных, существует много различных способов. Эти способы являются в основном эвристиками.

Мы не будем останавливаться на этом вопросе подробно, укажем лишь для примера определение нескольких операций. Например, операция "ИЛИ" часто задается так [Аверкин и др., 1986; Яшин, 1990]:

$$\mu(x) = \max(\mu 1(x), \mu 2(x)$$

(так называемая логика Заде) или так:

$$\mu(x) = \mu 1(x) + \mu 2(x - \mu 1(x) * \mu 2(x)$$

(вероятностный подход).

Усиление иди ослабление лингвистических понятий достигается введением специальных квантификаторов. Например, если понятие "старческий возраст" определяется как

$$\left\{ \frac{60}{0,6} + \frac{70}{0,8} + \frac{80}{0,9} + \frac{90}{1} \right\}$$

то понятие "очень старческий возраст" распознается как

$$con\ (A) = A^2 = \sum_i \frac{X_i}{\mu i 2}$$

т. е. очень старческий возраст определится так:

$$\left\{ \frac{60}{0,36} + \frac{70}{0,64} + \frac{80}{0,81} + \frac{90}{1} \right\}$$

Для вывода на нечетких множествах используются специальные отношения и операции над ними (подробнее см. [Орловский, 1981]).

Одним из первых применений теории НМ стало использование коэффициентов уверенности для вывода рекомендаций медицинской системы MYC1N [Shortliffe, 1976]. Этот метод использует несколько эвристических приемов. Он стал примером обработки нечетких знаний, повлиявших на последующие системы.

В настоящее время в большинство инструментальных средств разработки систем, основанных на знаниях, включены элементы работы с НМ, кроме того, разработаны специальные программные средства реализации так называемого нечеткого вывода, например "оболочка" FuzzyCLIPS.

1.5. Архитектура и особенности экспертных систем

Центральная парадигма интеллектуальных технологий сегодня — это обработка знаний. Системы, ядром которых является база знаний или модель предметной области, описанная на языке сверхвысокого уровня, приближенном к естественному, называют *интеллектуальными.*

Чаще всего интеллектуальные системы (ИС) применяются для решения сложных задач, где основная сложность решения связана с использованием слабоформализованных знаний специалистов-практиков и где логическая (или смысловая) обработка информации превалирует над вычислительной. Например, понимание естественного языка, поддержка принятия решения в сложных ситуациях, постановка диагноза и рекомендации по методам лечения, анализ визуальной информации, управление диспетчерскими пультами и др.

Фактически сейчас прикладные интеллектуальные системы используются в десятках тысяч приложений. А годовой доход от продаж программных и аппаратных средств искусственного интеллекта еще в 1989 г. в США составлял 870 млн. долларов, а в 1990 г. — 1,1 млрд. долларов [Попов, 1996]. В дальнейшем почти тридцати процентный прирост дохода сменился более плавным наращиванием темпов (по материалам [Поспелов, 1997; Хорошевский, 1997; Попов, 1996; Walker, Miller, 1987; Tuthill, 1994; Durkin, 1998]).

Наиболее распространенным видом ИС являются экспертные системы.

Определение 1.9

***Экспертные системы (ЭС)*— это наиболее распространенный класс ИС, ориентированный на тиражирование опыта высококвалифицированных специалистов в областях, где качество принятия решений традиционно зависит от уровня экспертизы,**

например таких, как медицина, юриспруденция, геология, экономика, военное дело и др.

ЭС эффективны лишь в специфических "экспертных" областях, где важен эмпирический опыт специалистов.

Только в США ежегодный доход от продаж инструментальных средств разработки ЭС составлял в начале 90-х годов 300—400 млн. долларов, а от применения ЭС — 80—90 млн. долларов [Попов, 1996]. Ежегодно крупные фирмы разрабатывают десятки ЭС типа "in-house" для внутреннего пользования. Эти системы интегрируют опыт специалистов компании по ключевым и стратегически важным технологиям. В начале 90-х гг. появилась новая наука— *"управление знаниями"* (knowledge management), ориентированная на методы обработки и управления корпоративными знаниями (Borghoff, 1998; Гаврилова, Хорошевский, 2001).

Современные ЭС — это сложные программные комплексы, аккумулирующие знания специалистов в конкретных предметных областях и распространяющие этот эмпирический опыт для консультирования менее квалифицированных пользователей. Разработка экспертных систем, как активно развивающаяся ветвь информатики, направлена на использование ЭВМ для обработки информации в тех областях науки и техники, где традиционные математические методы моделирования малопригодны. В этих областях важна смысловая и логическая обработка информации, важен опыт экспертов.

Основные факторы, влияющие на целесообразность и эффективность разработки ЭС (частично из [Уотермен, 1989]):

- нехватка специалистов, затрачивающих значительное время для оказания помощи другим;
- выполнение небольшой задачи требует многочисленного коллектива специалистов, поскольку ни один из них не обладает достаточным знанием;
- сниженная производительность, поскольку задача требует полного анализа сложного набора условий, а обычный специалист не в состоянии просмотреть (за отведенное время) все эти условия;
- большое расхождение между решениями самых хороших и самых плохих исполнителей;
- наличие экспертов, готовых поделиться своим опытом. Подходящие задачи

имеют следующие характеристики:

- не могут быть решены средствами традиционного математического моделирования;
- имеется "шум" в данных — некорректность определений, неточность, неполнота, противоречивость информации;
- условий, ограничений;
- являются узкоспециализированными;
- не зависят в значительной степени от общечеловеческих знаний или соображений здравого смысла;
- не являются для эксперта ни слишком легкими, ни слишком сложными. (Время, необходимое эксперту для решения проблемы, может составлять от трех часов до трех недель.)

Хотя экспертные системы достаточно молоды — первые системы такого рода MYCIN появились в США в середине 70-х годов. В настоящее время в мире насчитывается несколько тысяч промышленных ЭС, которые дают советы:

- при управлении сложными диспетчерскими пультами, например, сети распределения электроэнергии;
- при постановке медицинских диагнозов;
- при поиске неисправностей в электронных приборах, диагностика отказов контрольно-измерительного оборудования;
- по проектированию интегральных схем;
- по управлению перевозками;
- по прогнозу военных действий;
- по формированию портфеля инвестиций, оценке финансовых рисков, налогообложению и т. д.

Наиболее популярные приложения ИС отражены на рис. 1.11 [Durkin, 1998].

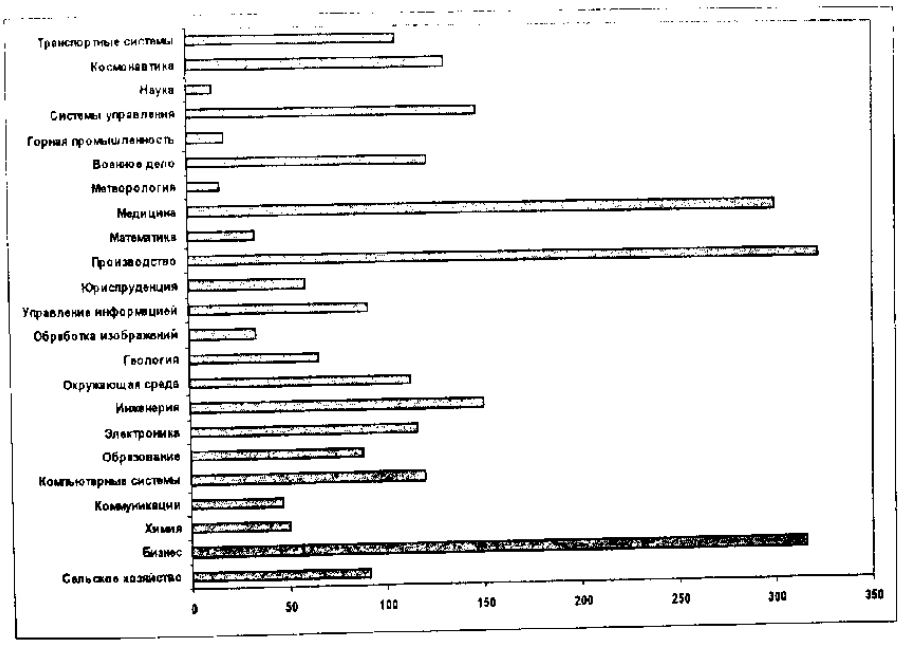

Рис. 1.11. Основные приложения ИС

Сейчас легче назвать области, не использующие ЭС, чем те, где они уже применяются. Уже в 1987 году опрос пользователей, проведенный журналом "Intelligent Technologies" (США), показал, что примерно:

- 25% пользователей используют ЭС;
- 25% собираются приобрести ЭС в ближайшие 2—3 года;
- 50% предпочитают провести исследование об эффективности их использования.

Главное отличие ИС и ЭС от других программных средств — это наличие базы знаний (БЗ), в которой знания хранятся в форме, понятной специалистам предметной области, и могут быть изменены и дополнены также в понятной форме. Это и есть языки представления знаний — ЯПЗ.

До последнего времени именно различные ЯПЗ были центральной проблемой при разработке ЭС.

Для перечисленных в *разд. 1.2* моделей существует соответствующая математическая нотация, разработаны системы программирования, реализующие эти ЯПЗ, и имеется большое количество реальных коммерческих ЭС. Подробнее вопросы программной реализации прикладных ИС рассмотрены в книге далее.

В России в исследования и разработку ЭС большой вклад внесли работы Д. А. Поспелова (основателя Российской ассоциации искусственного интеллекта и его первого президента), Э. В. Попова, В. Ф. Хорошевского, В. Л. Стефанюка, Г. С. Осипова, В. К. Финна, В. Л. Вагина, В. И. Городецкого и многих других.

Современное состояние разработок в области ЭС в России можно охарактеризовать как стадию все возрастающего интереса среди широких слоев специалистов — финансистов, топ-менеджеров, преподавателей, инженеров, медиков, психологов, программистов, лингвистов. В последние годы этот интерес имеет пока достаточно слабое материальное подкрепление — явная нехватка учебников и специальной литературы, отсутствие символьных процессоров и рабочих станций, ограниченное финансирование исследований в этой области, слабый отечественный рынок программных продуктов для разработки ЭС.

Поэтому появляется возможность распространения "подделок" под экспертные системы в виде многочисленных диалоговых систем и интерактивных пакетов прикладных программ, которые дискредитируют в глазах пользователей это чрезвычайно перспективное направление. Процесс создания экспертной системы требует участия высококвалифицированных специалистов в области искусственного интеллекта, которых пока готовит небольшое количество высших учебных заведений страны.

Наибольшие трудности в разработке ЭС вызывает сегодня не процесс машинной реализации систем, а домашинный этап анализа знаний и проектирования базы знаний. Этим занимается специальная наука — инженерия знаний *(см. гл. 2)*.

Обобщенная структура экспертной системы представлена на рис. 1.12. Следует учесть, что реальные ЭС могут иметь более сложную структуру, однако блоки, изображенные на рисунке, непременно присутствуют в любой действительно экспертной системе, поскольку представляют собой стандарт де-факто структуры современной ЭС.

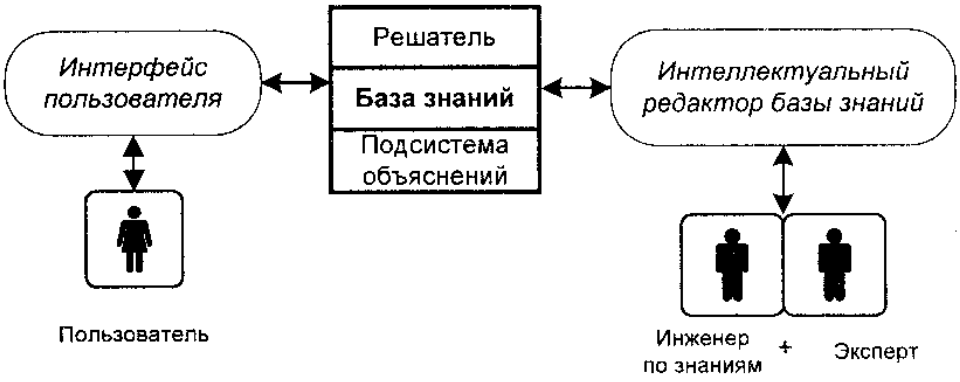

Рис. 1.12. Структура экспертной системы

В целом процесс функционирования ЭС можно представить следующим образом: пользователь, желающий получить необходимую информацию, через пользовательский интерфейс посылает запрос к ЭС; решатель, пользуясь базой знаний, генерирует и выдает пользователю подходящую рекомендацию, объясняя ход своих рассуждений при помощи подсистемы объяснений.

Так как терминология в области разработки ЭС постоянно модифицируется, определим основные термины в рамках данной книги:

- *Пользователь*— специалист предметной области, для которого предназначена система. Обычно его квалификация недостаточно высока и поэтому он нуждается в помощи и поддержке своей деятельности со стороны ЭС.
- *Инженер по знаниям* — специалист в области искусственного интеллекта, выступающий в роли промежуточного буфера между экспертом и базой знаний. Синонимы: *когнитолог, инженер-интерпретатор, аналитик.*
- *Интерфейс пользователя* — комплекс программ, реализующих диалог пользователя с ЭС как на стадии ввода информации, так и при получении результатов.
- *База знаний* (БЗ) — ядро ЭС, совокупность знаний предметной области, записанная на машинный носитель в форме, понятной эксперту и пользователю (обычно на некотором языке, приближенном к естественному).

Параллельно такому "человеческому" представлению существует БЗ во внутреннем "машинном" представлении.

- *Решатель* — программа, моделирующая ход рассуждений эксперта на основании знаний, имеющихся в БЗ. Синонимы: *дедуктивная машина, машина вывода, блок логического вывода.*
- *Подсистема объяснений* — программа, позволяющая пользователю получить ответы на вопросы: "Как была получена та или иная рекомендация?" и "Почему система приняла такое решение?" Ответ на вопрос "как" — это трассировка всего процесса получения решения с указанием использованных фрагментов БЗ, т. е. всех шагов цепи умозаключений. Ответ на вопрос "почему" — ссылка на умозаключение, непосредственно предшествовавшее полученному решению, т. е. отход на один шаг назад. Развитые подсистемы объяснений поддерживают и другие типы вопросов.
- *Интеллектуальный редактор БЗ* — программа, представляющая инженеру по знаниям возможность создавать БЗ в диалоговом режиме. Включает в себя систему вложенных меню, шаблонов языка представления знаний, подсказок ("help" — режим) и других сервисных средств, облегчающих работу с базой.

Еще раз следует подчеркнуть, что представленная на рис. 1.12 структура является минимальной, что означает обязательное присутствие указанных на ней блоков. Если система объявлена

разработчиками как экспертная, только наличие всех этих блоков гарантирует реальное использование аппарата обработки знаний. Однако промышленные прикладные ЭС могут быть существенно сложнее и дополнительно включать базы данных, интерфейсы обмена данными с различными пакетами прикладных программ, электронными библиотеками и т. д.

1.6. Классификация экспертных систем

Существуют различные подходы к классификации экспертных систем, т. к. класс ЭС сегодня объединяет несколько тысяч различных программных комплексов, которые можно классифицировать по десятку критериев. Полезными могут оказаться классификации, представленные на рис. 1.13.

1.6.1. Классификация по решаемой задаче

Традиционно ЭС решают следующие классы задач (примеры взяты из [Попов и др., 1996; Adeli, 1994]):

- *Интерпретация данных.* Это одна из традиционных задач для экспертных систем. Под интерпретацией понимается процесс определения смысла данных, результаты которого должны быть согласованными и корректными. Обычно предусматривается много вариантный анализ данных.

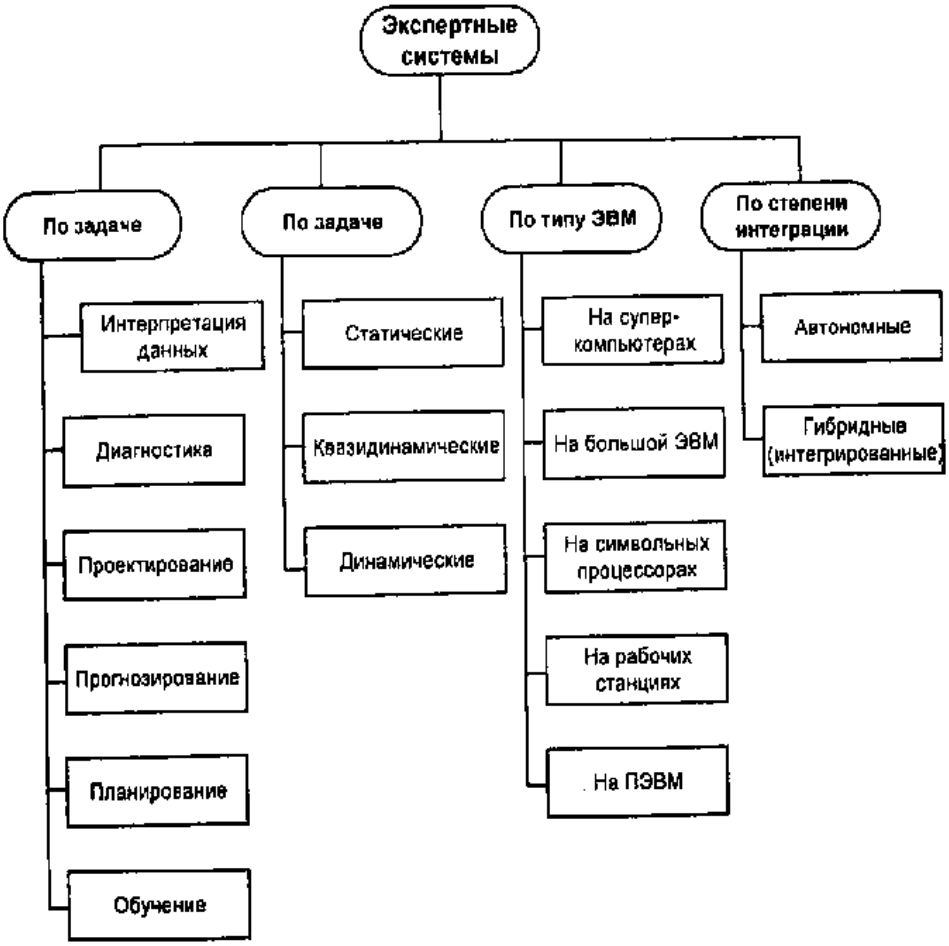

Рис. 1.13. Классификация экспертных систем

Например, обнаружение и идентификация различных типов океанских судов по результатам аэрокосмического сканирования — SIAP; определение основных свойств личности по результатам психодиагностического тестирования в системах АВТАНТЕСТ и МИКРОЛЮШЕР и др.

- *Диагностика.* Под диагностикой понимается процесс соотнесения объекта с некоторым классом объектов и/или обнаружение неисправности в некоторой системе. Неисправность — это отклонение от нормы. Такая трактовка позволяет с единых теоретических позиций рассматривать и неисправность оборудования в технических системах, и заболевания живых организмов, и всевозможные природные аномалии. Важной спецификой является здесь необходимость понимания функциональной структуры ("анатомии") диагностирующей системы. Например: диагностика и терапия

сужения коронарных сосудов — ANGY; диагностика ошибок в аппаратуре и математическом обеспечении ЭВМ — система CRIB и др.

- *Мониторинг.* Основная задача мониторинга — непрерывная интерпретация данных в реальном масштабе времени и сигнализация о выходе тех или иных параметров за допустимые пределы. Главные проблемы — "пропуск" тревожной ситуации и инверсная задача "ложного" срабатывания. Сложность этих проблем в размытости симптомов тревожных ситуаций и необходимость учета временного контекста. Например: контроль за работой электростанций СПРИНТ, помощь диспетчерам атомного реактора — REACTOR; контроль аварийных датчиков на химическом заводе — FALCON и др.
- *Проектирование.* Проектирование состоит в подготовке спецификаций на создание "объектов" с заранее определенными свойствами. Под спецификацией понимается весь набор необходимых документов — чертеж, пояснительная записка и т. д. Основные проблемы здесь — получение четкого структурного описания знаний об объекте и проблема "следа". Для организации эффективного проектирования и, в еще большей степени, перепроектирования необходимо формировать не только сами проектные решения, но и мотивы их принятия. Таким образом, в задачах проектирования тесно связываются два основных процесса, выполняемых в рамках соответствующей ЭС: *процесс вывода решения и процесс объяснения.* Например: проектирование конфигураций ЭВМ VAX — 11/780 в системе XCON (или R1), проектирование БИС — CADHELP; синтез электрических цепей — SYN и др.
- *Прогнозирование.* Прогнозирование позволяет предсказывать последствия некоторых событий или явлений на основании анализа имеющихся данных. Прогнозирующие системы логически выводят вероятные следствия из заданных ситуаций. В прогнозирующей системе обычно используется параметрическая динамическая модель, в которой значения параметров "подгоняются" под заданную ситуацию. Выводимые из этой модели следствия составляют основу для прогнозов с вероятностными оценками. Например: предсказание погоды — система WILLARD; оценки будущего урожая — PLANT; прогнозы в экономике — ECON и др.
- *Планирование.* Под планированием понимается нахождение планов действий, относящихся к объектам, способным выполнять некоторые функции. В таких ЭС используются модели поведения реальных объектов с тем, чтобы логически вывести последствия планируемой деятельности. Например: планирование поведения робота— STRIPS; планирование промышленных заказов — 1SIS; планирование эксперимента — MOLGEN и др.
- *Обучение.* Под обучением понимается использование компьютера для обучения какой-то дисциплине или предмету. Системы обучения диагностируют ошибки при изучении какой-либо дисциплины с помощью ЭВМ и подсказывают правильные решения. Они аккумулируют знания о гипотетическом "ученике" и его характерных ошибках, затем в работе они способны диагностировать слабости в познаниях обучаемых и находить соответствующие средства для их ликвидации. Кроме того, они планируют акт общения с учеником в зависимости от успехов ученика с целью передачи знаний. Например: обучение языку программирования LISP в системе "Учитель LISP"; система PROUST — обучение языку Паскаль и др.
- *Управление.* Под управлением понимается функция организованной системы, поддерживающая определенный режим деятельности. Такого рода ЭС осуществляют управление поведением сложных систем в соответствии с заданными спецификациями. Например: помощь в управлении газовой котельной — GAS; управление системой календарного планирования Project Assistant и др.
- *Поддержка принятия решений.* Поддержка принятия решения — это совокупность процедур, обеспечивающая принимающего решения индивидуума необходимой информацией и рекомендациями, облегчающими процесс принятия решения. Эти ЭС помогают специалистам выбрать и/или сформировать нужную альтернативу среди множества выборов при принятии ответственных решений. Например: выбор стратегии выхода фирмы из кризисной ситуации — CRYSIS; помощь в выборе страховой компании или инвестора — CHOICE и др.

В общем случае все системы, основанные на знаниях, можно подразделить *на системы, решающие задачи анализа и на системы, решающие задачи синтеза.* Основное отличие задач анализа от задач синтеза заключается в том, что если в задачах анализа множество решений может быть перечислено и включено в систему, то в задачах синтеза множество решений потенциально не ограничено и строится из решений компонентов или подпроблем. Задачами анализа являются: интерпретация данных, диагностика, поддержка принятия решения; к задачам синтеза относятся

проектирование, планирование, управление. Комбинированные: обучение, мониторинг, прогнозирование.

1.6.2. Классификация по связи с реальным временем

- *Статические ЭС* разрабатываются в предметных областях, в которых база знаний и интерпретируемые данные не меняются во времени. Они стабильны.

Пример: диагностика неисправностей в автомобиле.

- *Квазидинамические ЭС* интерпретируют ситуацию, которая меняется с некоторым фиксированным интервалом времени.

Пример: микробиологические ЭС, в которых снимаются лабораторные измерения с технологического процесса один раз в 4—5 часов (производство лизина, например) и анализируется динамика полученных показателей по отношению к предыдущему измерению.

- *Динамические ЭС* работают в сопряжении с датчиками объектов в режиме реального времени с непрерывной интерпретацией поступающих в систему данных.

Пример: управление гибкими производственными комплексами, мониторинг в реанимационных палатах.

Программный инструментарий для разработки динамических систем — G2 [Попов, 1996].

1.6.3.Классификация по типу ЭВМ

На сегодняшний день существуют:

- ЭС для уникальных стратегически важных задач на суперЭВМ (Эльбрус, CRAY, CONVEX и др.);
- ЭС на ЭВМ средней производительности (mainframe);
- ЭС на символьных процессорах и рабочих станциях (SUN, Silicon Graphics, APOLLO);
- ЭС на персональных компьютерах (IBM-совместимые, Macintosh).

1.6.4.Классификация по степени интеграции с другими программами

- *Автономные ЭС* работают непосредственно в режиме консультаций с пользователем для специфически "экспертных" задач, для решения которых не требуется привлекать традиционные методы обработки данных (расчеты, моделирование и т. д.).
- *Гибридные ЭС* представляют программный комплекс, агрегирующий стандартные пакеты прикладных программ (например, математическую статистику, линейное программирование или системы управления базами данных) и средства манипулирования знаниями. Это может быть интеллектуальная надстройка над ППП (пакетами прикладных программ) или интегрированная среда для решения сложной задачи с элементами экспертных знаний. Несмотря на внешнюю привлекательность гибридного подхода следует отметить, что разработка таких систем являет собой задачу на порядок более сложную, чем разработка автономной ЭС. Стыковка не просто разных пакетов, а различных методологий (что происходит в гибридных системах) порождает целый комплекс теоретических и практических трудностей.

1.7. Разработка экспертных систем

На сегодняшний день следует констатировать, что разработка программных комплексов экспертных систем как за рубежом, так и в нашей стране осталась скорее на уровне искусства, чем науки. Это связано с тем, что долгое время системы искусственного интеллекта внедрялись в основном во время фазы проектирования, а чаще всего разрабатывалось несколько прототипных версий программ, и на их основе уже создавался конечный продукт. Такой подход действует хорошо в исследовательских условиях, однако в коммерческих условиях он является слишком дорогим, чтобы оправдать затраты на разработку.

Процесс разработки промышленной экспертной системы, опираясь на традиционные технологии [Николов и др., 1990; Хейес-Рот и др., 1987; Tuthill, 1994], практически для любой предметной области можно разделить на шесть более или менее независимых этапов (рис. 1.14).

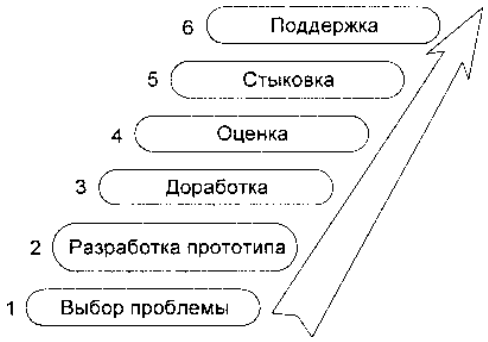

Рис. 1.14. Этапы разработки ЭС

Последовательность этапов дана только с целью получения общего представления о процессе создания идеального проекта, т. к. последовательность эта не вполне фиксирована. В действительности каждый последующий этап разработки может принести новые идеи, которые могут повлиять на предыдущие решения и даже привести к их переработке. Именно поэтому многие специалисты по информатике весьма критично относятся к методологии экспертных систем. Они считают, что расходы на разработку таких систем очень большие, время разработки слишком велико, а полученные в результате программы накладывают тяжелое бремя на вычислительные ресурсы.

В целом за разработку экспертных систем целесообразно браться организации, где накоплен опыт по автоматизации рутинных процедур обработки информации, таких как:

- формирование корпоративных информационных систем;
- организация сложных расчетов;
- работа с компьютерной графикой;
- обработка текстов и автоматизированный документооборот.

Решение таких задач, во-первых, подготавливает высококвалифицированных специалистов по информатике, необходимых для создания интеллектуальных систем, во-вторых, позволяет отделить от экспертных систем неэкспертные задачи.

1.7.1. Выбор подходящей проблемы

Этот этап определяет деятельность, предшествующую решению начать разрабатывать конкретную ЭС. Он включает [Николов и др., 1990]:

- определение проблемной области и задачи;
- нахождение эксперта, желающего сотрудничать при решении проблемы, и назначение коллектива разработчиков;
- определение предварительного подхода к решению проблемы;
- анализ расходов и прибылей от разработки;
- подготовку подробного плана разработки.

Правильный выбор проблемы представляет самую критическую часть разработки в целом. Если выбрать неподходящую проблему, можно очень быстро увязнуть в "болоте" проектирования задач, которые никто не знает, как решать. Неподходящая проблема может также привести к созданию экспертной системы, затраты на которую выше приносимой ею экономии. Дело будет обстоять еще хуже, если разработать систему, которая работает, но неприемлема для пользователей. Даже если разработка выполняется самой организацией для собственных целей, эта фаза является подходящим моментом для получения рекомендаций извне, чтобы гарантировать удачно выбранный и осуществимый с технической точки зрения первоначальный проект.

При выборе области применения следует учитывать, что если знание, необходимое для решения задач, постоянное, четко формулируемое, и связано с вычислительной обработкой, то обычные алгоритмические программы, по всей вероятности, будут самым целесообразным способом решения проблем в этой области.

Экспертная система ни в коем случае не устранит потребность в реляционных базах данных, статистическом программном обеспечении, электронных таблицах и системах текстовой обработки. Но если результативность задачи зависит от знания, которое является субъективным,

изменяющимся, символьным или вытекающим частично из соображений здравого смысла, тогда область может обоснованно выступать претендентом на экспертную систему.

Обычно экспертные системы разрабатываются путем получения специфических знаний от эксперта и ввода их в систему. Некоторые системы могут содержать стратегии одного индивида. Следовательно, найти подходящего эксперта — это ключевой шаг в создании экспертных систем.

В процессе разработки и последующего расширения системы инженер по знаниям и эксперт обычно работают вместе. Инженер по знаниям помогает эксперту структурировать знания, определять и формализовать понятия и правила, необходимые для решения проблемы.

Во время первоначальных бесед они должны решить, будет ли их сотрудничество успешным. Это немаловажно, поскольку обе стороны будут работать совместно, по меньшей мере, в течение одного года. Кроме них в коллектив разработчиков целесообразно включить потенциальных пользователей и профессиональных программистов. Подробно функции каждого члена коллектива описаны в следующем разделе.

Предварительный подход к программной реализации задачи определяется, исходя из характеристик задачи и ресурсов, выделенных на ее решение. Инженер по знаниям выдвигает обычно несколько вариантов, связанных с использованием имеющихся на рынке программных средств. Окончательный выбор возможен лишь на этапе разработки прототипа.

После того как задача определена, необходимо подсчитать расходы и прибыль от разработки экспертной системы. В расходы включаются затраты на оплату труда коллектива разработчиков. В дополнительные расходы будет включена стоимость приобретаемого программного инструментария, с помощью которого будет разработана экспертная система.

Прибыль может быть получена за счет снижения цены продукции, повышения производительности труда, расширения номенклатуры продукции или услуг или даже разработки новых видов продукции или услуг в области, в которой будет использоваться ЭС. Соответствующие расходы и прибыль от системы определяются относительно времени, в течение которого возвращаются средства, вложенные в разработку. На современном этапе большая часть фирм, развивающих крупные экспертные системы, предпочли разрабатывать дорогостоящие проекты, приносящие значительную прибыль.

Можно ожидать развития тенденции разработки менее дорогостоящих систем, хотя и с более длительным сроком окупаемости вложенных в них средств, т. к. программные средства разработки экспертных систем непрерывно совершенствуются.

После того как инженер по знаниям убедился, что:

- данная задача может быть решена с помощью экспертной системы;
- экспертную систему можно создать предлагаемыми на рынке средствами;
- имеется подходящий эксперт;
- предложенные критерии производительности являются разумными;
- затраты и срок их окупаемости приемлемы для заказчика,

он составляет план разработки. План определяет шаги процесса разработки и необходимые затраты, а также ожидаемые результаты.

1.7.2. Разработка прототипа

Прототипная система является усеченной версией экспертной системы, спроектированной для проверки правильности кодирования фактов, связей и стратегий рассуждения эксперта. Она также дает возможность инженеру по знаниям привлечь эксперта к активному участию в процессе разработки экспертной системы, и, следовательно, к принятию им обязательства приложить все усилия к созданию системы в полном объеме.

Объем прототипа — несколько десятков правил, фреймов или примеров. На рис. 1.15 изображены шесть стадий разработки прототипа и минимальный коллектив разработчиков, занятых на каждой из стадий (пять стадий заимствованы из [Хейес-Рот и др., 1987]). Приведем краткую характеристику всех стадий, хотя эта схема представляет собой грубое приближение к сложному, итеративному процессу.

Несмотря на то, что любое теоретическое разделение бывает часто условным, осознание коллективом разработчиков четких задач каждой стадии представляется целесообразным. Роли разработчиков (эксперт, программист, пользователь и аналитик) являются постоянными на протяжении всей разработки. Совмещение ролей нежелательно.

Сроки приведены условно, т. к, зависят от квалификации специалистов и особенностей задачи.

Идентификация проблемы

Уточняется формулировка (спецификация) задачи, планируется ход разработки прототипа экспертной системы, определяются:

- необходимые ресурсы (время, люди, компьютеры, деньги и т. д.);
- источники знаний (книги, дополнительные эксперты, методики);
- имеющиеся аналогичные экспертные системы;
- цели (распространение опыта, автоматизация рутинных действий и др.);
- классы решаемых задач и т. д.

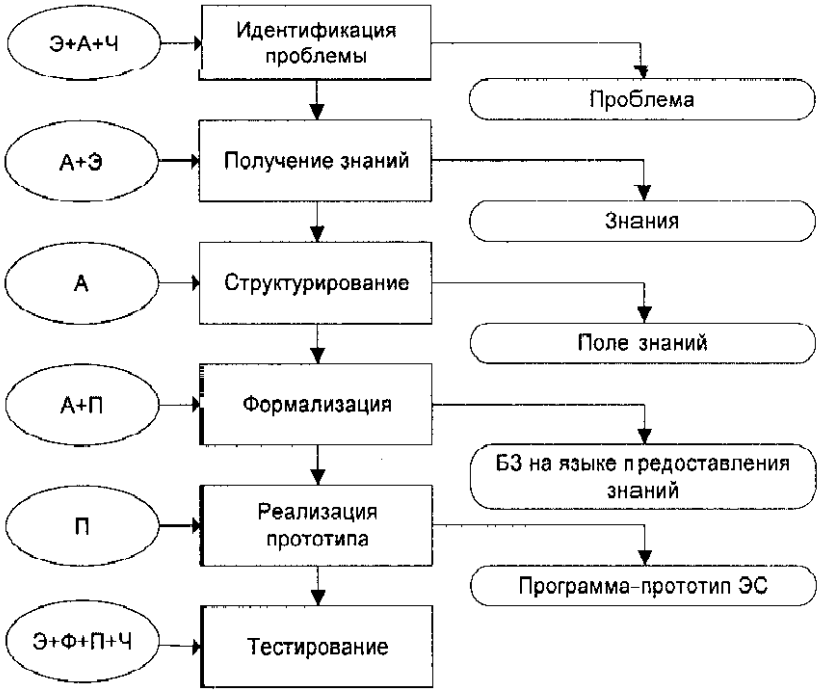

Рис. 1.15. Стадии разработки прототипа ЭС: Э — эксперт; А — аналитик (инженер по знаниям); П — программист; Ч — пользователь ("чайник")

Определение 1.10

Идентификация проблемы **— знакомство и обучение членов коллектива разработчиков, а также создание неформальной спецификации задачи.**

Средняя продолжительность 1—2 недели.

Извлечение знаний

На этой стадии происходит перенос компетентности от эксперта к инженеру по знаниям с использованием различных методов *(см. гл. 2):*

- анализ текстов;
- диалоги;
- экспертные игры;
- лекции;
- дискуссии;
- интервью;
- наблюдение и др.

Определение 1.11

Извлечение знаний **— получение инженером по знаниям наиболее полного из возможных представлений о предметной области и способах принятия решения в ней.**

Средняя продолжительность 1—3 месяца.

Структурирование или концептуализация знаний

Через выявление структуры полученных знаний о предметной области определяются:

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

Определение 1.12

***Структурирование (или концептуализация) знаний* — разработка неформального наглядного описания знаний о предметной области в виде графа, таблицы, диаграммы или текста, которое отражает основные концепции и взаимосвязи между понятиями предметной области.**

Такое описание называется *полем знаний.*

Средняя продолжительность этапа 2—4 недели. Подробно стадия структурирования описана в *разд. 2.6.*

Формализация знаний

Строится формализованное представление концепций предметной области на основе выбранного языка представления знаний (ЯПЗ). Традиционно на этом этапе используются:

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

Определение 1.13

***Формализация знаний* — разработка 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.**

Все чаще на этой стадии используется симбиоз языков представления знаний, например, в системе ОМЕГА [Справочник по ИИ, 1990] — фреймы + семантические сети + полный набор возможностей языка исчисления предикатов. Средняя продолжительность 1—2 месяца. Подробнее — далее в книге.

Программная реализация

Создается прототип экспертной системы, включающий базу знаний и остальные блоки, при помощи одного из следующих способов:

- программирование на традиционных языках типа Pascal, C++ и др.;
- программирование на специализированных языках, применяемых в задачах искусственного интеллекта: LISP [Хювянен, Сеппянен, 1991], FRL [Байдун, Бунин, 1990], SMALLTALK [Справочник по ИИ, 1990] и др.;
- использование инструментальных средств разработки ЭС PIES [Хорошевский, 1993]; G2 [Попов, Фоминых, Кисель, 1996];
- использование "пустых" ЭС или "оболочек" типа ЭКСПЕРТ [Кирсанов, Попов, 1990], ФИАКР [Соловьев, Соловьева, 1989] и др.

Определение 1.14

***Программная реализация*— разработка программного комплекса, демонстрирующего жизнеспособность подхода в целом.**

Чаще всего первый прототип отбрасывается на этапе реализации действующей ЭС.

Средняя продолжительность 1—2 месяца. Более подробно эти вопросы рассматриваются в следующих главах.

Тестирование

Оценивается и проверяется работа программ прототипа с целью приведения в соответствие с реальными запросами пользователей. Прототип проверяется:

- на удобство и адекватность интерфейсов ввода/вывода (характер вопросов в диалоге, связность выводимого текста результата и др.);
- на эффективность стратегии управления (порядок перебора, использование нечеткого вывода и др.);
- на качество проверочных примеров;
- на корректность базы знаний (полнота и непротиворечивость правил).

Определение 1.15

***Тестирование* — выявление ошибок в подходе и реализации прототипа и выработка рекомендаций по доводке системы до промышленного варианта.**

Средняя продолжительность 1—2 недели.

1.7.3. Развитие прототипа до промышленной ЭС

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 неподходящими, их необходимо изменить. Можно сделать оценку общего числа эвристических правил, необходимых для создания окончательного варианта экспертной системы. Иногда [Хювянен, Сеппянен, 1991] при разработке промышленной и/или коммерческой системы выделяют следующие дополнительные этапы для перехода (табл. 1.2):

- демонстрационный прототип;
- исследовательский прототип;
- действующий прототип;
- промышленная система;
- коммерческая система.

Однако чаще реализуется плавный переход от демонстрационного прототипа к промышленной системе, при этом, если программный инструментарий был выбран удачно, не обязательно даже переписывать окончательный вариант другими программными средствами.

Понятие же коммерческой системы в нашей стране входит в понятие "промышленный программный продукт", или "промышленная ЭС" (в этой работе).

Таблица 1.2. *Переход от прототипа к промышленной экспертной системе*

Этапы развития прототипа	**Функциональность прототипа**
Демонстрационный прототип ЭС	Система решает часть задач, демонстрируя жизнеспособность подхода (несколько десятков правил или понятий)
Исследовательский прототип ЭС	Система решает большинство задач, но неустойчива в работе и не полностью проверена (несколько сотен правил или понятий)
Действующий прототип ЭС	Система надежно решает все задачи на реальных примерах, но для сложной задачи требует много времени и памяти

Промышленная система	Система обеспечивает высокое качество решений при минимизации требуемого времени и памяти; переписывается с использованием более эффективных средств представления знаний
Коммерческая система	Промышленная система, пригодная к продаже, хорошо документирована и снабжена сервисом

Основная работа на данном этапе заключается в существенном расширении базы знаний, т. е. 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 трудноуловимых 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 особое 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 право собственности и контроля системы эксперту для уточнения, детальной разработки и обслуживания.

1.7.4. Оценка системы

После завершения этапа разработки промышленной экспертной системы необходимо провести ее тестирование в отношении критериев эффективности. К тестированию широко привлекаются другие эксперты с целью апробирования работоспособности системы на различных примерах. Экспертные системы оцениваются главным образом для того, чтобы проверить точность работы программы и ее полезность. Оценку можно проводить, исходя из различных критериев, которые сгруппируем следующим образом:

- критерии пользователей (понятность и "прозрачность" работы системы, удобство интерфейсов и др.);
- критерии приглашенных 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, когда система не может принять решение, анализ чувствительности программы к незначительным изменениям в представлении знаний, весовых коэффициентах, применяемых в механизмах логического вывода, данных и т. п.).

1.7.5. Стыковка системы

На этом этапе осуществляется стыковка экспертной системы с другими программными средствами в среде, в которой она будет работать, и обучение людей, которых она будет обслуживать. Иногда это означает внесение существенных изменений. Такие изменения требуют непременного 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 выполняет эти задачи. Для того чтобы система была одобрена, необходимо 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).

Так успешно была состыкована со своим окружением система PUFF — экспертная система для диагностики заболеваний легких [Хейес-Рот и др., 1987]. После того как PUFF была закончена и все были удовлетворены ее работой, систему перекодировали с LISP на Бейсик. Затем систему перенесли на ПЭВМ, которая уже работала в больнице. В свою очередь, эта ПЭВМ была связана с измерительными приборами. Данные с измерительных приборов сразу поступают в ПЭВМ. PUFF обрабатывает эти данные и печатает рекомендации для врача. Врач в принципе не взаимодействует с PUFF. Система полностью интегрирована со своим окружением — она представляет собой интеллектуальное расширение аппарата исследования легких, который врачи давно используют.

Другой системой, которая хорошо функционирует в своем окружении, являлась CAT-1 [Уотермен, 1990] — экспертная система для диагностики неисправностей дизелей локомотивов.

Эта система была разработана также на языке LISP, а затем была переведена на FORTH с тем, чтобы ее можно было более эффективно использовать в различных локомотивных цехах. Мастер по ремонту запрашивает систему о возможных причинах неисправности дизеля. Система связана с видеодиском, с помощью которого мастеру показывают визуальные объяснения и подсказки, касающиеся более подробных проверок, которые он должен сделать.

Кроме того, если оператор не уверен в том, как устранить неисправность, система предоставляет ему обучающие материалы, которые фирма подготовила предварительно, и показывает ему их на видеотерминале. Таким образом, мастер по ремонту может с помощью экспертной системы диагностировать проблему, найти тестовую процедуру, которую он должен использовать, получить на дисплее объяснение, как провести тест, или получить инструкции о том, как справиться с возникшей проблемой.

1.7.6. Поддержка системы

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

Удачным и ставшим уже хрестоматийным примером ЭС, внедренной таким образом, является XCON (R1) — ЭС, которую фирма DEC использовала для комплектации ЭВМ семейства VAX. Одной из ключевых проблем, с которой столкнулась фирма DEC, являлась необходимость постоянного внесения изменений для новых версий оборудования, новых спецификаций и т. д. Для этой цели XCON поддерживается в программной среде OPS5.

1.8. Человеческий фактор при разработке ЭС

При разработке ЭС наиболее критическим фактором является человеческий, поскольку разработка таких систем требует высочайшей квалификации от коллектива разработчиков.

Под коллективом разработчиков (КР) будем понимать группу специалистов, ответственных за создание ЭС.

Как видно из рис. 1.12, в состав КР входят, по крайней мере, три человека — пользователь, эксперт и инженер по знаниям. В действительности, безусловно, нужен как минимум один программист, и обязательно надо привлекать к работе заказчика, хотя бы на ранних стадиях. Таким образом, минимальный состав КР включает пять человек; реально же он разрастается до 8—10 человек. Численное увеличение коллектива разработчиков происходит по следующим причинам:

- необходимость учета мнения нескольких пользователей;

- привлечение к экспертизе нескольких экспертов;
- потребность как в прикладных, так и системных программистах.

На Западе в этот коллектив дополнительно традиционно включают менеджера и одного технического помощника.

Если использовать аналогии из близких областей, то КР более всего схож с группой администратора базы данных при построении интегрированных информационных систем или бригадой программистов, разрабатывающих сложный программный комплекс. При отсутствии профессионального менеджера руководителем КР, участвующим во всех стадиях разработки, является инженер по знаниям, поэтому к его квалификации предъявляются самые высокие требования. В целом уровень и численность группы зависят от характеристик поставленной задачи.

Практически все психологи отмечают, что на любой коллективный процесс влияет атмосфера, возникающая в группе. Существуют эксперименты, результаты которых неоспоримо говорят, что часто дружеская атмосфера в коллективе больше влияет на результат, чем индивидуальные способности отдельных членов группы [Немов, 1984]. Особенно важно, чтобы в коллективе разработчиков складывались кооперативные, а не конкурентные отношения. Для кооперации характерна атмосфера сотрудничества, взаимопомощи, заинтересованности в успехах друг друга, т. е. уровень нравственного общения, а для отношений конкурентного типа — атмосфера индивидуализма и межличностного соперничества (более низкий уровень общения).

В настоящее время, прогнозировать совместимость в общении со 100%-ной гарантией невозможно. Однако можно выделить ряд факторов и черт личности, характера и других особенностей участников общения, несомненно, оказывающих влияние на эффективность процедуры, что в том числе обусловлено и психологической совместимостью членов группы. Следовательно, при формировании КР желательно учитывать психологические свойства участников.

В настоящий момент в психологии существуют несколько десятков методик по определению свойств личности, широко используемых в вопросах профессиональной ориентации. Эти психодиагностические методики, часть из которых уже автоматизирована, различаются направленностью, глубиной, временем опроса и способами интерпретации. В частности, система АВАНТЕСТ (Автоматический Анализ тестов) [Гаврилова, 1984] позволяет моделировать рассуждения психолога при анализе результатов тестирования по 16-факторному опроснику Р. Кэттела и выдает связное психологическое заключение на естественном русском языке, характеризующее такие свойства личности, как общительность, аналитичность, добросовестность, самоконтроль и т. п.

Рассмотрим минимальные требования. Ниже приведены два аспекта характеристик членов КР: А — психофизиологический, Б — профессиональный.

1.8.1. Пользователь

А. К пользователю практически профессиональных требований не предъявляют, поскольку его не выбирают. Он является в некотором роде заказчиком системы. Желательные качества:

а) дружелюбие;

б) умение объяснить, что же он хочет от системы;

в) отсутствие психологического барьера к применению вычислительной техники;

г) интерес к новому.

От пользователя зависит, будет ли применяться разработанная ЭС. Замечено, что наиболее ярко качества в) и г) проявляются в молодом возрасте, поэтому иногда такие пользователи охотнее используют ЭС, не испытывая при этом комплекса неполноценности оттого, что ЭВМ им что-то подсказывает.

Б. Необходимо, чтобы пользователь имел некоторый базовый уровень квалификации, который позволит ему правильно истолковать рекомендации ЭС. Кроме того, должна быть полная совместимость в терминологии интерфейса к ЭС с той, которая привычна и удобна для пользователя. Обычно требования к квалификации пользователя не очень велики, иначе он переходит в разряд экспертов и совершенно не нуждается в ЭС.

1.8.2. Эксперт

А. Эксперт — чрезвычайно важная фигура в группе КР. В конечном счете, его подготовка определяет уровень компетенции базы знаний. Желательные качества:

а) доброжелательность;

б) готовность поделиться своим опытом;

в) умение объяснить (педагогические навыки);

г) заинтересованность (моральная, а лучше еще и материальная) в успешности разработки.

Возраст эксперта обычно зрелый, что необходимо учитывать всем членам группы. Часто встает вопрос о количестве экспертов. Поскольку проблема совмещения подчас противоречивых знаний остается открытой, обычно с каждым из экспертов работают индивидуально, иногда создавая альтернативные базы.

Другие определения эксперта из "околонаучного" фольклора: "Человек, который перестал думать на том основании, что он знает", "Такой же специалист как у нас есть, но из другого города", "Человек, который знает, что будет завтра, а послезавтра может объяснить, почему этого не случилось".

Б. Помимо, безусловно, высокого профессионализма в выбранной предметной области, желательно знакомство эксперта с популярной литературой по искусственному интеллекту и экспертным системам для того, чтобы эффективнее прошел этап извлечения знаний.

1.8.3.Программист

А. Известно, что программисты обладают самой низкой потребностью в общении среди представителей разных профессий. Однако при разработке ЭС необходим тесный контакт членов группы, поэтому желательны следующие его качества:

а) общительность;

б) способность отказаться от традиционных навыков и освоить новые методы;

в) интерес к разработке.

Б. Поскольку современные ЭС — сложнейшие и дорогостоящие программные комплексы, программисты в КР должны иметь опыт и навыки разработки программ. Обязательно знакомство с основными структурами представления знаний и механизмами вывода, состоянием отечественного и мирового рынка программных продуктов для разработки ЭС и диалоговых интерфейсов.

1.8.4.Инженер по знаниям

А. Существуют такие профессии и виды деятельности, для которых природные качества личности (направленность, способности, темперамент) могут иметь характер абсолютного показания или противопоказания к занятиям. По-видимому, инженерия знаний принадлежит к таким профессиям. По различным оценкам это одна из самых малочисленных, высокооплачиваемых и дефицитных в мире специальностей. Попытаемся дать наброски к портрету инженера по знаниям (без претензии на полноту и точность определений).

Пол. Психологи утверждают, что мужчины более склонны к широкому охвату явлений и в среднем у них выше аналитичность, чрезвычайно полезная инженеру по знаниям, которому надо иметь развитое логическое мышление и умение оперировать сложными формальными структурами. Кроме того, при общении с экспертами, которые в большинстве своем настроены скептически по отношению к будущей ЭС, инженер по знаниям - мужчина вызывает более высокое доверие со стороны эксперта. С другой стороны, известно, что у женщин в среднем выше коммуникабельность, наблюдательность к отдельным деталям объектов. Так что пол не является окончательным показанием или противопоказанием к данной профессии.

Интеллект. Это понятие вызывает самые бурные споры психологов; существует до 50 определений интеллекта, но с прагматической точки зрения очевидно, что специалист в области искусственного интеллекта должен стремиться к максимальным оценкам по тестам как вербального, так и невербального интеллекта.

Стиль общения. Инженер по знаниям "задает тон" в общении с экспертом, он ведет диалог, и от него, в конечном счете, зависит его продуктивность. Можно выделить два стиля общения: деловой (или жесткий) и дружеский (или мягкий, деликатный). Нам кажется, что дружеский будет заведомо более успешным, т. к. снижает "эффект фасада" у эксперта, раскрепощает его. Деликатность, внимательность, интеллигентность, ненавязчивость, скромность, умение слушать и задавать вопросы, хорошая коммуникабельность и в то же время уверенность в себе — вот рекомендуемый стиль общения. Безусловно, что это дар и искусство одновременно, однако занятия по психологическому тренингу могут дать полезные навыки.

Портрет инженера по знаниям можно было бы дополнить другими характеристиками — широтой взглядов и интересов, артистичностью, чувством юмора, обаянием и т. д.

Интересные результаты были получены в [Воинов, Долныкова, Чудова, 1988] при исследовании психологических особенностей аналитиков. Выяснилось, что одним из отличительных свойств хорошего аналитика является ожидание высокой позитивной оценки при общении, т. е. установка на успех.

Как писал Э. Берн [Берн, 1988], существует 4 типа людей:

1.I am OK, you are OK (я хороший, ты хороший).

2.I am OK, you are not OK (я хороший, ты нехороший).

3.I am not OK, you are OK(я нехороший, ты хороший).

4.I am not OK, you are not OK(я нехороший, ты нехороший).

Очевидно, что успешным является только первый тип.

Несомненным становится вопрос об определении профессиональной пригодности инженеров по знаниям и необходимости предварительного психологического тестирования при подготовке инженеров по знаниям. Здесь только приведем каталог свойств идеального интервьюера [Ноэль, 1978]. На наш взгляд, это вполне подходящий образец портрета инженера по знаниям перед серией свободных диалогов: "Он должен выглядеть здоровым, спокойным, уверенным, внушать доверие, быть искренним, веселым, проявлять интерес к беседе, быть опрятно одетым, ухоженным". Хороший аналитик может личным обаянием и умением скрыть изъяны подготовки. Блестящая краткая характеристика интервьюера приведена в той же работе — *"общительный педант "*.

Б. При определении профессиональных требований к аналитику следует учитывать, что ему необходимы различные навыки и умения для грамотного и эффективного проведения процессов извлечения, концептуализации и формализации знаний.

Инженер по знаниям имеет дело со всеми формами знаний *(см. разд. 1.1):*

Z1 (знания в памяти) => Z2 (знания в книгах) => Z3 (поле знаний) => Z4 (модель знаний) => Z5 (база знаний).

Работа на уровне Z1 требует от инженера по знаниям знакомства с элементами когнитивной психологии и способами репрезентации понятий и процессов в памяти человека, с двумя основными механизмами мышления — логическим и ассоциативным, с такими способами активизации мышления, как игры, мозговой штурм и др., с различными моделями рассуждений.

Изучение и анализ текстов на уровне Z2 подразумевает широкую общенаучную подготовку инженера; знакомство с методами реферирования и аннотирования текстов; владение навыками быстрого чтения, а также текстологическими методами извлечения знаний.

Разработка поля знаний на уровне Z3 требует квалифицированного знакомства с методологией представления знаний, системным анализом, теорией познания, аппаратом многомерного шкалирования, кластерным и факторным анализом.

Разработка формализованного описания Z4 предусматривает предварительное изучение аппарата математической логики и современных языков представления знаний. Модель знаний разрабатывается на основании результатов глубокого анализа инструментальных средств разработки ЭС и имеющихся "оболочек". Кроме того, инженеру по знаниям необходимо владеть методологией разработки ЭС, включая методы быстрого прототипирования.

И наконец, реализация базы знаний Z5, в которой инженер по знаниям участвует вместе с программистом, подразумевает овладение практическими навыками работы на ЭВМ и, возможно, одним из языков программирования.

Так как инженеров по знаниям "выращивают" из программистов, уровень Z5 обычно не вызывает затруднения, особенно если разработка ведется на традиционных языках типа Си или Паскаль. Специализированные языки искусственного интеллекта LISP и Пролог требуют некоторой перестройки архаично-алгоритмического мышления.

Следует констатировать, что поскольку профессиональных аналитиков не готовит ни один вуз, необходима специальная подготовка этих специалистов. Подробную информацию *см. в разд. 2.4.*

Успешность выбора и подготовки коллектива разработчиков ЭС определяет эффективность и продолжительность всего процесса разработки.

ГЛАВА 2 Введение в инженерию знаний

2.1. Определение и структура инженерии знаний

Основные трудности в разработке экспертных систем связаны с проблемой извлечения и структурирования знаний. Именно эти вопросы исследует наука под названием — *инженерия знаний* (knowledge engineering). Это достаточно молодое направление искусственного интеллекта, появившееся тогда, когда практические разработчики столкнулись с весьма нетривиальными проблемами трудности "добычи" и формализации знаний. В первых книгах по искусственному интеллекту (ИИ) эти факты обычно только постулировались, в дальнейшем начались серьезные исследования по выявлению оптимальных стратегий выявления знаний [Boose, 1990; Wielinga, Schreiber, Breuker, 1992; Tuthill, 1994; Adeli, 1994; Leondes, 2000].

Определение 2.1

***Инженерия знаний* — направление исследований и разработок в области интеллектуальных систем, ставящее целью разработку моделей, методов и систем для получения, структурирования и формализации знаний специалистов с целью проектирования баз знаний.**

Основные направления исследований инженерии знаний представлены на рис. 2.1.

2.1.1. Поле знаний

Данная глава целиком посвящена теоретическим проблемам инженерии знаний, другими словами — проектированию баз знаний. Центральным понятием на стадиях получения и структурирования является так называемое *поле знаний,* уже упоминавшееся в *разд. 1.1*

Рис. 2.1. Структура инженерии знаний

Определение 2.2

***Поле знаний*— это условное неформальное описание основных понятий и взаимосвязей между понятиями предметной области, выявленных из системы знаний эксперта, в виде графа, диаграммы, таблицы или текста.**

Поле знаний P_Z формируется на третьей стадии разработки ЭС *(см. разд. 1.7.2)* — стадии структурирования.

Поле знаний, как первый шаг от структурирования к формализации, представляет модель знаний о предметной области в том виде, в каком ее сумел выразить аналитик на некотором "своем" языке.

Обобщенно синтаксическую структуру поля знаний можно представить как

$$Pz = (I, O, M),$$

где I — структура исходных данных, подлежащих обработке и интерпретации в экспертной системе; O — структура выходных данных, т. е. результата работы системы; M— операциональная модель предметной области, на основании которой происходит модификация I в O.

Включение компонентов I и O в P_z обусловлено тем, что составляющие и структура этих интерфейсных компонентов неявно присутствуют в модели репрезентации в памяти эксперта. Операциональная модель M может быть представлена как совокупность концептуальной структуры S_k, отражающей понятийную структуру предметной области, и функциональной структуры Sf, моделирующей схему рассуждений эксперта.

$$M=(S_k,Sf).$$

S_k выступает как статическая, неизменная составляющая P_z, в то время как S_f представляет динамическую, изменяемую составляющую.

Формирование Sk основано на выявлении понятийной структуры предметной области. Далее описан достаточно универсальный алгоритм проведения концептуального анализа на основе модификации парадигмы структурного анализа [Yourdon, 1989] и построения иерархии понятий (так называемая "пирамида знаний"). Пример S_k и S_f представлен на рис. 2.2 и 2.3.

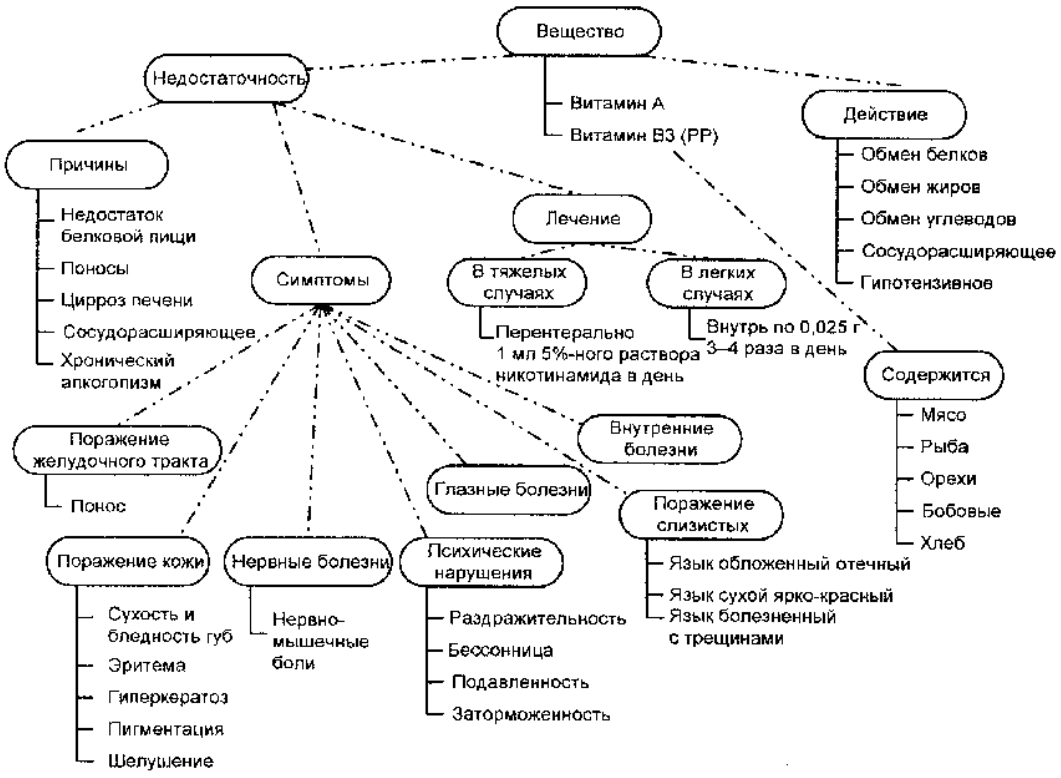

Рис. 2.2. Концептуальная составляющая поля знаний

Признак 1	Признак 2	Признак 3	...	Диагноз	Лечение
Поражение кожи	**Поражение глаз**	**Нервные расстройства**	...	**Недостаток вещества**	**Продукты**
Сухость губ или шелушение	Ослабление зрения		...	Витамин А	Сливочное масло или морковь
		Бессонница	...	Витамин В3	Мясо или рыба
		Раздражительность	...	Витамин В3	Мясо или рыба
...			...		

Рис. 2.3. Функциональная составляющая поля знаний

В последние годы концептуальную структуру называют *онтологией* предметной области [Gruber, 1993], она включает упорядоченные понятия предметной области (ПО) А и моделирует основные функциональные связи R_A или отношения между понятиями, образующими *Sk.* Помимо онтологии понимание задачи отражает модель или стратегия принятия решения *Sf* в выбранной ПО. Таким образом, S_f образует стратегическую составляющую *M,* часто она имеет форму простой таблицы решений, как на рис. 2.4.

Схему, отображающую отношения между реальной действительностью и полем знаний, можно представить так, как показано на рис. 2.4.

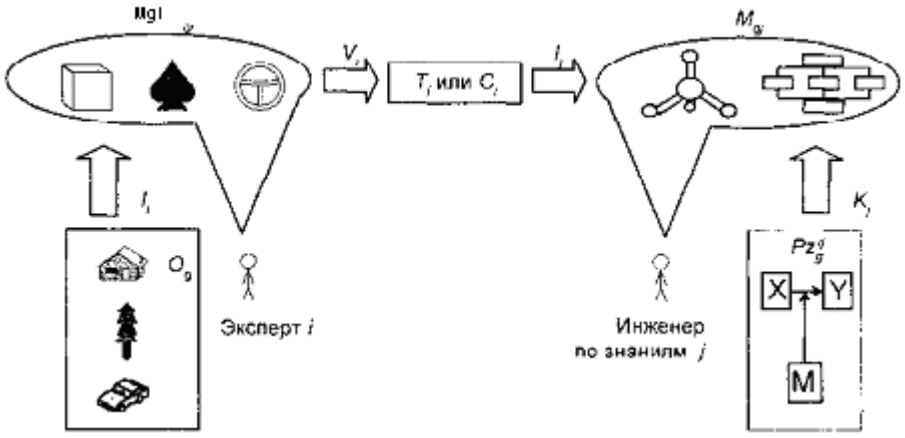

Рис. 2.4. "Испорченный телефон" при формировании поля знаний

Как следует из рисунка, поле Pz_g^{ij} — это результат, полученный "после 4-й трансляции" (если говорить на языке информатики).

1-я трансляция (*Ii*) — это восприятие и интерпретация действительности *O* предметной области *g i*-м экспертом. В результате *Ii* в памяти эксперта образуется модель *Mgi* как семантическая репрезентация действительности и его личного опыта по работе с ней.

2-я трансляция *(Vi,}* — это вербализация опыта *i*-го эксперта, когда он пытается объяснить свои рассуждения S_i и передать свои знания *Zi,* инженеру по знаниям. В результате *Vi,* образуется либо текст T*i,* либо речевое сообщение *Ci*

3-я трансляция *(Ij)* — это восприятие и интерпретация сообщений *Ti* или *Ci j*-м инженером по знаниям. В результате в памяти инженера по знаниям образуется модель мира *Mgj.*

4-я трансляция *(Kj)* — это кодирование и вербализация модели *Mgj* в форме поля знаний Pz_g^{ij}.

Более всего эта схема напоминает детскую игру в "испорченный телефон"; перед инженером по знаниям стоит труднейшая задача — добиться максимального соответствия M_{gi} и Pz_g^{ij} К сожалению, Pz_g не является отражением действительности O_g, т. к. знания — вещь сугубо авторизованная, субъективная. Так следовало бы на каждой ЭС ставить четкий ярлык *i — j,* т. е. "база знаний эксперта *i* в понимании инженера по знаниям *j".* Стоит заменить, например, инженера по знаниям Петрова на Сидорова, и получится совсем другая картина.

Приведем пример влияния субъективных взглядов эксперта на M_{gi} и V_i. Реальность *(O_g)*: два человека прибегают на вокзал за 2 минуты до отхода поезда. В кассы — очередь. В автоматических кассах свободно, но ни у того, ни у другого нет мелочи. Следующий поезд через 40 минут. Оба опаздывают на важную встречу.

- **Интерпретация 1-го эксперта *(I1):*** нельзя приходить на вокзал менее чем за 10 минут.
- **Интерпретация 2-го эксперта (I2):** надо всегда иметь мелочь в кармане.
- **Вербализация 1-го эксперта (V1):** опоздал к нужному поезду, т. к. не рассчитал время.
- **Вербализация 2-го эксперта *(V2):*** опоздал, т. к. на вокзале неразбериха, в кассах толпа.

Последующие трансляции еще больше будут искажать и видоизменять модель, но теперь уже с учетом субъективного восприятия инженеров по знаниям.

Таким образом, если считать поле знаний смысловой (семантической) моделью предметной области, то эта модель дважды субъективна. И если модель *Mgi* (см. рис. 2.4) — это усеченное отображение O_g, то само P_Z — лишь отблеск *Mgi* через призму *Vi* и *Mgj*.

2.1.2. "Пирамида" знаний

Иерархичность понятийной структуры сознания подчеркивается в работах многих психологов [Брунер, 1971; Веккер, 1976]. Поле знаний можно стратифицировать, т. е. рассматривать на различных уровнях абстракции понятий. В "пирамиде знаний" каждый следующий уровень служит для восхождения на новую ступень обобщения и углубления знаний в предметной области. Таким образом, возможно наличие нескольких уровней понятийной структуры S_k

Представляется целесообразным связать это с глубиной профессионального опыта (например, как в системе АВТАНТЕСТ [Гаврилова, Червинская,1992]) или с уровнем иерархии в структурной лестнице организации (рис. 2.5).

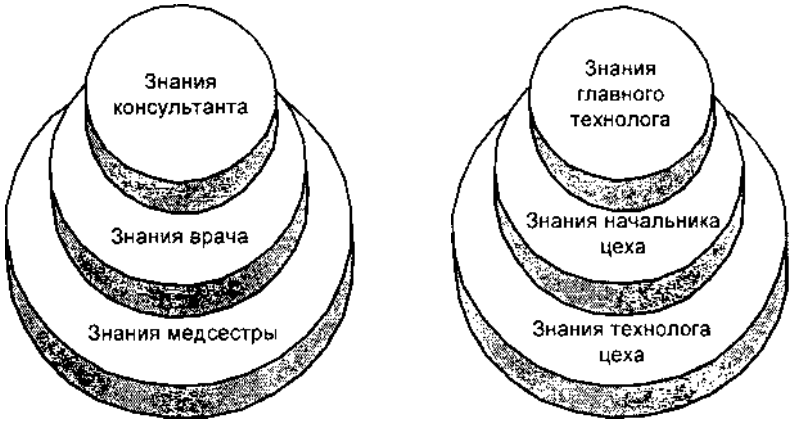

Рис. 2.5. Пирамиды знаний

Естественно, что и стратегии принятия решений, т. е. функциональные структуры S_f на различных уровнях, будут существенно отличаться.

2.2. Стратегии получения знаний

При формировании поля знаний ключевым вопросом является сам процесс получения знаний, когда происходит перенос компетентности экспертов на инженеров по знаниям. Для названия этого процесса в литературе по ЭС получили распространение несколько терминов: приобретение, добыча, извлечение, получение, выявление, формирование знаний. В англоязычной специальной литературе в основном используются два термина: acquisition (приобретение) и elicitation (выявление, извлечение, установление).

Термин *"приобретение" трактуется* либо очень широко — тогда он включает весь процесс передачи знаний от эксперта к базе знаний ЭС, либо уже как способ автоматизированного построения базы знаний посредством диалога эксперта и специальной программы (при этом структура поля знаний заранее закладывается в программу). В обоих случаях термин "приобретение" не касается самого таинства экстрагирования структуры знаний из потока информации о предметной области. Этот процесс описывается понятием *"извлечение "*.

Авторы склонны использовать этот термин как более емкий и более точно выражающий смысл процедуры переноса компетентности эксперта через инженера по знаниям в базу знаний ЭС.

Определение 2.3

***Извлечение знаний* (knowledge elicitation) — это процесс взаимодействия аналитика с источником знаний, в результате которого становится явным процесс рассуждений специалиста при принятии решения и структура его представлений о предметной области.**

Во все времена большинство разработчиков ЭС отмечало, что процесс *извлечения знаний* остается самым "узким" местом при построении промышленных ЭС. При этом им приходится практически самостоятельно разрабатывать методы извлечения, сталкиваясь со следующими трудностями [Gaines, 1989]:

- организационные неувязки;
- неудачный метод извлечения, не совпадающий со структурой знаний в данной области;

- неадекватная модель (язык) для представления знаний.

Можно добавить к этому [Гаврилова, Червинская, 1992]:

- неумение наладить контакт с экспертом;
- терминологический разнобой;
- отсутствие целостной системы знаний в результате извлечения только "фрагментов";
- упрощение "картины мира" эксперта и др.

Процесс извлечения знаний — это длительная и трудоемкая процедура, в которой инженеру по знаниям, вооруженному специальными знаниями по когнитивной психологии, системному анализу, математической логике и пр., необходимо воссоздать модель предметной области, которой пользуются эксперты для принятия решения. Часто начинающие разработчики ЭС, желая упростить эту процедуру, пытаются подменить инженера по знаниям самим экспертом. По многим причинам это нежелательно.

Во-первых, большая часть знаний эксперта — это результат многочисленных наслоений, ступеней опыта. И часто зная, что из A следует B, эксперт не отдает себе отчета, что цепочка его рассуждений была гораздо длиннее, например A -> D -> C -> B, или A -> Q -> R -> B.

Во-вторых, как было известно еще Платону, мышление диалогично. И поэтому диалог инженера по знаниям и эксперта — наиболее естественная форма изучения лабиринтов памяти эксперта, в которых хранятся знания, частью носящие невербальный характер, т. е. выраженные не в форме слов, а в форме наглядных образов, например. И именно в процессе объяснения инженеру по знаниям эксперт на эти размытые ассоциативные образы надевает четкие словесные ярлыки, т. е. вербализует знания.

В-третьих, эксперту труднее создать модель предметной области вследствие глубины и объема информации, которой он владеет. Еще в ситуационном управлении [Поспелов, 1986] было выявлено, что объекты реального мира связаны более чем 200 типами отношений (временные, пространственные, причинно-следственные, типа "часть-целое" и др.). Эти отношения и связи предметной области образуют сложную систему, из которой выделить "скелет" или главную структуру иногда доступнее аналитику, владеющему к тому же системной методологией.

Термин "приобретение" в рамках данной книги оставлен за автоматизированными системами прямого общения с экспертом. Они действительно непосредственно приобретают уже готовые фрагменты знаний в соответствии со структурами, заложенными разработчиками систем. Большинство этих инструментальных средств специально ориентировано на конкретные ЭС с жестко обозначенной предметной областью и моделью представления знаний, т. е. не являются универсальными.

Определение 2.4

Приобретение знаний (knowledge acquisition) — процесс заполнения базы знаний экспертом с использованием специализированных программных средств.

Например, система TEIRESIAS [Davis, 1982], ставшая прародительницей всех инструментариев для приобретения знаний, предназначена для пополнения базы знаний системы **MYCIN** или ее дочерних ветвей, построенных на "оболочке" EMYCIN [Shortliffe et al, 1979] в области медицинской диагностики с использованием продукционной модели представления знаний.

Термин формирование знаний (machine learning) традиционно закрепился за чрезвычайно перспективной и активно развивающейся областью инженерии знаний, которая занимается разработкой моделей, методов и алгоритмов обучения. Она включает индуктивные модели формирования знаний и автоматического порождения гипотез, например ДСМ-метод [Финн, 2000], на основе обучающих выборок, обучение по аналогии и другие методы. Эти модели позволяют выявить причинно-следственные эмпирические зависимости в базах данных с неполной информацией, содержащих структурированные числовые и символьные объекты (часто в условиях неполноты информации).

Определение 2.5

***Формирование знаний* (machine learning) — процесс анализа данных и выявления скрытых закономерностей с использованием специального математического аппарата и программных средств.**

Традиционно к задачам формирования знаний или машинного обучения относятся задачи прогнозирования, идентификации (синтеза) функций, расшифровки языков, индуктивного вывода и синтеза с дополнительной информацией [Епифанов, 1984]. В широком смысле к обучению по примерам можно отнести и методы обучения распознаванию образов [Аткинсон, 1989; Schwartz, 1988].

Для того чтобы эти методы стали элементами технологии интеллектуальных систем, необходимо решить ряд задач [Осипов, 1997]:

- обеспечить механизм сопряжения независимо созданных баз данных, имеющих различные схемы, с базами знаний интеллектуальных систем;
- установить соответствие между набором полей базы данных и множеством элементов декларативного компонента базы знаний;
- выполнить преобразование результата работы алгоритма обучения в способ представления, поддерживаемый программными средствами интеллектуальной системы.

Помимо перечисленных существуют также и другие стратегии получения знаний, например, в случае обучения на примерах (case-based reasoning), когда источник знаний — это множество примеров предметной области [Осипов, 1997; Попов, Фоминых, Кисель, 1996]. Обучение на основе примеров (прецедентов) включает настройку алгоритма распознавания на задачу посредством предъявления примеров, классификация которых известна.

Обучение на примерах тесно связано с машинным обучением. Различие заключается в том, что результат обучения в рассматриваемом здесь случае должен быть интерпретирован в некоторой модели, в которой, возможно, уже содержатся факты и закономерности предметной области, и преобразован в способ представления, который допускает использование результата обучения в базе знаний, для моделирования рассуждений, для работы механизма объяснения и т. д., т. е. делает результат обучения элементом соответствующей технологии.

Например, в системе INDUCE [Коов и др., 1988] порождается непротиворечивое описание некоторого класса объектов по множествам примеров и контрпримеров данного класса. В качестве языка представления используется язык переменно-значной логики первого порядка (вариант языка многозначной логики первого порядка).

В последнее время широкое распространение получили термины data mining и knowledge discovery, означающие, по сути, тот же процесс формирования знаний и поиск закономерностей, осуществляемый на больших выборках данных, обычно находящихся в *хранилищах данных* (data warehouse).

Таким образом, можно выделить три основных стратегии проведения стадии получения знаний при разработке ЭС (рис. 2.6):

- с использованием ЭВМ при наличии подходящего программного инструментария — *приобретение знаний',*
- с использованием программ обучения при наличии репрезентативной (т. е. достаточно представительной) выборки примеров принятия решений в предметной области и соответствующих пакетов прикладных программ —- *формирование знаний;*
- без использования вычислительной техники путем непосредственного контакта инженера по знаниям и источника знаний (будь то эксперт, специальная литература или другие источники) — *извлечение знаний.*

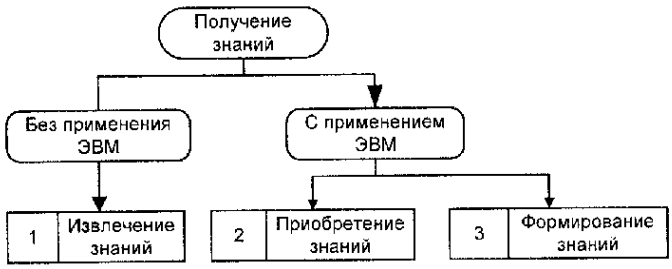

Рис. 2.6. Стратегии получения знаний

Далее в этой главе подробно будут рассматриваться процессы извлечения знаний, т. к. на современном этапе разработки ЭС эти стратегии являются наиболее эффективными и перспективными. Формирование знаний, тяготеющее в большей степени к области machine learning, т. е. индуктивному обучению, основываясь на хорошо исследованном аппарате распознавания образов [Гаек, Гавранек, 1983] и обнаружения сходства объектов [Гусакова, Финн, 1987], выходит за рамки данной книги. Также за рамками книги остались вопросы приобретения знаний [Осипов, 1997] и формирования знаний из данных (data mining, knowledge discovery) и др.

2.3. Теоретико-методические аспекты извлечения и структурирования знаний

Рассмотрим подробнее ключевую проблему основной стратегии получения знаний — непосредственного извлечения знаний "из" памяти эксперта. Можно выделить три основных аспекта этого процесса (рис. 2.7).

- психологический;
- лингвистический;
- гносеологический.

2.3.1. Психологический аспект

Ведущим аспектом извлечения знаний является психологический, поскольку он определяет успешность и эффективность взаимодействия инженера по знаниям (аналитика) с основным источником знаний — экспертом-профессионалом. Психологический аспект выделяется еще и потому, что извлечение знаний происходит чаще всего в процессе непосредственного общения

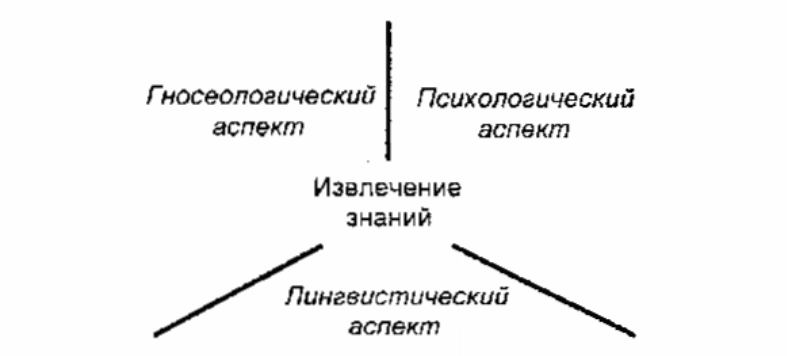

Рис. 2.7. Теоретико-методические аспекты инженерии знаний

разработчиков системы и экспертов, и психологические особенности могут полностью свести на нет высокий интеллектуальный потенциал разработки.

К сожалению, все члены коллектива разработчиков зачастую имеют техническое образование и не владеют даже зачатками профессиональных приемов общения. Общение, или коммуникация (от лат. *communication* — связь) — это междисциплинарное понятие, обозначающее все формы непосредственных контактов между людьми — от дружеских до деловых.

Существует несколько десятков теорий общения, и единственное, в чем сходятся все авторы, — это сложность и многоплановость процедуры общения. Подчеркивается, что общение — не просто однонаправленный процесс передачи сообщений и но двухтактный обмен порциями сведений, а нерасчлененный процесс циркуляции информации, т. е. совместный поиск истины [Каган, 1988] (рис. 2.8).

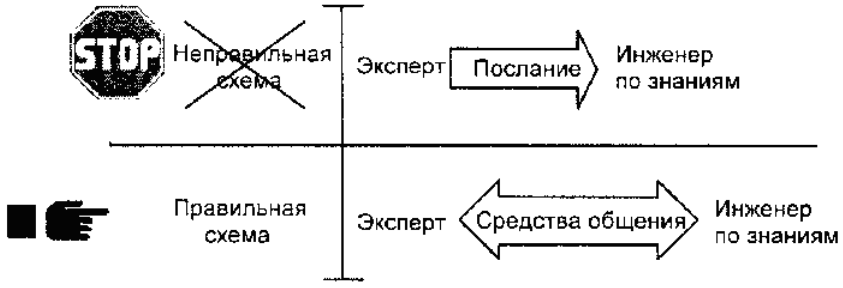

Рис. 2.8.. Структура процесса общения.

Итак, общение есть процесс выработки новой информации, общей для общающихся людей и рождающей их Общность, И хотя общение первый вид деятельности, которым овладевает Человек в своем развитии, 110-настоящему владеют Культурой и наукой общения единицы.

Выделяют четыре основных уровня общения [Сагатовский, 1980]:

- *уровень манипулирования*, котла один субъект рассматривает другого как средство или помеху по отношению к проекту своей деятельности;
- *уровень «рефлексивной игры*», когда в процессе своей деятельности человек учитывает «контрпроект» другого субъекта, но не признает за ним самоценность и стремится к «выигрышу», к реализации своего проекта;
- *уровень правового общения*, когда субъекты признают право на существование проектов деятельности друг друга и пытаются согласовать их хотя бы внешне;
- *уровень нравственного общения*, когда субъекты внутренне принимают общий проект взаимной деятельности.

Стремление и умение общаться на высшем, четвертом, уровне может характеризовать степень профессионализма инженера по знаниям.

Известно, что потери информации при разговорном общении велики [Мицич, 1987] (рис. 2.9).

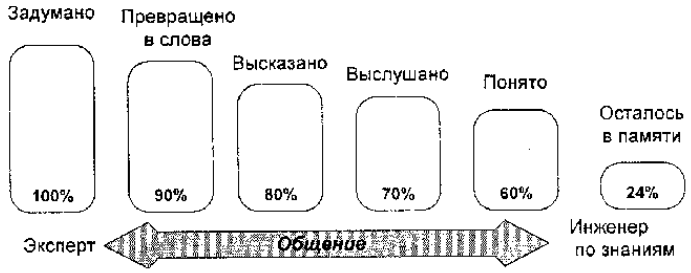

Рис. 2.9. Потери информации при разговорном общении

Рис. 2.9. Потери информации при разговорном общении.

В связи с этим рассмотрим проблему увеличения информативности общения аналитика и эксперта за счет использования психологических знаний.

Можно выделить такие структурные компоненты модели общения при извлечении знаний:

- участники общения (партнеры);
- средства общения (процедура);
- предмет общения (знания).

В соответствии с этой структурой выделим три «слоя» психологических проблем возникающих при извлечении знаний (рис. 2.10):

- контактный;
- процедурный;
- когнитивный.

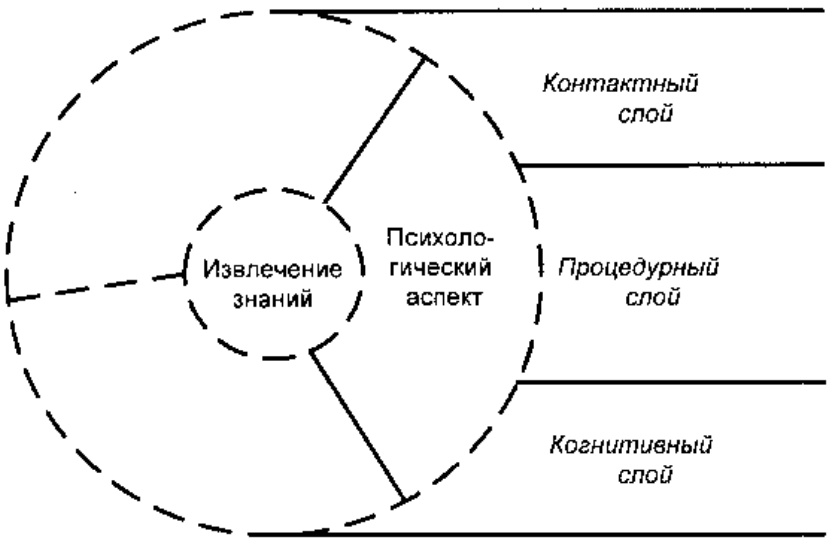

Рис. 2.10. Психологический аспект извлечения знаний

Контактный слой

Разработка проблематики контактного слоя позволила выявить следующие параметры партнеров, влияющие на результаты процедуры извлечения знаний:

- пол и возраст;
- личность;
- темперамент;
- мотивация и др.

Пол и возраст

Значения параметров пола и возраста хотя и влияют на эффективность контакта, но не являются критическими. В литературе [Иванов, 1986] отмечается, что хорошие результаты дают гетерогенные пары (мужчина/женщина), т. к. повышается мотивация.

Что касается возраста, то рабочая формула имеет вид

$$20 > (B_э - Ba) > 5,$$

где *Вэ* — возраст эксперта; Ва — возраст аналитика.

Личность

Под личностью обычно понимается устойчивая система психологических черт, характеризующая индивидуальность человека. Рекомендуемые компоненты личностного портрета аналитика исследованы в работе [Гаврилова,1990] и дополнены качествами из руководства для журналистов в [schouk-smith G., 1978]: *доброжелательность, аналитичность, хорошая память, внимание, наблюдательность, воображение, впечатлительность, большая собранность, настойчивость, общительность, находчивость.*

Подробную информацию *см. в разд. 1. 7.*

.

Темперамент

Со времен Гачена и Гиппократа, выделивших четыре классических типа темперамента, вошли в научную терминологию понятия — *холерик, сангвиник, меланхолик, флегматик.*

Известно, что флегматики и меланхолики медленнее усваивают информацию [Лунева, Хорошилова, 1987]. И для обеспечения психологического контакта с ними не следует задавать беседе слишком быстрый темп, торопить их с ответом. Зато они гораздо лучше усваивают новое, в отличие от холериков, для которых свойственно поверхностное усваивание информации. Последних следует специально наводить на размышление и рефлексию. У меланхоликов часто занижена самооценка, они застенчивы и в беседе их надо подбадривать. Таким образом, наиболее успешными в рамках контактного слоя являются сангвиники и холерики.

Мотивация

На эффективность коллективного решения задач влияет также и мотивация, т. е. стремление к успеху. Инженер по знаниям в зависимости от условий разработки должен изыскивать разнообразные стимулы для экспертов (включая, разумеется, и материальные). Эксперт передает аналитику один из самых дорогих в мире продуктов — знания. И если одни люди делятся опытом добровольно и с удовольствием, то другие весьма неохотно приоткрывают свои профессиональные тайны. Иногда полезно, оказывается, возбудить в эксперте дух соперничества, конкуренции (не нарушая, естественно, обстановки кооперативности в коллективе).

Процедурный слой

Параметры процедурного слоя описывают непосредственно процесс проведения процедуры извлечения знаний. Фактически это профессиональные параметры:

- ситуация общения (место, время и продолжительность);
- оборудование (вспомогательные средства, освещенность, мебель);
- профессиональные приемы (темп, стиль, учет невербальных компонентов общения, методы и др.).

Инженер по знаниям, успешно овладевший наукой установления атмосферы доверия и взаимопонимания с экспертом (контактный слой), должен еще суметь воспользоваться благоприятным воздействием этой атмосферы. Проблема процедурного слоя касается проведения самой процедуры извлечения знаний. Здесь мало проницательности и обаяния, полезных для решения проблемы контакта, тут необходимы профессиональные знания.

Остановимся на общих закономерностях проведения процедуры.

Ситуация общения

Она определяется следующими компонентами:

- место проведения сеансов;
- продолжительность и время проведения сеансов.

Место. Беседу с экспертом лучше всего проводить в небольшом помещении наедине, поскольку посторонние люди нарушают доверительность беседы и могут породить эффект "фасада". Рабочее место эксперта является не самым оптимальным вариантом, т. к. его могут отвлекать телефонные звонки, сотрудники и пр. Атмосфера замкнутого пространства и уединенности положительно влияют на эффективность.

Американский психолог И. Атватер считает, что для делового общения наиболее благоприятная дистанция от 1,2 до 3 м [Schouksmith, 1978]. Минимальным "комфортным" расстоянием можно считать 0,7—0,8 м.

Продолжительность и время. Реконструкция собственных рассуждений — трудоемкий процесс, и поэтому длительность одного сеанса обычно не превышает 1,5—2 часа.

Эти два часа лучше выбрать в первой половине дня, например с 10 до 12 часов, если эксперт типа "жаворонок", и в районе 16—17 часов, если он "сова". Эти два пика активности (11 и 16 часов) установлены исследователями 24-часового ритмоцикла человеческой активности.

Известно, что взаимная утомляемость партнеров при беседе наступает обычно через 20—25 минут [Ноэль, 1978], поэтому в сеансе нужны паузы.

Оборудование

Обычно включает:

- вспомогательные средства;
- освещенность и мебель.

Вспомогательные средства представляют собой средства для увеличения эффективности самого процесса извлечения знаний и средства для протоколирования результатов.

К средствам для увеличения эффективности процесса извлечения знаний, прежде всего, относится наглядный материал. Независимо от метода извлечения *(см. разд. 2.4),* выбранного в конкретной ситуации, его реализация возможна разными способами.

Например, широко известно, что людей, занимающихся интеллектуальной деятельностью, можно отнести к художественному либо мыслительному типу. Термины тут условны и не имеют отношения к той деятельности, которую традиционно называют художественной или мыслительной. Важно, что, определив тип эксперта, инженер по знаниям может плодотворнее использовать любой из методов извлечения, зная, что люди художественного типа легче воспринимают зрительную информацию в форме рисунков, графиков, диаграмм, т. к. эта информация воспринимается через первую сигнальную систему. Напротив, эксперты мыслительного типа лучше понимают язык формул и текстовую информацию. При этом учитывается факт, что большую часть информации человек получает от зрения. Совет пользоваться активнее наглядным материалом из [Хейес-Рот, Уотермена, Ленат, 1987] можно считать универсальным. Такие методы, как свободный диалог и игры, предоставляют богатые возможности использовать слайды, чертежи, рисунки.

Для протоколирования результатов в настоящее время используются следующие способы:

- запись на бумагу непосредственно по ходу беседы (недостатки — это часто мешает беседе, кроме того, трудно успеть записать все, даже при наличии навыков стенографии);
- магнитофонная запись (диктофон), помогающая аналитику проанализировать весь ход сеанса и свои ошибки (недостаток — может сковывать эксперта);
- запоминание с последующей записью после беседы (недостаток — годится только для аналитиков с блестящей памятью).

Наиболее распространенным способом на сегодня является первый. При этом наибольшая опасность тут — потеря знаний, поскольку любая запись ответов — это уже интерпретация, т. е. привнесение субъективного понимания предмета.

Естественным является требование отключать мобильный телефон во время сеанса (хотя бы со стороны аналитика).

Освещенность и мебель. Значения параметров *освещенности* и *мебели* также влияют на эффективность процедуры извлечения знаний. Посадите эксперта на неудобную табуретку, направьте ему свет в глаза и посмотрите, сколько минут он выдержит...

Профессиональные приемы аналитика

Они включают, в частности:

- темп и стиль;
- учет невербальных компонентов общения;
- методы.

Темп и стиль. Учет индивидуального темпа и стиля эксперта позволяет аналитику снизить напряженность процедуры извлечения знаний. Типичной ошибкой является навязывание собственного темпа и стиля. Необходимо умение перенимать темп и стиль общения эксперта (эффект зеркала).

На успешность также влияет длина фраз, которые произносит инженер по знаниям. Этот факт был установлен американскими учеными — лингвистом Ингве и психологом Миллером при проведении исследования причин низкого усвоения команд на Военно-морском флоте США [Gammack, Young, J985].

Причина была в длине команд. Оказалось, что человек лучше всего воспринимает предложения глубиной (или длиной) 7+2(—2) слова. Это число *(7+2)* получило название "число Ингве—Миллера". Можно считать его мерой "разговорности" речи. Опытные лекторы используют в лекции в основном короткие фразы, уменьшая потерю информации с 20—30% (у плохих лекторов) до 3-4% [Горелов, 1987].

Учет невербальных компонентов общения. Большая часть информации поступает к инженеру по знаниям в форме предложений на естественном языке. Однако внешняя речь эксперта есть воспроизведение его внутренней речи (мышления), которая гораздо богаче и многообразнее. При этом для передачи этой внутренней речи эксперт использует и невербальные средства, такие как: интонация, мимика, жесты. Опытный инженер по знаниям старается записывать по возможности в протоколы (в форме ремарок) эту дополнительную интонацию.

В целом, невербальный компонент стиля общения важен и для проблем контактного слоя при установлении контакта, когда по отдельным жестам и выражению лица эксперта инженер по знаниям может установить границу возможной "дружественности" общения.

Методы. Конкретные методы подробно рассмотрены в следующем разделе, исходя из позиции, что метод должен подходить к эксперту как "ключ к замку".

Когнитивный слой

Когнитивные (от англ, *cognition* — познание) науки исследуют познавательные процессы человека с позиций возможности их моделирования (психология, нейрофизиология, эргономика, инженерия знаний). Наименее исследованы на сегодняшний день проблемы когнитивного слоя, связанные с изучением семантического пространства памяти эксперта и реконструкцией его понятийной структуры и модели рассуждений.

Основными факторами, влияющими на когнитивную адекватность поля знаний, будут:

- когнитивный стиль;
- семантическая репрезентативность поля знаний и концептуальной модели.

Когнитивный стиль

Под *когнитивным стилем* человека понимается совокупность критериев предпочтения при решении задач и познании мира, специфическая для каждого человека. Когнитивный стиль определяет не столько эффективность деятельности, сколько способ достижения результата [Алахвердов, 1986]. Это способ познания, который позволяет людям с разными способностями добиваться одинаковых результатов в деятельности. Это система средств и индивидуальных приемов, к которым прибегает человек для организации своей деятельности.

Инженеру по знаниям полезно изучить и прогнозировать свой когнитивный стиль, а также стиль эксперта. Особенно важны такие характеристики когнитивного стиля, как:

- полезависимость — поленезависимость;
- импульсивность — рефлективность (рефлексивность);
- ригидность — гибкость;
- диапазон когнитивной эквивалентности.

Поленезависимость. Это свойство позволяет человеку акцентировать внимание лишь на тех аспектах проблемы, которые необходимы для решения конкретной задачи, и умение отбрасывать все лишнее, т. е. не зависеть от фона или окружающего задачу шумового поля. Эта характеристика коррелирует с такими чертами личности, как невербальный интеллект, аналитичность мышления, способность к пониманию сути. Очевидно, что помимо того, что самому аналитику необходимо иметь высокое значение параметра, поленезависимый эксперт — это тоже желательный фактор. Однако приходится учитывать, что больше нуждаются в общении полезависимые люди, а потому они и более контактны [Орехов, 1985].

Особенно полезны для общения гетерогенные (смешанные) пары, например "полезависимый — поленезависимый" [Иванов, 1986]. В литературе описаны различные эксперименты, моделирующие общение, требующее понимания и совместной деятельности. Наиболее успешным в понимании оказались поленезависимые испытуемые (92% успеха), для сравнения полезависимые давали 56% успеха [Кулюткин, Сухобская, 1971].

Для совместной профессиональной деятельности важна также гибкость когнитивной организации, которая связана с поленезависимостью. Итак, большую способность к адекватному пониманию партнера демонстрируют субъекты с высокой психологической дифференциацией, т. е. поленезависимостью.

Поленезависимость является одной из характерных профессиональных черт когнитивного стиля наиболее квалифицированных инженеров по знаниям. По некоторым результатам [Алахвердов, 1986] мужчины более поленезависимы, чем женщины.

Импульсивность. Под *импульсивностью* понимается быстрое принятие решения (часто без его достаточного обоснования), а под *рефлексивностью* — склонность к рассудительности. Рефлексивность по экспериментальным данным коррелирует со способностью к формированию понятий и продуктивностью стратегий решения логических задач [Кулюткин, Сухобская, 1971]. Таким образом, и инженеру по знаниям, и эксперту желательно быть рефлексивным, хотя собственный стиль изменяется лишь частично и с большим напряжением.

Ригидность. *"Ригидность — гибкость"* характеризует способность человека к изменению установок и точек зрения в соответствии с изменяющейся ситуацией. Ригидные люди не склонны менять свои представления и структуру восприятия, напротив, гибкие легко приспосабливаются к новой обстановке. Очевидно, что если эксперт еще может себе позволить ригидность (что характерно для долго работающих над одной проблемой специалистов, особенно старшего возраста), то для инженера по знаниям эта характеристика когнитивного стиля явно противопоказана. Увеличение ригидности с возрастом отмечается многими психологами [Кулюткин, Сухобская, 1971; Орехов, 1985].

Диапазон когнитивной эквивалентности. *Когнитивная эквивалентность* характеризует способность человека к различению понятий и разбиению их на классы и подклассы. Чем уже диапазон когнитивной эквивалентности, тем более тонкую классификацию способен провести индивид, тем большее количество признаков понятий он может выделить. Обычно у женщин диапазон когнитивной эквивалентности уже, чем более широкий диапазон у мужчин.

Семантическая репрезентативность

Эта проблема подразумевает подход, исключающий традиционное навязывание эксперту некой модели представлений (например, продукционной или фреймовой), и, напротив, заставляет инженера по знаниям последовательно воссоздавать модель мира эксперта, используя как неформальные методы, так и математический аппарат, например многомерное шкалирование [Петренко, 1988; Воинов, Гаврилова, 1996]. Проблема семантической репрезентативности ориентирована на достижение когнитивной адекватности поля знаний и концептуальной модели. В настоящий момент она может быть сформулирована как проблема "испорченного телефона" [Гаврилова, Червинская, 1992] *(см. разд. 2.1,* рис. 2.4) — возможные трансформации и потери в цепи передачи информации.

2.3.2. Лингвистический аспект

Лингвистический аспект касается исследований языковых проблем, т. к. язык — это основное средство общения в процессе извлечения знаний.

Сразу же следует оговорить, что поскольку книга посвящена проблемам разработки ЭС, то область разработки естественно-языковых интерфейсов и весь спектр проблем, связанных с ней — лексических, синтаксических, семантических, прагматических и т. д. [Мальковский, 1985; Попов, 1982], не рассматривается.

В инженерии знаний можно выделить три слоя лингвистических проблем (рис. 2.11):

- "общий код";
- понятийная структура;
- словарь пользователя.

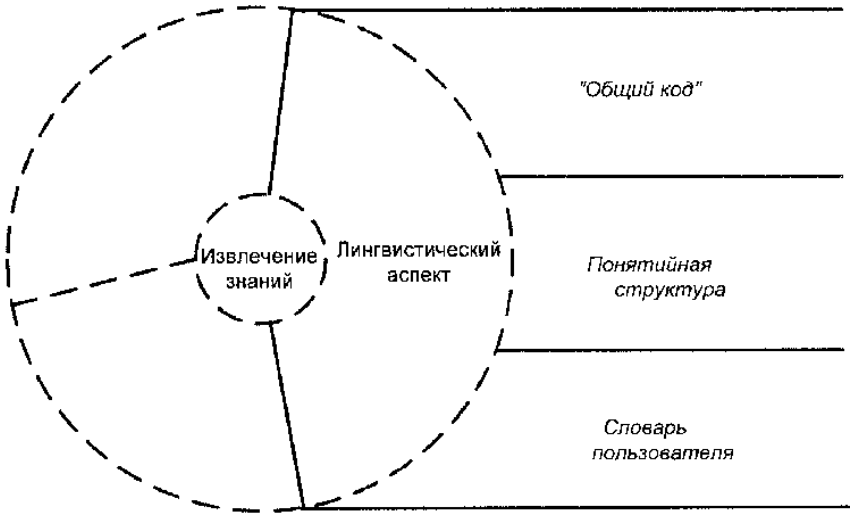

Рис. 2.11. Лингвистический аспект извлечения знаний

"Общий код"

"Общий код" решает проблему языковых ножниц между профессиональной терминологией эксперта и обыденной литературной речью инженера по знаниям и включает следующие компоненты:

- общенаучная терминология;
- специальные понятия из профессиональной литературы;
- элементы бытового языка;
- неологизмы, сформированные за время совместной работы;
- профессиональный жаргон и др.

Детализируя схему общения (см. рис. 2.8), можно представить средства общения как два потока [Горелов, 1987], состоящих из компонентов V_1 и V_2 — соответственно языки, на которых говорят аналитик и эксперт (V_1' и V_2' — невербальные компоненты). Различия языков V_1 и V_2 и обуславливает "языковый барьер" или "языковые ножницы" в общении инженера по знаниям и эксперта.

Эти два языка являются отражением "внутренней речи" эксперта и аналитика, поскольку большинство психологов и лингвистов считают, что язык — это основное средство мышления, наряду с другими знаковыми системами "внутреннего пользования" (универсальный семантический код — УСК [Мартынов. 1977], языки "смысла" [Мельчук, 1974], концептуальные языки [Шенк, 1980] и др.).

Язык аналитика V_1 состоит из трех компонентов:

- общенаучной терминологии из его "теоретического багажа";
- терминов предметной области, которые он почерпнул из специальной литературы в период подготовки;
- бытового разговорного языка, которым пользуется аналитик.

Язык эксперта V_2 включает:

- общенаучную терминологию;

- специальную терминологию, принятую в предметной области;
- бытовой язык;
- неологизмы, созданные экспертом за время работы, т. е. его профессиональный жаргон.

Если считать, что бытовой и общенаучный язык у двух участников общения примерно совпадает (хотя реально объем второго компонента у эксперта существенно больше), то некоторый общий язык или код, который необходимо выработать партнерам для успешного взаимодействия, будет складываться из потоков, представленных на рис. 2.12.

В дальнейшем этот общий код преобразуется в некоторую понятийную (семантическую) сеть, которая является прообразом поля знаний предметной области.

Выработка общего кода начинается с выписыванием аналитиком всех терминов, употребляемых экспертом, и уточнения их смысла. Фактически это составление словаря предметной области. Затем следует группирование терминов и выбор синонимов (слов, означающих одно и то же). Разработка общего кода заканчивается составлением словаря терминов предметной области с предварительной группировкой их по смыслу, т. е. по понятийной близости (это уже первый шаг структурирования знаний).

На этом этапе аналитик должен с большим вниманием отнестись ко всем специальным терминам, пытаясь максимально вникнуть в суть решаемых проблем и терминологию. Освоение аналитиком языка предметной области — первый рубеж на подступах к созданию адекватной базы знаний.

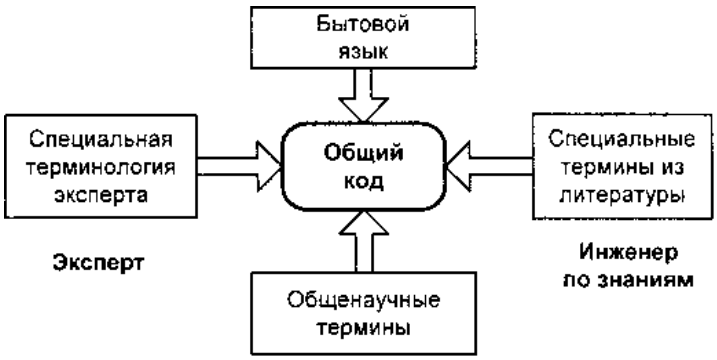

Рис. 2.12. Структура общего кода

Рис. 2.13 дает представление о процессе неоднозначности интерпретации терминов двумя специалистами. В семиотике, науке о знаковых системах, проблема интерпретации является одной из центральных. Интерпретация связывает "знак" и "означаемый предмет". Только в интерпретации знак получает смысл. Так, на рис. 2.13 слова "прибор Х" для эксперта означают некоторую конкретную схему, которая соответствует схеме оригинала прибора, а в голове начинающего аналитика слова "прибор Х" вызывают пустой образ или некоторый черный ящик с ручками

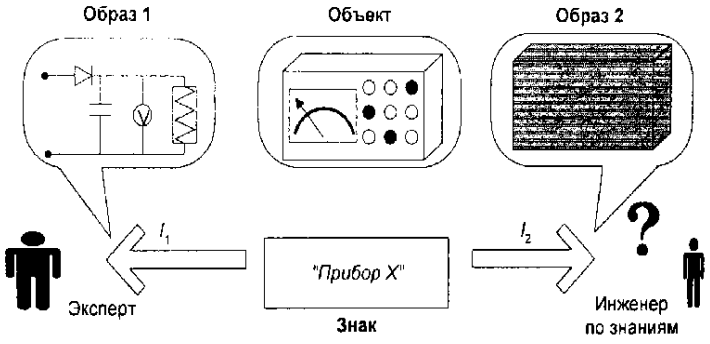

Рис. 2.13. Неоднозначность интерпретации

Внимание к лингвистическому аспекту проблемы извлечения знаний способствует сближению образа 1 с образом 2 и интерпретации *I1* с интерпретацией I2, а слова "прибор X" перейдут в действительно "общий" код.

Понятийная структура

Особенности формирования понятийной структуры обусловлены установленным постулатом когнитивной психологии о взаимосвязи понятий в памяти человека и наличии семантической сети, объединяющей отдельные термины во фрагменты, фрагменты в сценарии и т. д. Построение иерархической сети понятий, так называемой "пирамиды знаний" *(см. разд. 2.1),* — важнейшее звено в проектировании интеллектуальных систем.

Большинство специалистов по искусственному интеллекту и когнитивной психологии считают, что основная особенность естественного интеллекта и памяти в частности — это связанность всех понятий в некоторую сеть. Поэтому для разработки базы знаний и нужен не словарь, а "энциклопедия" [Шенк, Бирнбаум, Мей, 1989], в которой все термины объяснены в словарных статьях со ссылками на другие термины.

Таким образом, лингвистическая работа инженера по знаниям на данном слое проблем заключается в построении таких связанных фрагментов с помощью "сшивания" терминов. Фактически эта работа является подготовкой к этапу концептуализации, где это "шитье" (по Шенку — КОП, концептуальная организация памяти [Шенк, Хантер, 1987]) приобретает некоторый законченный вид.

При тщательной работе аналитика и эксперта в понятийных структурах начинает просматриваться иерархия понятий. Такие структуры имеют важнейшее гносеологическое и дидактическое значение и в последнее время для них используется специальный термин — *онтология* [Gruber, 1989; Гав-рилова, Хорошевский, 2001]. Следует заметить, что эта иерархическая организация хорошо согласуется с теорией универсального предметного кода (УПК) [Горелов, 1987; Жинкин, 1982], согласно которой при мышлении используются не языковые конструкции, а их коды в форме некоторых абстракций, что в общем согласуется с результатами когнитивной психологии [Величковский, 1982].

Онтология или иерархия абстракций — это глобальная схема, которая может быть положена в основу концептуального анализа структуры знаний любой предметной области. Лингвистический эквивалент иерархии — иерархия понятий, которую необходимо построить в понятийной структуре, формируемой инженером по знаниям (рис. 2.14).

Следует подчеркнуть, что работа по составлению словаря и понятийной структуры требует лингвистического "чутья", легкости манипулирования терминами и богатого словарного запаса инженера по знаниям, т. к. зачастую аналитик вынужден самостоятельно разрабатывать словарь признаков. Чем богаче и выразительнее получается общий код, тем полнее база знаний.

Аналитик вынужден все время помнить о трудности передачи образов и представлений в вербальной форме. Полезными тут оказываются свойства многозначности слов естественного языка. Часто инженеру по знаниям приходится подсказывать слова и выражения эксперту, и такие новые лексические конструкции оказываются полезными.

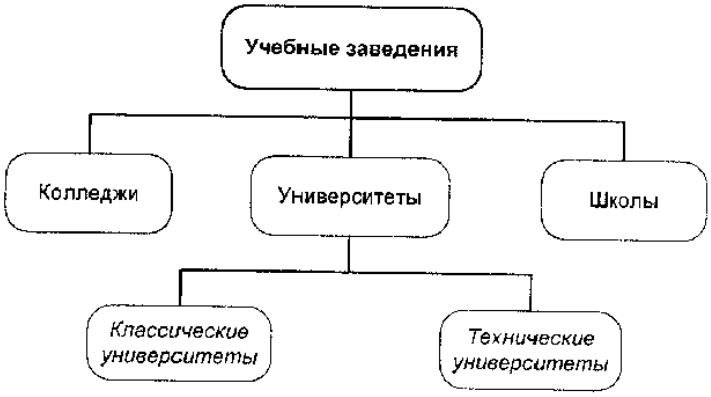

Рис. 2.14. Пример иерархии абстракций

Способность к словесной интерпретации зависит и от пола аналитика. Установлено, что традиционно женщины придают большую значимость невербальным компонентам общения, а в вербальных имеют более обширный алфавит признаков. И вообще, существуют половые различия восприятия не только в бытовой сфере, что очевидно, но и в профессиональной. Следовательно, у эксперта-мужчины и у эксперта-женщины могут существенно отличаться алфавиты для вербализации признаков воспринимаемых объектов.

Словарь пользователя

Формирование общего кода и разработка понятийной структуры направлены на создание адекватной базы знаний. Однако часто профессиональный уровень конечного пользователя не позволяет ему применить специальный язык предметной области в полном объеме.

Неожиданными для начинающих разработчиков являются проблемы формирования отдельного словаря для создания дружественного интерфейса с пользователем ЭС. Необходимы специальные приемы, увеличивающие "прозрачность" и доступность системы. Для разработки пользовательского интерфейса требуется дополнительная доработка словаря общего кода с поправкой на доступность и "прозрачность" системы.

Так, при разработке экспертной системы по психодиагностике АВТАНТЕСТ [Гаврилова, 1984] пришлось разработать два словаря терминов — один для психологов-профессионалов, второй — для неспециалистов (испытуемых). Поскольку результат психодиагностического тестирования всегда интересен испытуемому, ему выдается листинг с психологическим заключением на общелитературном языке без употребления специальных терминов. Интересно, что при внедрении системы использовался в основном этот второй словарь; даже профессиональные психологи предпочитали получать тексты на обыденном языке.

2.3.3. Гносеологический аспект

Гносеологический аспект извлечения знаний объединяет методологические проблемы получения нового научного знания, поскольку при создании БЗ эксперт часто впервые формулирует некоторые закономерности, до того момента составлявшие его личный опыт. Гносеология это раздел философии, связанный с теорией познания, или теорией отражения действительности в сознании человека.

Инженерия знаний как наука, если можно так выразиться, дважды гносеологична — сначала действительность *(О)* отражается в сознании эксперта *(M_1),* а затем деятельность и опыт эксперта интерпретируются сознанием инженера по знаниям (M_2), что служит уже основой для построения третьей интерпретации *(P_2)* — поля знаний экспертной системы (см. рис. 2.4). Процесс познания, в сущности, направлен на создание внутренней репрезентации окружающего мира в сознании человека.

Если описать процессы интерпретации I_2 и I_3 в терминологии, введенной в *гл. 1,* то мы имеем дело с превращением экспертного знания и теоретического (книжного) опыта Z_1 в поле знаний Z_2, которое есть материализация модели мира М2 инженера по знаниям.

В процессе извлечения знаний аналитика в основном интересует компонент Z_1, связанный с неканоническими индивидуальными знаниями экспертов, поскольку предметные области, требующие именно такого типа знаний, считаются наиболее восприимчивыми к внедрению экспертных систем. Эти области обычно называют эмпирическими, т. к. в них накоплен большой объем отдельных эмпирических фактов и наблюдений, в то время как их теоретическое обобщение — вопрос будущего.

Если считать, что инженер по знаниям извлекает только фрагмент Z_1', т. е. часть из системы знаний эксперта Z_1, то его задача, во-первых, стараться, чтобы структура Z_1' соответствовала Z_1 и, во-вторых, чтобы Z_1'как можно более полно отражал Z_1.

Познание часто сопровождается созданием новых понятий и теорий. Иногда эксперт порождает новые знания прямо в ходе беседы с аналитиком. Такая генерация знаний полезна и самому эксперту, который до того момента мог не осознавать ряд соотношений и закономерностей предметной области. Аналитику может помочь тут и инструментарий системной методологии, позволяющий использовать известные принципы логики научных исследований, понятийной иерархии науки. Эта методология заставляет его за частным всегда стремиться увидеть общее, т. е. строить цепочки:

факт -> обобщенный факт -> эмпирический закон -> теоретический закон.

Не всегда удается дойти до последнего звена этой цепочки, но уже само стремление к продвижению бывает чрезвычайно плодотворным. Такой подход полностью согласуется со структурой самого знания, которое имеет два уровня:

- эмпирический (наблюдения, явления);

- теоретический (законы, абстракции, обобщения).

Основными методологическими критериями стройности выявленной системы знаний можно считать [Коршунов, Манталов, 1988]:

- внутреннюю согласованность;
- системность;
- объективность;
- историзм.

Внутренняя согласованность

На первый взгляд критерий внутренней согласованности знания не соответствует реальным характеристикам, описывающим знания. Любое эмпирическое знание *противоречиво и неполно.* Эти характеристики эмпирических знаний подчеркивают его "многоукладность" — столь часто факты не согласуются друг с другом, определения противоречат, критерии диффузны и т. д. Аналитику, знающему особенности эмпирического знания, приходится сглаживать эти "шероховатости" эмпирики. Возможная противоречивость эмпирического знания — естественное следствие из основных законов диалектики, и противоречия эти не всегда должны разрешаться в поле знаний, а напротив, именно противоречия служат чаще всего отправной точкой в рассуждениях экспертов.

Неполнота знания связана с невозможностью полного описания предметной области. Задача аналитика эту неполноту ограничить определенными рамками "полноты", т. е. сузить границы предметной области, либо ввести ряд ограничений и допущений, упрощающих проблему.

Кроме того, эмпирическое знание модально. Модальность знания означает возможность его существования в различных категориях, т. е. в конструкциях существования и долженствования. Таким образом, часть закономерностей возможна, другая обязательна и т. д. Кроме того, приходится различать такие оттенки модальности, как:

- эксперт знает, что...;
- эксперт думает, что...;
- эксперт хочет, чтобы...;
- эксперт считает, что...

Системность

Системно-структурный подход к познанию (восходящий еще к Гегелю) ориентирует аналитика на рассмотрение любой предметной области с позиций закономерностей системного целого и взаимодействия составляющих его частей.

Современный структурализм исходит из многоуровневой иерархической организации любого объекта, т. е. все процессы и явления можно рассматривать как множество более мелких подмножеств (признаков, деталей) и, наоборот, любые объекты можно (и нужно) рассматривать как элементы более высоких классов обобщений. Например, системный взгляд на проблематику структурирования знаний позволяет увидеть его иерархическую организацию. Подробнее об этом *см. долее в настоящей главе.*

Объективность

Процесс познания глубоко субъективен, т. е. он существенно зависит от особенностей самого познающего субъекта. "Факты существуют для одного глаза и отсутствуют для другого" (Виппер). Таким образом, субъективность начинается уже с описания фактов и увеличивается по мере углубления идеализации объектов.

Следовательно, более корректно говорить о глубине понимания, чем об объективности знания. Понимание — это сотворчество, процесс истолкования объекта с точки зрения субъекта. Это сложный и неоднозначный процесс, совершающийся в глубинах человеческого сознания и требующий мобилизации всех интеллектуальных и эмоциональных способностей человека. Все свои усилия аналитик должен сосредоточить на понимании проблемы.

В психологии известен результат [Величковский, Капица, 1987], подтверждающий факт, что люди, быстро и успешно решающие интеллектуальные задачи, большую часть времени тратят на понимание ее, в то время как плохие решатели быстро приступают к поискам решения и чаще всего не могут его найти.

Историзм

Этот критерий связан с развитием. Познание настоящего — есть познание породившего его прошлого. И хотя большинство экспертных систем дают "горизонтальный" срез знаний — без учета времени (в статике), инженер по знаниям должен всегда рассматривать процессы с учетом временных изменений — как связь с прошлым, так и связь с будущим. Например, структура поля знаний и база знаний должны допускать подстройку и коррекцию как в период разработки, так и во время эксплуатации ЭС.

Методология процесса получения нового знания

Методологически деятельность аналитика может быть представлена как некоторая последовательность этапов [Коршунов, Манталов, 1988]:

- Э_1: описание и обобщение фактов;
- Э_2: установление логических и математических связей, дедукция и индукция законов;
- Э_3: построение модели;
- Э_4: объяснение и предсказание явлений.

Э_1: описание и обобщение фактов

Тщательность и полнота ведения протоколов во время процесса извлечения и пунктуальная "домашняя работа" над ними — вот залог продуктивного первого этапа познания и материал для описания и обобщения фактов.

На практике оказывается трудным придерживаться принципов объективности и системности, описанных выше. Чаще всего на этом этапе факты просто собирают и как бы бросают в "общий мешок"; опытный инженер по знаниям часто сразу пытается найти "полочку" или "ящичек" для каждого факта, тем самым подспудно готовясь к этапу концептуализации.

Э_2: установление связей

В памяти эксперта все понятия увязаны и закономерности установлены, хотя часто и неявно, задача инженера — выявить каркас умозаключений эксперта. Реконструируя рассуждения эксперта, инженер по знаниям может опираться на две наиболее популярные теории мышления — логическую и ассоциативную. При этом если логическая теория благодаря горячим поклонникам в лице математиков широко цитируется и всячески эксплуатируется в работах по искусственному интеллекту, то вторая, ассоциативная, гораздо менее известна и популярна, хотя имеет также древние корни. Так, Р. Фейнман в своих "Лекциях по физике" отмечает, что в физике по-прежнему преобладающим является вавилонский, а не греческий метод построения знаний. Известно, что древневосточные математики умели делать сложные вычисления, но формулы их не были логически увязаны. Напротив, греческая математика дедуктивна (например, "Начала" Евклида).

Традиционная логика формирует критерии, которые гарантируют точность, валидность, непротиворечивость общих понятий рассуждений и выводов. Ее основы заложены еще в "Органоне" Аристотеля в IV в. до н. э. Большой вклад в развитие логики внес Джон Стюарт Милль (1806—1873).

Инженер по знаниям и сам использует операции традиционной логики и выделяет их в схеме рассуждений эксперта. Это следующие операции:

- определение;
- сравнение и различение;
- анализ;
- абстрагирование;
- обобщение;
- классификация;
- категоризация;
- образование суждений;
- умозаключение;
- составление силлогизмов и т. д.

Однако красота и стройность логической теории не должны заслонять того, что человек редко мыслит в категориях математической логики [Поспелов, 1989].

Теория ассоциаций представляет мышление как цепочку идей, связанных общими понятиями. Основными операциями такого мышления являются:

- ассоциации, приобретенные на основе различных связей;
- припоминание прошлого опыта;
- пробы и ошибки со случайными успехами;
- привычные ("автоматические") реакции и пр.

Однако эти две теории не исчерпывают всего многообразия психологических школ. Большой интерес для инженерии знаний может представлять гештальтпсихология. Одним из ее основателей является выдающийся немецкий психолог М. Вертгеймер (1880—1943). Под гештальтом (нем. Gestalt) понимается принцип целостности восприятия — как основа мышления. Гештальтпсихологи стараются во всем выделять некий целостный образ или структуру как базис для понимания процессов и явлений окружающего мира. Эта теория близка теории фреймов и объектному подходу и направлена на постижение глубинного знания, которое характеризуется стабильностью и симметрией. При этом важен так называемый "центр ситуации", относительно которого развивается знание о предметной области.

Для инженера по знаниям это означает, что, выявляя различные фрагменты знаний, он не должен забывать о главном, о гештальте фрагмента, который влияет на остальные компоненты и связывает их в некоторую структурную единицу. Гештальтом может быть некий главный принцип, или идея, или гипотеза эксперта, или его вера в силу каких-то отдельных концепций. Этот принцип редко формулируется экспертом явно, он всегда как бы за "кадром", и искусство инженера по знаниям — обнаружить этот основной гештальт эксперта.

В гештальттеории существует закон "стремления к хорошему гештальту", согласно которому структуры сознания стремятся к гармонии, связности, простоте. Это близко к старинному классическому принципу "бритвы Оккама" — "сущности не должны умножаться без необходимости" — и формулируется как принцип прегнантности Вертгеймера [Вертгеймер, 1987]: "Организация поля имеет тенденцию быть настолько простой и ясной, насколько позволяют данные условия". Рассуждения о гештальте подводят вплотную к третьему этапу в структуре познания.

Э_3: построение модели

Необходим специализированный язык, посредством которого можно описывать и конструировать те идеализированные модели мира, которые возникают в процессе мышления. Язык этот создается постепенно с помощью категориального аппарата, принятого в соответствующей предметной области, а также формально-знаковых средств математики и логики. Для эмпирических предметных областей такой язык пока не разработан и поле знаний, которое полуформализованным способом опишет аналитик, может быть первым шагом к созданию такого языка.

Любое познавательное отражение включает в себя условность, т. е. упрощение и идеализацию. Инженеру по знаниям необходимо овладение такими специфическими гносеологическими приемами, как идеализация, огрубление, абстрагирование, которые позволяют адекватно отображать в модели реальную картину мира. Эти приемы доводят свойства и признаки объектов до пределов, позволяющих воспроизводить законы действительности в более лаконичном виде (без влияния несущественных деталей).

На тернистом пути познания проверенный диалектический подход оказывается лучшим "поводырем". Инженер по знаниям, который стремится познать проблемную область, должен быть готов постоянно изменять свои уже утвердившиеся способы восприятия и оценки мира и даже отказываться от них. При этом тщательнее всего следует проверять правильность суждений, которые кажутся самыми очевидными.

Э_4: объяснение и предсказание явлений

Этот завершающий этап является одновременно и частичным критерием истинности полученного знания. Если выявленная система знаний эксперта полна и объективна, то на ее основании можно делать прогнозы и объяснять любые явления из данной предметной области. Обычно базы знаний ЭС страдают фрагментарностью и модульностью (несвязанностью) компонентов. Все это не позволяет создавать действительно интеллектуальные системы, которые, равняясь на человека, могли бы предсказывать новые закономерности и объяснять случаи, не указанные в явном виде в базе. Исключением тут являются обучающие системы, которые ориентированы на генерацию новых знаний и на "предсказание".

Предлагаемая методология вооружает аналитика аппаратом, позволяющим избежать традиционных ошибок, приводящих к неполноте, противоречивости, фрагментарности БЗ, и указывает

направление, в котором необходимо двигаться разработчикам. И хотя на сегодняшний день большинство БЗ прорабатываются лишь до этапа Э_3, знание полной схемы обогащает и углубляет процесс проектирования.

2.4. Методы практического извлечения знаний

Рассмотрим, какими практическими методами можно получить знания. В предыдущих разделах по умолчанию предполагалось, что это некоторое взаимодействие инженера по знаниям и эксперта в форме непосредственного живого общения. В литературе [Волков, Ломнев, 1989; Осипов, 1998, Boose, 1989; Cullen, Bryman, 1988; Adeli, 1994] упоминаются около 15 ручных (неавтоматизированных) методов извлечения и более 20 автоматизированных методов приобретения и формирования знаний.

Рассмотрим классификацию методов извлечения знаний, которая позволит инженерам по знаниям в зависимости от конкретной задачи и ситуации выбрать подходящий метод (рис. 2.15).

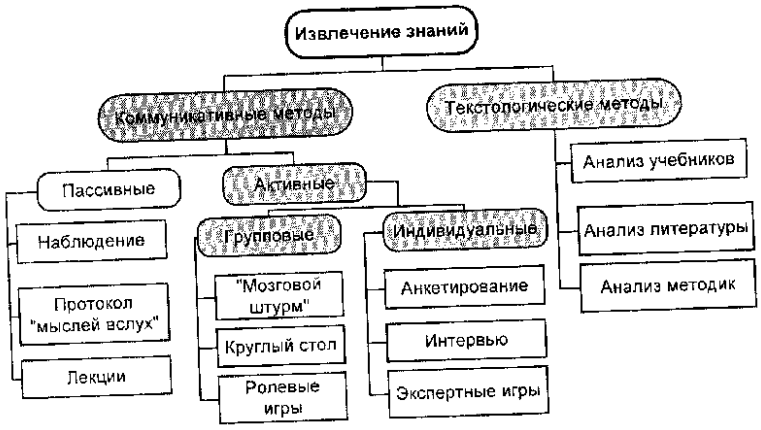

Рис. 2.15. Классификация методов извлечения знаний

При такой классификации в качестве основного принципа деления выступает источник знаний. Соответственно все методы делятся на:

- коммуникативные;
- текстологические.

Определение 2.6

Коммуникативные методы извлечения знаний **— это набор приемов и процедур, предполагающих контакт инженера по знаниям с непосредственным источником знаний — экспертом, а** ***текстологические*** **включают методы извлечения знаний из документов (методик, пособий, руководств) и специальной литературы (статей, монографий, учебников).**

Разделение этих групп методов на верхнем уровне классификации не означает их антагонистичности, обычно инженер по знаниям комбинирует различные методы, например сначала изучает литературу, затем беседует с экспертами, или наоборот.

В свою очередь, коммуникативные методы можно также разделить на две группы: *пассивные* и *активные*.

Пассивные методы подразумевают, что ведущая роль в процедуре извлечения передается эксперту, а инженер по знаниям только протоколирует рассуждения эксперта во время его реальной работы по принятию решений или записывает то, что эксперт считает нужным самостоятельно рассказать в форме лекции. В активных методах, напротив, инициатива полностью в руках инженера по знаниям, который активно контактирует с экспертом различными способами — в играх, диалогах, беседах за круглым столом и т. д.

Следует еще раз подчеркнуть, что и активные и пассивные методы могут чередоваться даже в рамках одного сеанса извлечения знаний. Например, если инженер по знаниям застенчив и не

имеет большого опыта, то вначале он может использовать пассивные методы, а постепенно, ближе знакомясь с экспертом, захватывать инициативу и переходить "в наступление".

Пассивные методы на первый взгляд достаточно просты, но на самом деле требуют от инженера по знаниям умения четко анализировать поток сознания эксперта и выявлять в нем значимые фрагменты знаний. Отсутствие обратной связи (пассивность инженера по знаниям) значительно ослабляет эффективность этих методов, чем и объясняется их обычно вспомогательная роль при активных методах.

Активные методы можно разделить на две группы в зависимости от числа экспертов, отдающих свои знания. Если их число больше одного, то целесообразно помимо серии индивидуальных контактов с каждым применять и методы групповых обсуждений предметной области.

Такие групповые методы обычно активизируют мышление участников дискуссий и позволяют выявлять весьма нетривиальные аспекты их знаний. В свою очередь, индивидуальные методы на сегодняшний день остаются ведущими, поскольку столь деликатная процедура, как "отъем знаний", не терпит лишних свидетелей.

Игровые методы сейчас широко используются в социологии, экономике, менеджменте, педагогике для подготовки руководителей, учителей, врачей и других специалистов. Игра — это особая форма деятельности и творчества, где человек раскрепощается и чувствует себя намного свободнее, чем в обычной трудовой деятельности.

На выбор метода влияют три фактора: личностные особенности инженера по знаниям, личностные особенности эксперта и характеристика предметной области.

Одна из возможных классификаций людей по психологическим характеристикам [Обозов, 1986] делит всех на три типа:

- мыслитель (познавательный тип);
- собеседник (эмоционально-коммуникативный тип);
- практик (практический тип).

Мыслители ориентированы на интеллектуальную работу, учебу, теоретические обобщения и обладают такими характеристиками когнитивного стиля, как поленезависимость и рефлексивность *(см. разд. 2.3).*

Собеседники — это общительные, открытые люди, готовые к сотрудничеству. Практики предпочитают действие разговорам, хорошо реализуют замыслы других, направлены на результативность работы.

Для характеристики предметных областей можно предложить следующую классификацию (рис. 2.16):

- хорошо документированные;
- средне документированные;
- слабо документированные.

Эта классификация связана с соотношением двух видов знаний Z_1 и Z_2, введенных в *разд. 1.1,* где Z_1 — это экспертное "личное" знание, а Z_2 — материализованное в книгах "общее" знание в данной конкретной области. Если представить знания *Zno* предметной области как объединение Z_1 и Z_2, т. е. *Zno* = Z1˘Z2, то рис. 2.16 наглядно иллюстрирует предложенную классификацию.

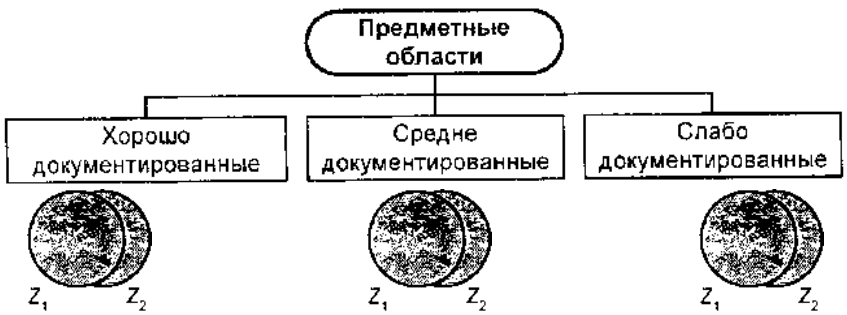

Рис. 2.16. Классификация предметных областей

Кроме этого, предметные области можно разделить по критерию структурированности знаний. Под структурированностью будем понимать степень теоретического осмысления и выявленности основных

закономерностей и принципов, действующих в данной предметной области. И хотя ЭС традиционно применяются в слабо структурированных предметных областях, сейчас наблюдается тенденция расширения сферы внедрения экспертных систем.

По степени структурированности знаний предметные области могут быть:

- *хорошо структурированными* — с четкой аксиоматизацией, широким применением математического аппарата, устоявшейся терминологией;
- *средне структурированными* — с определившейся терминологией, развивающейся теорией, явными взаимосвязями между явлениями;
- *слабо структурированными* — с размытыми определениями, богатой эмпирикой, скрытыми взаимосвязями, с большим количеством “белых пятен”.

Введенные в данном разделе классификации методов и предметных областей помогут инженеру по знаниям, четко определив свою предметную область, соотнести ее с предложенными типами и наметить подходящий метод или группу методов извлечения знаний. Однако, скорее всего, реальная работа полностью зачеркнет его выбор, и окажется, что его хорошо документированная область является слабо документированной, а метод наблюдений надо срочно заменять играми!
Такова реальная сложность процедур извлечения знаний, требующая специальной подготовки аналитика. Лучшая подготовка — это повышение квалификации на специальных курсах (например, в “Школе аналитика” (**http//:www.ber.ru**). Самоподготовка (рис. 2.17) включает:

- общую;
- специальную (частично может совпадать с предлагаемой в [Шумилина, 1973] подготовкой к журналистскому интервью), дополненную в связи со спецификой инженерии знаний;
- конкретную (базируется на прочтении специальных текстов, выбранных совместно с экспертами из некоторого "базового" списка литературы, который постепенно введет аналитика в предметную область. В этом списке могут быть учебники для начинающих, главы и фрагменты из монографий, популярные издания);
- психологическую.

Подготовка занимает разное время в зависимости от степени профессионализма аналитика, но в любом случае она необходима, т. к. несколько уменьшает вероятность самого нерационального метода — метода проб и ошибок.

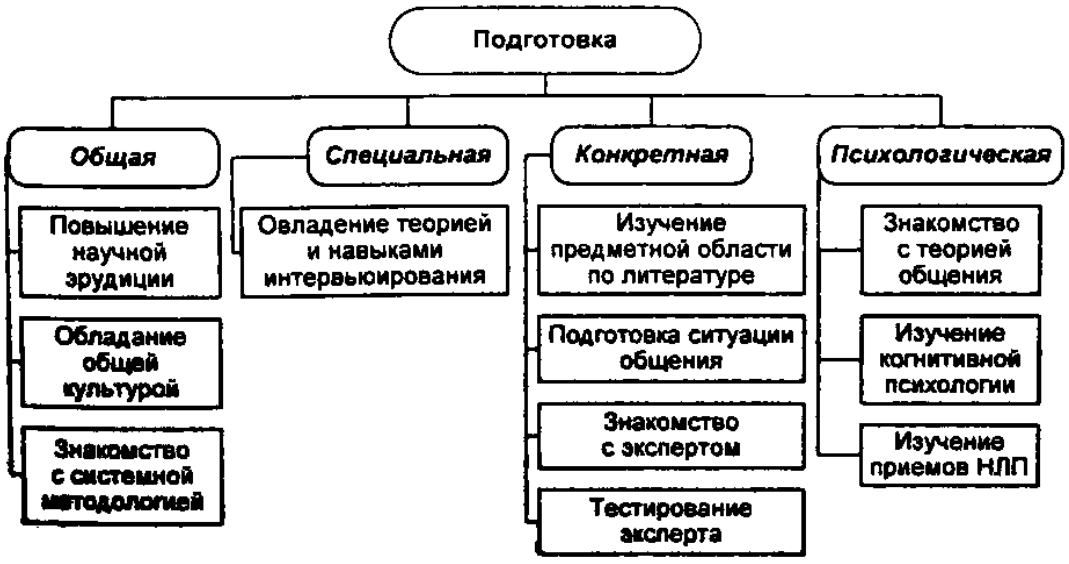

Рис. 2.17. Подготовка к извлечению знаний

2.5. Практикум по инженерии знаний

2.5.1. Текстологические методы

С извлечением знаний из текстов мы сталкиваемся ежедневно. Этот способ всегда должен предшествовать коммуникативным методам, подготавливая аналитика и знакомя его с терминологией и основными идеями.

Группа текстологических методов объединяет методы извлечения знаний, основанные на изучении специальных текстов из учебников, монографий, статей, методик и других носителей профессиональных знаний.

В буквальном смысле текстологические методы не относятся к текстологии, науке, которая родилась в русле филологии с целью критического прочтения литературных текстов, изучения и интерпретации источников с узкоприкладной задачей — подготовки текстов к изданию. Сейчас текстология расширила свои границы включением аспектов смежных наук — герменевтики (науки правильного толкования древних текстов — библии, античных рукописей и др.), семиотики, психолингвистики и др.

Текстологические методы извлечения знаний, безусловно, используя основные положения текстологии, отличаются принципиально от ее методологии, во-первых, характером и природой своих источников (профессиональная специальная литература, а не художественная, живущая по своим особым законам), а во-вторых, жесткой прагматической направленностью извлечения конкретных профессиональных знаний.

Среди методов извлечения знаний эта группа является наименее разработанной, по ней практически нет никакой библиографии, поэтому дальнейшее изложение является как бы введением в методы изучения текстов в том виде, как это представляют авторы.

Понимание текста

Задачу извлечения знаний из текстов можно сформулировать как задачу понимания и выделения смысла текста. Сам текст на естественном языке является лишь проводником смысла, а замысел и знания автора лежат во вторичной структуре (смысловой структуре или макроструктуре текста), настраиваемой над естественным текстом [Величковский, Капица, 1987], или, как сформулировано в [Фаин, 1987], "текст не содержит и не передает смысл, а является лишь инструментом для автора текста".

При этом можно выделить две такие смысловые структуры: M_1 — смысл, который пытался заложить автор, это его модель мира, и M_2 — смысл, который постигает читатель, в данном случае инженер по знаниям (рис. 2.18) в процессе интерпретации I. При этом T— это словесное одеяние M_1, т. е. результат вербализации V

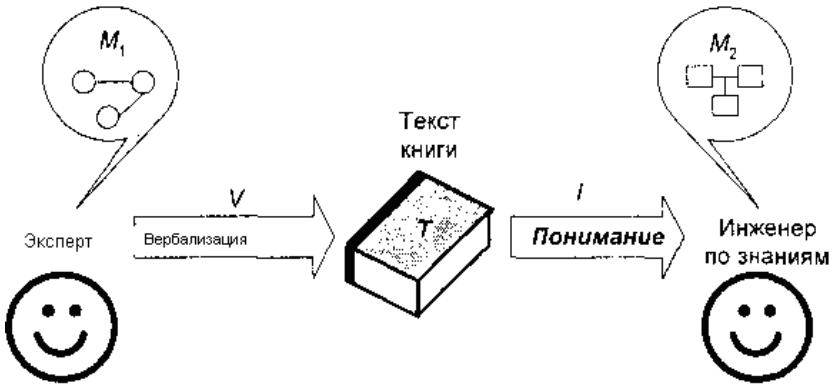

Рис. 2.18. Схема извлечения знаний из специальных текстов

Сложность процесса заключается в принципиальной невозможности совпадения знаний, образующих M_1 и M_2, из-за того, что M_1 образуется за счет всей совокупности представлений, потребностей, интересов и опыта автора, лишь малая часть которых находит отражение в тексте T. Соответственно и M_2 образуется в процессе интерпретации текста T за счет привлечения всей совокупности профессионального и человеческого багажа читателя. Таким образом, два инженера по знаниям извлекут из одного T две различные модели: M^i_1 и M^i_2.

Встает задача: выяснить, за счет чего можно достичь максимальной адекватности M_1 и M_2, помня при этом, что понимание всегда относительно, поскольку это синтез двух смыслов "свое — чужое" [Бахтин, 1975].

Рассмотрим подробнее, какие источники питают модель M_1 и создают текст T. В [Сергеев, 1987] указаны два компонента любого профессионального текста:

- первичный фактический материал;
- система понятий, сложившаяся в данной предметной области в момент создания текста.

В дополнение к этому, на наш взгляд, помимо объективных данных экспериментов и наблюдений, в тексте обязательно присутствуют субъективные взгляды автора, результат его личного опыта, а также некоторые "общие места" или "вода". Кроме этого, любой текст содержит заимствования из других источников (методики, указания, документы, статьи, монографии) и т. д.

При извлечении знаний аналитику, интерпретирующему текст, приходится решать задачу декомпозиции этого текста на перечисленные выше компоненты для выделения истинно значимых для реализации базы знаний фрагментов. Сложность интерпретации профессиональных текстов заключается еще и в том, что любой текст приобретает смысл только в контексте, где под контекстом понимается окружение, в которое "погружен" текст.

Различают микро- и макроконтекст. *Микроконтекст* — это ближайшее окружение текста. Так, предложение получает смысл в контексте абзаца, абзац в контексте главы и т. д. *Макроконтекст* — это вся система знаний, связанная с предметной областью (т. е. знания об особенностях и свойствах, явно не указанных в тексте). Другими словами, любое знание обретает смысл в контексте некоторого метазнания.

Теперь несколько подробнее о центральном звене процедуры извлечения знания — о понимании текста. Классическим в текстологии является определение немецкого философа и языковеда Гумбольдта [Гумбольдт, 1984]:

> "...Люди понимают друг друга не потому, что передают собеседнику знаки предметов, и даже не потому, что взаимно настраивают друг друга на точное и полное воспроизведение идентичного понятия, а потому, что взаимно затрагивают друг в друге одно и то же звено цепи чувственных представлений и зачатков внутренних понятий, прикасаются к одним и тем же клавишам инструмента своего духа, благодаря чему у каждого вспыхивает в сознании соответствующие, но не тождественные смыслы".

Говоря на языке современного языкознания, *понимание* — это формирование "второго текста", т. е. семантической структуры (понятийной структуры) [Сиротко-Сибирский, 1968]. В нашей терминологии — это попытка воссоздания семантической структуры M_1 в процессе формирования модели M_2, т. е. это первый шаг структурирования знаний.

Как происходит процесс понимания? Одна из возможных схем изложена в [Соколов, 1947; Соколов, 1968]. Мы внесли несколько изменений в эту схему в связи с тем, что в ней трактуется понимание текста на иностранном языке, а нас интересует понимание текста в новой для познающего субъекта предметной области. Кроме этого, дополним ее некоторыми положениями герменевтики. В целом полученная схема согласуется со стратегией изучения всего нового.

Основными моментами понимания текста являются следующие шаги:

1.Выдвижение предварительной гипотезы о смысле всего текста (предугадывание).

2.Определение значений непонятных слов (т. е. специальной терминологии).

3.Возникновение общей гипотезы о содержании текста (о знаниях).

4.Уточнение значения терминов и интерпретация отдельных фрагментов текста под влиянием общей гипотезы (от целого к частям).

5.Формирование некоторой смысловой структуры текста за счет установления внутренних связей между отдельными важными (ключевыми) словами и фрагментами, а также за счет образования абстрактных понятий, обобщающих конкретные фрагменты знаний.

6.Корректировка общей гипотезы относительно содержащихся в тексте фрагментов знаний (от частей к целому).

7.Принятие основной гипотезы, т. е. формирование M_2.

Следует отметить наличие как дедуктивной (от целого к частям), так и индуктивной (от частей к целому) составляющей процесса понимания. Такой двуединый подход позволяет охватывать текст как смысловое единство особого рода с его основными признаками, такими как связность, цельность, законченность и др. [Сиротко-Сибирский, 1968].

Смысловая структура текста

Центральными моментами процесса *I* являются шаги 5 и 7, т. е. формирование смысловой структуры или выделение "опорных", ключевых, слов или "смысловых вех" [Сиротко-Сибирский, 1968], а также заключительное связывание "смысловых вех" в единую семантическую структуру.

При анализе текста важно выявление внутренних связей между отдельными элементами текста и понятиями. Традиционно выделяют два вида связей в тексте — *эксплицитные* (или явные связи), которые выражаются во внешнем дроблении текста, и *имплицитные* (скрытые связи).

Эксплицитные связи делят текст на параграфы с помощью перечисления компонентов, вводных слов (или коннекторов) типа "во-первых..., во-вторых..., однако и т. д.".

Имплицитные, или внутренние, связи между отдельными "смысловыми вехами" вызывают основное затруднение при понимании.

Итак, семантическая структура текста образуется в сознании познающего субъекта с помощью знаний о языке, знаний о мире, а также общих (фоновых) знаний в той предметной области, которой посвящен текст. "Тексты пишут для посвященных". Другими словами, если текст не является научно-популярным, то для его адекватного прочтения требуется некоторая подготовка, например чтение популярных учебников на уровне менеджеров среднего звена.

Процесс *I* — это сложный, не поддающийся формализации процесс, на который существенным образом влияют такие чисто индивидуальные компоненты, как когнитивный стиль познания, интеллектуальные характеристики и др.

Но процедура разбивки текста на части ("смысловые группы"), а затем сгущение, сжатие содержимого каждого смыслового куска в "смысловую веху" является, видимо, основой для любого индивидуального процесса понимания. Такая компрессия (сжатие) текста в виде набора ключевых слов, передающих основное содержание текста, может служить удобной методологической основой для проведения текстологических процедур извлечения знаний.

В качестве ключевого слова может служить любая часть речи (существительное, прилагательное, глагол и т. д.) или их сочетание. *Набор ключевых слов* (НКС) — это набор опорных точек, по которым развертывается текст при кодировании в память и осознается при декодировании, это семантическое ядро цельности [Сиротко-Сибирский, 1968].

Одна из гипотез лингвостатистики о том, что чаще всего употребляемые слова являются наиболее важными с точки зрения содержания текста, т. е. отражают его тематическую структуру, подтверждается при работе с профессиональными текстами.

Следует сказать несколько слов о том, почему мы выделяем три вида текстологических методов (см. рис. 2.15):

- анализ специальной литературы;
- анализ учебников;
- анализ методик.

Перечисленные три метода отличаются, во-первых, по степени концентрированности специальных знаний, и, во-вторых, по соотношению специальных и фоновых знаний. Наиболее простым методом является анализ учебников, в которых логика изложения обычно соответствует логике предмета и поэтому структура такого текста будет, наверное, более значима, чем структура текста какой-нибудь специальной статьи. Анализ методик затруднен как раз сжатостью изложения и практическим отсутствием комментариев, т. е. фоновых знаний, облегчающих понимание для неспециалистов. Поэтому можно рекомендовать для практической работы комбинацию перечисленных методов.

Алгоритм извлечения знаний из текста

В заключение предложим одну из возможных практических методик анализа текстов с целью извлечения и структурирования знаний.

1.Составление "базового" списка литературы для ознакомления с предметной областью и чтение по списку.

2.Первое знакомство с текстом (беглое прочтение) с выписыванием незнакомых слов.

3.Консультации со специалистами или привлечение справочной литературы для их понимания.

4.Внимательное (второе) прочтение текста с выписыванием наборов ключевых слов (ИКС), т. е. выделение "смысловых вех" (компрессия текста).

5.Определение связей между НКС, разработка семантической структуры текста в форме графа или "сжатого" текста (реферата), фактически формирование поля знаний.

6.Третье прочтение текста и коррекция поля знаний при необходимости.

2.5.2. Коммуникативные методы

Пассивные методы

Термин "пассивные" не должен вызывать иллюзий, в реальности же пассивные методы требуют от инженера по знаниям не меньшей отдачи, чем такие активные методы, как игры и диалог.

Определение 2.7

Пассивные методы **извлечения знаний включают методы, где ведущая роль в процедуре извлечения фактически передается эксперту, а инженер по знаниям только фиксирует рассуждения эксперта во время работы по принятию решений.**

Согласно классификации (см. рис. 2.14) к этой группе относятся:

- наблюдения;
- анализ протоколов "мыслей вслух";
- лекции.

Наблюдения

В процессе наблюдений инженер по знаниям находится непосредственно рядом с экспертом во время его профессиональной деятельности или имитации этой деятельности. При подготовке к сеансу извлечения эксперту необходимо объяснить цель наблюдений и попросить максимально комментировать свои действия.

Во время сеанса аналитик записывает все действия эксперта, его реплики и объяснения. Полезной может оказаться и видеозапись в реальном масштабе времени, если эксперт согласится. Непременное условие этого метода — невмешательство аналитика в работу эксперта хотя бы на первых порах. Именно метод наблюдений является единственно "чистым" методом, исключающим вмешательство инженера по знаниям и навязывание им каких-то своих структур представлений.

Существуют две основные разновидности проведения наблюдений:

- наблюдение за реальным процессом;
- наблюдение за имитацией процесса.

Обычно используются обе разновидности. Сначала инженеру по знаниям полезно наблюдать за реальным процессом, чтобы глубже понять предметную область и отметить все внешние особенности процесса принятия решения. Это необходимо для проектирования эффективного интерфейса пользователя. Ведь будущая ЭС должна работать именно в контексте такого реального производственного процесса. Кроме того, только наблюдение позволит аналитику увидеть предметную область, а, как известно, "лучше один раз увидеть, чем сто раз услышать".

Наблюдение за имитацией процесса проводят обычно также за рабочим местом эксперта, но сам процесс деятельности запускается специально для аналитика. Преимущество этой разновидности состоит в том, что эксперт менее напряжен, чем в первом варианте, когда он работает на "два фронта" — и ведет профессиональную деятельность, и демонстрирует ее. Недостаток совпадает с преимуществом — именно меньшая напряженность эксперта может повлиять на результат — раз работа ненастоящая, то и решение может отличаться от настоящего.

Наблюдения за имитацией проводят также и в тех случаях, когда наблюдения за реальным процессом по каким-либо причинам невозможны (например, профессиональная этика врача-психиатра может не допускать присутствия постороннего на приеме).

Сеансы наблюдений могут потребовать от инженера по знаниям:

- овладения техникой стенографии для фиксации действий эксперта в реальном масштабе времени;
- ознакомления с методиками хронометража для четкого структурирования производственного процесса по времени;
- развития навыков "чтения по глазам", т. е. наблюдательности к жестам, мимике и другим невербальным компонентам общения;
- серьезного предварительного знакомства с предметной областью, т. к. из-за отсутствия "обратной связи" иногда многое непонятно в действиях экспертов.

Протоколы наблюдений после сеансов в ходе домашней работы тщательно расшифровываются, а затем обсуждаются с экспертом.

Таким образом, наблюдения — один из наиболее распространенных методов извлечения знаний на начальных этапах разработки. Обычно он применяется не самостоятельно, а в совокупности с другими методами.

Анализ протоколов "мыслей вслух"

Протоколирование "мыслей вслух", или "вербальные отчеты" [Моргоев, 1988], отличается от наблюдений тем, что эксперта просят не просто прокомментировать свои действия и решения, но и объяснить, как это решение было найдено, т. е. продемонстрировать всю цепочку своих рассуждений. Во время рассуждений эксперта все его слова, весь "поток сознания" протоколируется инженером по знаниям, при этом полезно отметить даже паузы и междометия.

Вопрос об использовании для этой цели магнитофонов и диктофонов является дискуссионным, поскольку магнитофон иногда парализующе действует на эксперта, разрушая атмосферу доверительности, которая может и должна возникать при непосредственном общении.

Основной трудностью при протоколировании "мыслей вслух" является принципиальная сложность для любого человека объяснить, как он думает. При этом существуют экспериментальные психологические доказательства того факта, что люди не всегда в состоянии достоверно описывать мыслительные процессы. Кроме того, часть знаний, хранящихся в невербальной форме (например, различные процедурные знания типа "как завязывать шнурки"), вообще слабо коррелируют с их словесным описанием. Автор теории фреймов М. Минский считает, что "только как исключение, а не как правило, человек может объяснить то, что он знает" [Minsky, 1981]. Однако существуют люди, склонные к рефлексии, для которых эта работа является вполне доступной. Следовательно, описанная в *разд. 3.3* такая характеристика когнитивного стиля, как рефлексивность, является для эксперта более чем желательной.

Расшифровка полученных протоколов производится инженером по знаниям самостоятельно, с коррекциями на следующих сеансах извлечения знаний. Удачно проведенное протоколирование "мыслей вслух" является одним из наиболее эффективных методов извлечения, поскольку в нем эксперт может проявить себя максимально ярко, он ничем не скован, никто ему не мешает, он как бы свободно парит в потоке собственных умозаключений и рассуждений. Он может здесь блеснуть эрудицией, продемонстрировать глубину своих познаний. Для большого числа экспертов это самый приятный и лестный способ извлечения знаний.

От инженера по знаниям метод "мысли вслух" требует тех же умений, что и метод наблюдений. Обычно "мысли вслух" дополняются потом одним из активных методов для реализации обратной связи между интерпретацией инженера по знаниям и представлениями эксперта.

Лекции

Лекция является самым старым способом передачи знаний. Лекторское искусство издревле очень высоко ценилось во всех областях науки и культуры.

Но нас сейчас интересует не столько способность к подготовке и чтению лекций, сколько способность эту лекцию слушать, конспектировать и усваивать. Уже говорилось, что чаще всего экспертов не выбирают, и поэтому учить эксперта чтению лекции инженер по знаниям не сможет. Но если эксперт имеет опыт преподавателя (например, профессор клиники или опытный руководитель производства), то можно воспользоваться таким концентрированным фрагментом знаний, как лекция.

В лекции эксперту предоставлено много степеней свободы для самовыражения; при этом необходимо сформулировать эксперту тему и задачу лекции. Например, тема цикла лекций "Постановка диагноза — воспаление легких", тема конкретной лекции "Рассуждения по анализу рентгенограмм", задача — научить слушателей по перечисленным экспертом признакам ставить диагноз воспаления легких и делать прогноз. При такой постановке опытный лектор может заранее структурировать свои знания и ход рассуждений. От инженера по знаниям в этой ситуации требуется лишь грамотно законспектировать лекцию и в конце задать необходимые вопросы.

Студенты хорошо знают, что конспекты лекций одного и того же лектора у разных студентов существенно отличаются. Списать конспект лекций просят, как правило, у одного-двух студентов из группы. Люди, умело ведущие конспект, обычно сильные студенты. Обратное не верно. В чем же заключается искусство ведения конспекта? В "помехоустойчивости". Записывать главное, опускать второстепенное, выделять фрагменты знаний (параграфы, подпараграфы), записывать только осмысленные предложения, уметь обобщать.

Хороший вопрос по ходу лекции помогает и лектору, и слушателю. Серьезные и глубокие вопросы могут существенно поднять авторитет инженера по знаниям в глазах эксперта.

Опытный лектор знает, что все вопросы можно условно разбить на три группы:

- умные вопросы, углубляющие лекцию;
- глупые вопросы или вопросы не по существу;
- вопросы "на засыпку", или провокационные.

Если инженер по знаниям задает вопросы второго типа, то возможны две реакции. Вежливый эксперт будет разговаривать с таким аналитиком как с ребенком, который сейчас не понимает и все равно ничего уже не поймет. Заносчивый эксперт просто выйдет из контакта, не желая терять время. Если же инженер по знаниям захочет продемонстрировать свою эрудицию вопросами третьего типа, то ничего, кроме раздражения и отчуждения, он, по-видимому, в ответ не получит.

Продолжительность лекции рекомендуется стандартная — от 40 до 50 минут и через 5—10 минут — еще столько же. Курс обычно от двух до пяти лекций.

Метод извлечения знаний в форме лекций, как и все пассивные методы, используют в начале разработки как эффективный способ быстрого погружения инженера по знаниям в предметную область.

Активные индивидуальные методы

Активные индивидуальные методы извлечения знаний на сегодняшний день — наиболее распространенные. В той или иной степени к ним прибегают при разработке практически любой ЭС.

К основным активным методам можно отнести:

- анкетирование;
- интервью;
- игры с экспертом.

Во всех этих методах активную функцию выполняет инженер по знаниям, который пишет сценарий и режиссирует сеансы извлечения знаний. Игры с экспертом существенно отличаются от других *вопросных* методов.

Анкетирование

Анкетирование — самый жесткий метод, т. е. наиболее стандартизированный. В этом случае инженер по знаниям заранее составляет вопросник или анкету, размножает ее и использует для опроса нескольких экспертов. Это основное преимущество анкетирования.

Сама процедура может проводиться двумя способами:

- аналитик вслух задает вопросы и сам заполняет анкету по ответам эксперта;
- эксперт самостоятельно заполняет анкету после предварительного инструктирования.

Выбор способа зависит от конкретных условий (например, от оформления анкеты, ее понятности, готовности эксперта). Второй способ нам кажется предпочтительнее, т. к. у эксперта появляется неограниченное время на обдумывание ответов.

Основными факторами, на которые можно существенно повлиять при анкетировании, являются средства общения (в данном случае это вопросник) и ситуация общения.

Вопросник (анкета) заслуживает особого разговора. Существует несколько общих рекомендаций при составлении анкет. Эти рекомендации являются универсальными, т. е. не зависят от предметной области. Наибольший опыт работы с анкетами накоплен в социологии и психологии, поэтому часть рекомендаций заимствована из [Ноэль, 1978; Погосян, 1985].

- Анкета не должна быть монотонной и однообразной, т. е. вызывать скуку или усталость. Это достигается вариациями формы вопросов, сменой тематики, вставкой вопросов-шуток и игровых вопросов.
- Анкета должна быть приспособлена к языку экспертов.
- Следует учитывать, что вопросы влияют друг на друга, и поэтому последовательность вопросов должна быть строго продумана.
- Желательно стремиться к оптимальной избыточности. Известно, что в анкете всегда много лишних вопросов, часть из них необходима — это так называемые контрольные вопросы *(см. о них ниже),* а другая часть должна быть минимизирована.

  Лишние вопросы появляются, например, в таких ситуациях. Фрагмент анкеты: "В12. Считаете ли Вы, что для лечения ангины эффективен эритромицин? В13. Какие дозы эритромицина Вы обычно рекомендуете?"
  При отрицательном ответе на 12-й вопрос 13-й является лишним. Его можно избежать, усложнив вопрос.
  "В12. Применяете ли Вы эритромицин для лечения ангины и если да, то в каких дозах?"

- Анкета должна иметь "хорошие манеры", т. е. ее язык должен быть ясным, понятным, предельно вежливым.

Методическим мастерством составления анкеты можно овладеть только на практике.

Интервью

Под *интервью* будем понимать специфическую форму общения инженера по знаниям и эксперта, в которой инженер по знаниям задает эксперту серию заранее подготовленных вопросов с целью извлечения знаний о предметной области. Наибольший опыт в проведении интервью накоплен, наверное, в журналистике и социологии. Большинство специалистов этих областей отмечают, тем не менее, крайнюю недостаточность теоретических и методических исследований по тематике интервьюирования [Ноэль, 1978; Шумилина, 1973].

Интервью очень близко тому способу анкетирования, когда аналитик сам заполняет анкету, занося туда ответы эксперта. Основное отличие интервью в том, что оно позволяет аналитику опускать ряд вопросов в зависимости от ситуации, вставлять новые вопросы в анкету, изменять темп, разнообразить ситуацию общения. Кроме этого, у аналитика появляется возможность "взять в плен" эксперта своим обаянием, заинтересовать его самой процедурой и, тем самым, увеличить эффективность сеанса извлечения.

Вопросы для интервью. Теперь несколько подробнее о центральном звене активных индивидуальных методов — о вопросах. Инженеры по знаниям редко сомневаются в своей способности задавать вопросы. В то время как и в философии, так в математике эта проблема обсуждается с давних лет. Существует даже специальная ветвь математической логики — эротетическая логика (логика вопросов). Есть интересная работа Белнапа и Стила "Логика вопросов и ответов" [Белнап, Стил, 1981], но, к сожалению, использовать результаты, полученные логиками, непосредственно при разработке интеллектуальных систем не удается.

Все вопросительные предложения можно разбить на два типа [Сергеев, Соколов, 1986]:

- вопросы *с неопределенностью,* относящейся ко всему предложению ("Действительно, введение больших доз антибиотиков может вызвать анафилактический шок?);
- вопросы *с неполной информацией* ("При каких условиях необходимо включать кнопку?"), часто начинающиеся со слов "кто", что", "где", "когда" и т. д.

Это разделение можно дополнить классификацией, частично описанной в [Шумилина, 1973] и представленной на рис. 2.19.

Открытый вопрос называет тему или предмет, оставляя полную свободу эксперту по форме и содержанию ответа ("Не могли бы Вы рассказать, как лучше сбить высокую температуру у больного с воспалением легких?").

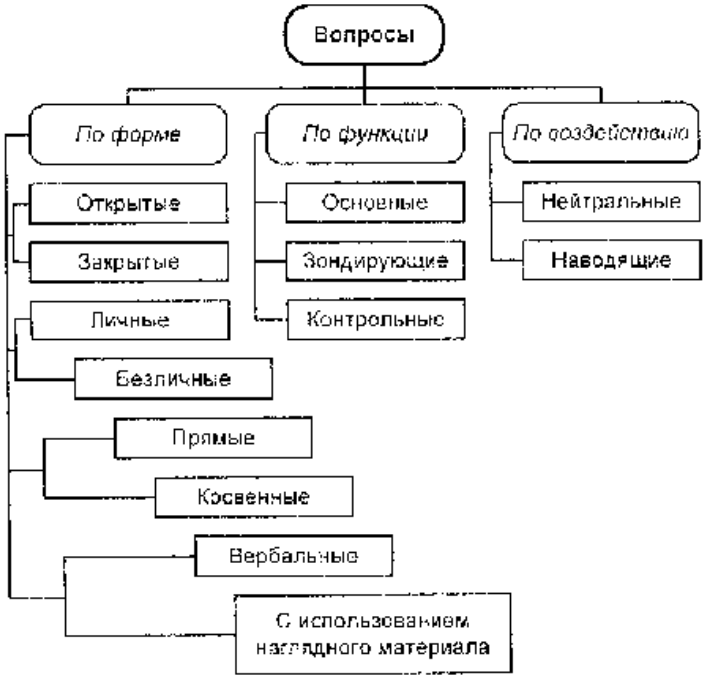

Рис. 2.19. Классификация вопросов

В закрытом вопросе эксперт выбирает ответ из набора предложенных ("Укажите, пожалуйста, что Вы рекомендуете при ангине: а) антибиотики, б) полоскание, в) компрессы, г) ингаляции"). Закрытые вопросы легче обрабатывать при последующем анализе, но они более опасны, т. к. "закрывают" ход рассуждений эксперта и "программируют" его ответ в определенном направлении. При составлении сценария интервью полезно чередовать открытые и закрытые вопросы, особенно тщательно продумывать закрытые, поскольку для их составления требуется определенная эрудиция в предметной области.

Личный вопрос касается непосредственно индивидуального опыта эксперта ("Скажите, пожалуйста, Иван Данилович, в Вашей практике Вы применяете вулнузан при фурункулезах?").

Личные вопросы обычно активизируют мышление эксперта, "играют" на его самолюбии, они всегда украшают интервью.

Безличный вопрос направлен на выявление наиболее распространенных и общепринятых закономерностей предметной области ("Что влияет на скорость процесса ферментации лизина?").

При составлении вопросов следует учитывать, что языковые способности эксперта, как правило, ограничены и вследствие скованности, замкнутости, робости он не может сразу высказать свое мнение и предоставить знания, которые от него требуются (даже если предположить, что он их четко для себя формулирует). Поэтому часто при "зажатости" эксперта используют не прямые вопросы, которые непосредственно указывают на предмет или тему ("Как Вы относитесь к методике доктора Сухарева?"), а косвенные, которые лишь косвенно указывают на интересующий предмет ("Применяете ли Вы методику доктора Сухарева? Опишите, пожалуйста, результаты лечения"). Иногда приходится задавать несколько десятков косвенных вопросов вместо одного прямого.

Вербальные вопросы — это традиционные устные вопросы. Вопросы с использованием наглядного материала вносят разнообразие в интервью и снижают утомляемость эксперта. В этих вопросах используют фотографии, рисунки и карточки. Например, эксперту предлагаются цветные картонные карточки, на которых выписаны признаки заболевания. Затем аналитик просит разложить эти карточки в порядке убывания значимости для постановки диагноза.

Деление вопросов по функции на основные, зондирующие, контрольные связано с тем, что часто основные вопросы интервью, направленные на выявление знаний, не срабатывают — эксперт по каким-то причинам уходит в сторону от вопроса, отвечает нечетко.

Тогда аналитик использует зондирующие вопросы, которые направляют рассуждения эксперта в нужную сторону. Например, если не сработал основной вопрос: "Какие параметры определяют момент окончания процесса ферментации лизина?", аналитик начинает задавать зондирующие вопросы:

"Всегда ли процесс ферментации длится 72 часа? А если он заканчивается раньше, как это узнать? Если он продлится больше, то что заставит микробиолога не закончить процесс на 72-м часу?" и т. д.

Контрольные вопросы применяют для проверки достоверности и объективности информации, полученной в интервью ранее ("Скажите, пожалуйста, а московская школа психологов так же как Вы трактует шкалу К опросника MMPI?" или "Рекомендуете ли Вы инъекции АТФ?" (АТФ — препарат, снятый с производства). Контрольные вопросы должны быть "хитро" составлены, чтобы не обидеть эксперта недоверием (для этого используют повторение вопросов в другой форме, уточнения, ссылки на иные источники). "Лучше два раза спросить, чем один раз напутать" (Шолом Алейхем).

И наконец, о нейтральных и наводящих вопросах. В принципе интервьюеру (в нашем случае инженеру по знаниям) рекомендуют быть беспристрастным, отсюда и вопросы его должны носить *нейтральный характер,* т. е. не должны указывать на отношение интервьюера к данной теме. Напротив, наводящие вопросы заставляют респондента (в данном случае эксперта) прислушаться или даже принять во внимание позицию интервьюера. Нейтральный вопрос: "Совпадают ли симптомы кровоизлияния в мозг и сотрясения мозга?" Наводящий вопрос: "Не правда ли, очень трудно дифференцировать симптомы кровоизлияния в мозг?"

Кроме перечисленных выше, полезно различать и включать в интервью следующие вопросы [Ноэль, 1978]:

- контактные ("ломающие лед" между аналитиком и экспертом);
- буферные (для разграничения отдельных тем интервью);
- оживляющие память экспертов (для реконструкции отдельных случаев из практики);
- "провоцирующие" (для получения спонтанных, неподготовленных ответов).

Три основные характеристики вопросов [Шумилина, 1973], которые влияют на качество интервью:

- язык вопроса (понятность, лаконичность, терминология);
- порядок вопросов (логическая последовательность и немонотонность);
- уместность вопросов (этика, вежливость).

Вопрос в интервью — это не просто средство общения, но и способ передачи мыслей и позиции аналитика.

"Вопрос представляет собой форму движения мысли, в нем ярко выражен момент перехода от незнания к знанию, от неполного, неточного знания к более полному и более точному"

[Лимантов, 1971]. Отсюда необходимость в протоколах фиксировать не только ответы, но и вопросы, предварительно тщательно отрабатывая их форму и содержание.

Очевидно, что любой вопрос имеет смысл только в контексте. Поэтому вопросы может готовить инженер по знаниям, уже овладевший ключевым набором знаний.

Вопросы имеют для эксперта диагностическое значение — несколько откровенных "глупых" вопросов могут полностью разочаровать эксперта и отбить у него охоту к дальнейшему сотрудничеству. Известен ответ Маркса на вопрос Прудона: "Вопрос был до такой степени неправильно поставлен, что на него невозможно было дать правильный ответ".

У опытного аналитика интервью внешне может быть похоже на свободный диалог — это метод извлечения знаний в форме беседы инженера по знаниям и эксперта, в которой нет жесткого регламентированного плана и вопросника. Это определение не означает, что к свободному диалогу не надо готовиться. Напротив, внешне свободная и легкая форма этого метода требует высочайшей профессиональной и психологической подготовки.

Квалифицированная подготовка к диалогу помогает аналитику стать истинным драматургом или сценаристом будущих сеансов, т. е. запланировать гладкое течение процедуры извлечения: от приятного впечатления в начале беседы переход к профессиональному контакту через пробуждение интереса и завоевание доверия эксперта. При этом для обеспечения желания эксперта продолжить беседу необходимо проводить "поглаживания" (терминология Э. Берна [Берн, 1988]), т. е. подбадривать эксперта и подтверждать всячески его уверенность в собственной компетентности (фразы-вставки: "Я Вас понимаю...", "...это очень интересно" и т. д.)

Так, в одном из исследований до техники ведения профессиональных журналистских диалогов [Matarozzo, Wetttnan, Weins, 1963] было экспериментально доказано, что одобрительное и поощрительное "хмыканье" интервьюера увеличивает длину ответов респондента. При этом одобрение должно быть искренним, как показал опрос интервьюеров Института демоскопии Германии: "Лучшая уловка — это избегать всяких уловок: относиться к опрашиваемому с истинным человеколюбием, не с наигранным, а с подлинным интересом" [Ноэль, 1978]. Чтобы разговорить собеседника, можно сначала аналитику рассказать о себе, о работе, т. е. поговорить самому.

В свободном диалоге важно также выбрать правильный темп или ритм беседы: без больших пауз, т. к. эксперт может отвлечься, но и без гонки, иначе быстро утомляются оба участника, и нарастает напряженность, кроме того, некоторые люди говорят и думают очень медленно. Умение чередовать разные темпы, напряжение и разрядку в беседе существенно влияет на результат.

Подготовка к диалогу так же, как и к другим активным методам извлечения знаний, включает составление плана проведения сеанса извлечения, в котором необходимо предусмотреть следующие стадии:

1. Начало беседы (знакомство, создание у эксперта "образа" аналитика, объяснение целей и задач работы).
2. Диалог по извлечению знаний.
3. Заключительная стадия (благодарность эксперту, подведение итогов, договор о последующих встречах).

Девизом для инженера по знаниям могут послужить взгляды одного из классиков отечественного литературоведения М. М. Бахтина [Бахтин, 1975].

Диалог предполагает:

- уникальность каждого партнера и их принципиальное равенство друг другу;
- различие и оригинальность их точек зрения;
- ориентацию каждого на понимание и на активную интерпретацию его точки зрения партнером;
- ожидание ответа и его предвосхищение в собственном высказывании;
- взаимную дополнительность позиций участников общения, соотнесение которых и является целью диалога".

Активные групповые методы

К групповым методам извлечения знаний относятся:

- ролевые игры;
- дискуссии за "круглым столом" с участием нескольких экспертов;
- "мозговые штурмы".

Основное достоинство групповых методов — это возможность одновременного "поглощения" знаний от нескольких экспертов, взаимодействие которых вносит в этот процесс элемент принципиальной новизны от наложения разных взглядов и позиций.

Активные групповые методы обычно используются в качестве острой приправы при извлечении знаний, сами по себе они не могут служить источником более или менее полного знания. Их применяют как дополнительные к традиционным индивидуальным методам (наблюдения, интервью и т. д.), для активизации мышления и поведения экспертов.

Поскольку эти методы менее популярны, чем индивидуальные (что связано со сложностью организации), опишем их подробно.

"Круглый стол"

Метод круглого стола (термин заимствован из журналистики) предусматривает обсуждение какой-либо проблемы из выбранной предметной области, в котором принимают участие с равными правами несколько экспертов. Обычно вначале участники высказываются в определенном порядке, а затем переходят к живой свободной дискуссии. Число участников дискуссии колеблется от трех до пяти — семи.

Большинство общих рекомендаций по извлечению знаний, предложенных ранее, применимо и к данному методу. Однако существует и специфика, связанная с поведением человека в группе.

Во-первых, от инженера по знаниям подготовка "круглого стола" потребует дополнительных усилий: как организационных (место, время, обстановка, минеральная вода, чай, кворум и т. д.), так и психологических (умение вставлять уместные реплики, чувство юмора, память на имена и отчества, способность гасить конфликтные ситуации и т. д.).

Во-вторых, большинство участников будет говорить под воздействием "эффекта фасада" совсем не то, что они сказали бы в другой обстановке, т. е. желание произвести впечатление на других экспертов будет существенно "подсвечивать" их высказывания. Этот эффект часто наблюдается на защитах диссертаций. Члены ученого совета спрашивают обычно не то, что им действительно интересно, а то, что демонстрирует их собственную компетентность.

Ход беседы за круглым столом удобно записывать на магнитофон, а при расшифровке и анализе результатов учитывать этот эффект, а также взаимные отношения участников.

Задача дискуссии — коллективно, с разных точек зрения, под различными углами исследовать спорные гипотезы предметной области. Обычно эмпирические области богаты таким дискуссионным материалом. Для остроты на "круглый стол" приглашают представителей разных научных направлений и разных поколений, это также уменьшает опасность получения односторонних однобоких знаний.

Обмен мнениями по научным вопросам имеет давнюю традицию в истории человечества (античная Греция, Индия). До наших дней дошли литературные памятники обсуждения спорных вопросов (например, Протагор "Искусство спорить", работы софистов), послужившие первоосновой диалектики — науки вести беседу, спорить, развивать теорию. В самом слове дискуссия (от лат. *discussio* — исследование) содержится указание на то, что это метод научного познания, а не просто споры (для сравнения, полемика — от греч. *polemikos* — воинственный, враждебный).

Несколько практических советов по процедурным вопросам "круглого стола" из [Соколов, 1980]. Перед началом дискуссии ведущему полезно:

- убедиться, что все правильно понимают задачу (т. е. происходит сеанс извлечения знаний);
- установить регламент;
- четко сформулировать тему.

По ходу дискуссии важно проследить, чтобы слишком эмоциональные и разговорчивые эксперты не подменили тему, и чтобы критика позиций друг друга была обоснованной.

Научная плодотворность дискуссий делает этот метод привлекательным и для самих экспертов, особенно для тех, кто знает меньше. Это заметил еще Эпикур: "При философской дискуссии больше выигрывает побежденный — в том отношении, что он умножает знания".

"Мозговой штурм"

"Мозговой штурм" или "мозговая атака" — один из наиболее распространенных методов раскрепощения и активизации творческого мышления. Другие методы (метод фокальных объектов, синектика, метод контрольных вопросов [Шепотов, Шмаков, Крикун, 1985]) применяются гораздо реже из-за меньшей эффективности.

Впервые этот метод был использован в 1939 г. в США А. Осборном как способ получения новых идей в условиях запрещения критики. Замечено, что боязнь критики мешает творческому мышлению, поэтому основная идея штурма — это отделение процедуры генерирования идей в замкнутой группе специалистов от процесса анализа и оценки высказанных идей.

Как правило, штурм длится недолго (около 40 минут). Участникам (до 10 человек) предлагается высказывать любые идеи (шутливые, фантастические, ошибочные) на заданную тему (критика запрещена). Обычно высказывается более 50 идей. Регламент до 2 минут на выступление. Самый интересный момент штурма — это наступление пика (ажиотажа), когда идеи начинают "фонтанировать", т. е. происходит непроизвольная генерация гипотез участниками. Этот пик имеет теоретическое обоснование в работах выдающегося швейцарского психолога и психиатра З. Фрейда о бессознательном. При последующем анализе всего лишь 10—15% идей оказываются разумными, но среди них бывают весьма оригинальные. Оценивает результаты обычно группа экспертов, не участвовавшая в генерации.

Ведущий "мозгового штурма" — инженер по знаниям — должен свободно владеть аудиторией, подобрать активную группу экспертов — "генераторов", не зажимать плохие идеи — они могут служить катализаторами хороших. Искусство ведущего — это искусство задавать вопросы аудитории, "подогревая" генерацию. Вопросы служат "крючком" [Шепотов, Шмаков, Крикун, 1985], которым извлекаются идеи. Вопросы также могут останавливать многословных экспертов и служить способом развития идей других.

Основной девиз штурма — "чем больше идей, тем лучше". Фиксация хода сеанса — традиционная (протокол или магнитофон).

Экспертные игры

Игрой называют такой вид человеческой деятельности, который отражает (воссоздает) другие ее виды [Комаров, 1989]. При этом для игры характерны одновременно условность и серьезность.

Понятие экспертной игры или игры с экспертами в целях извлечения знаний восходит к трем источникам — это деловые игры, широко используемые при подготовке специалистов и моделировании [Борисова, Соловьева и др., 1988; Бурков, 1980; Комаров, 1989]; диагностические игры, описанные в [Алексеевская, Недоступ, 1988; Гельфанд, Розенфельд, Шифрин, 1988]; и компьютерные игры, все чаще применяемые в обучении [Пажитнов, 1987].

В настоящее время в психолого-педагогических науках нет развитой теоретической концепции деловых игр и других игровых методов обучения. Тем не менее на практике эти игры широко используются. Под *деловой* игрой чаще всего понимают эксперимент, где участникам предлагается производственная ситуация, а они на основе своего жизненного опыта, своих общих и специальных знаний и представлений принимают решения [Бурков, 1980]. Решения анализируются, и вскрываются закономерности мышления участников эксперимента. Именно эта анализирующая часть деловой игры полезна для получения знаний. И если участниками такой игры становятся эксперты, то игра из деловой превращается в экспертную. Из трех основных типов деловых игр (учебных, планово-производственных и исследовательских) к экспертам ближе всего исследовательские, которые используются для анализа систем, проверки правил принятия решений.

Диагностическая игра — эта та же деловая игра, но применяемая конкретно для диагностики методов принятия решения в медицине (диагностика методов диагностики). Эти игры возникли при исследовании способов передачи знаний от опытных врачей новичкам. В нашем понимании диагностическая игра — это игра, безусловно, экспертная без всяких оговорок, только с жестко закрепленной предметной областью — медициной.

Плодотворность моделирования реальных ситуаций в играх подтверждается сегодня практически во всех областях науки и техники. Они развивают логическое мышление, умение быстро принимать решения, вызывают интерес у экспертов.

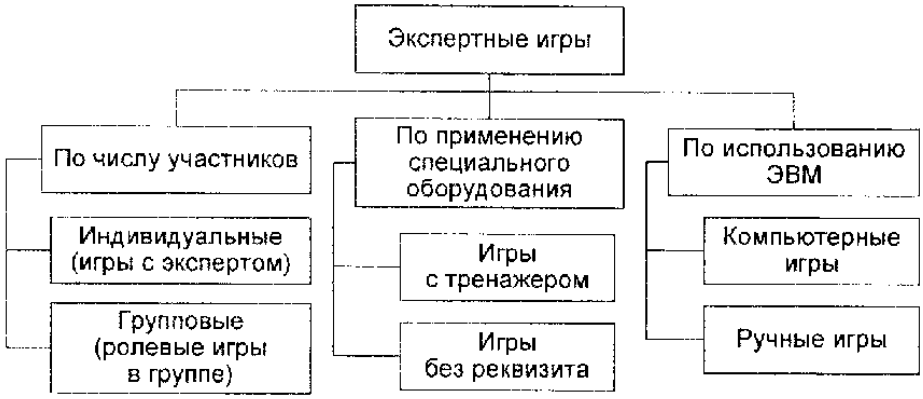

Рис. 2.20. Классификация экспертных игр

В соответствии с классификацией, введенной в *разд. 2.4,* будем разделять экспертные игры на :

- индивидуальные ;
- групповые.

Кроме того (рис. 2.20), продолжим и разовьем эту классификацию введением других критериев:

- использование специального оборудования;
- применение вычислительной техники.

Игры с экспертом

В этом случае с экспертом играет инженер по знаниям, который берет на себя какую-нибудь роль в моделируемой ситуации. Например, одному из авторов часто приходилось разыгрывать с экспертом игру "Учитель и ученик", в которой инженер по знаниям берет на себя роль ученика и на глазах у эксперта выполняет его работу (например, пишет психодиагностическое заключение), а эксперт поправляет ошибки "ученика". Эта игра — удобный способ разговорить застенчивого эксперта.

Другая игра (заимствована из [Гельфанд, Розенфельд, Шифрин, 1988]) заставляет инженера по знаниям взять на себя роль врача, который знает хорошо больного, а эксперт играет роль консультанта. Консультант задает вопросы и делает прогноз о целесообразности применения того или иного вида лечения (в описанной игре это был прогноз целесообразности электростимульной терапии при сердечной аритмии). Игра "двух врачей" позволила выявить, что эксперту понадобилось всего 30 вопросов для успешного прогноза, в то время как первоначальный вариант вопросника, составленный медиками для той же цели, содержал 170 вопросов.

Еще один пример игры. Сначала эксперта просят написать обоснование для собственного прогноза. Например, почему он считает, что язва у больного *X* заживает. Накапливается несколько таких обоснований [Гельфанд, Розенфельд, Шифрин, 1988], а через некоторое время эксперту зачитывают только его обоснование и просят сделать прогноз. Как правило, этого он сделать не может, т. е. обоснование (или его знания) было неполным. Эксперт дополняет обоснование, тем самым выявляются скрытые (для самого эксперта) пласты знаний. Так, в играх "Обоснование прогноза рецидива язвенного кровотечения" удалось выявить, что значимыми для прогноза являются всего три правила. Причем два правила входили в традиционно-диагностический вопросник, а третье было сформулировано во время игры.

Игра "Фокусировка на контексте": эксперт играет роль ЭС, а инженер по знаниям — роль пользователя. Разыгрывается ситуация консультации. Первые вопросы эксперта выявляют наиболее значимые понятия, самые важные аспекты проблемы. Роль пользователя может взять на себя и другой эксперт [Rabbits, Wright, 1987].

Основные советы инженеру по знаниям по проведению индивидуальных игр:

- играйте смело, весело, нешаблонно, придумывайте игры сами;
- не навязывайте игру эксперту, если он не расположен;
- в игре "не давите" на эксперта, не забывайте цели игры;
- не забывайте о времени и о том, что игра утомительна для эксперта.

Ролевые игры в группе

Групповые игры предусматривают участие в игре нескольких экспертов. К такой игре обычно заранее составляется сценарий, распределяются роли, к каждой роли готовится портрет-описание (лучше с девизом) и разрабатывается система оценивания игроков [Борисова, Соловьева и др., 1988].

Существует несколько способов проведения ролевых игр. В одних играх игроки могут придумать себе новые имена и играть под ними. В других все игроки переходят на "ты". В третьих роли выбирают игроки, в четвертых роли вытягивают по жребию. Роль — это комплекс образцов поведения. Роль связана с другими ролями. "Короля играет свита". Поскольку в нашем случае режиссером и сценаристом игры является инженер по знаниям, то ему и предоставляется полная свобода в выборе формы проведения игры.

Так, в [Лазарева, Пашинин, 1987] описана игра "План", предназначенная для извлечения знаний из специалистов предприятия, разрабатывающих производственные планы выпуска для цехов и принимающих различные решения по управлению производством.

В игре экспертов разбили на три игровые группы: $ЛПР_1$ — группа планирования; $ЛПР_2$ — группа менеджеров; Э — группа экспертизы по оцениванию действия $ЛПР_1$ и $ЛПР_2$. Группам $ЛПР_1$ и $ЛПР_2$ задавались различные производственные ситуации и тщательно протоколировались их споры, рассуждения, аргументы по принятии решений. В результате игры был создан прототип базы знаний экспертной системы планирования.

Обычно в игре принимает участие от трех до шести экспертов, если их больше, то можно разбить всех на несколько конкурирующих игровых бригад. Элемент состязательности оживляет игру. Например, чей диагноз окажется ближе к истинному, чей план рациональнее использует ресурсы, кто быстрее определит причину неисправности в техническом блоке.

Создание игровой обстановки потребует немало фантазии и творческой выдумки от инженера по знаниям. Ролевая игра, как правило, требует некоторых простейших заготовок (например, табличек "Директор", "Бухгалтерия", "Плановый отдел", специально напечатанных инструкцией с правилами игры). Но главное, конечно, чтобы эксперты в игре действительно "заиграли", раскрепостились и "раскрыли свои карты".

Игры с тренажерами

Игры с тренажерами в значительной степени ближе не к играм, а к имитационным упражнениям в ситуации, приближенной к действительности.

Наличие тренажера позволяет воссоздать почти производственную ситуацию и понаблюдать за экспертом. Тренажеры широко применяют для обучения (например, летчиков или операторов атомных станций). Очевидно, что применение тренажеров для извлечения знаний позволит зафиксировать фрагменты "летучих" знаний, возникающих во время и на месте реальных ситуаций и выпадающих из памяти при выходе за пределы ситуации.

Компьютерные экспертные игры

Идея использовать компьютеры в деловых играх известна давно. Но только когда компьютерные игры взяли в плен практически всех пользователей персональных ЭВМ от мала до велика, стала очевидной особая притягательность такого рода игр.

Приведем традиционную современную классификацию компьютерных игр из журнала GAME.EXE.

- Action/Arcade games (экшн/аркады). Игры-действия. Требуют хорошего глазомера и быстрой реакции.
- Simulation games (симуляторы). Базируются на моделировании реальной действительности и отработки практических навыков, например в вождении автомобиля, пассажирского самолета, поезда, авиадиспетчера и даже симуляторы рыбной ловли. Также популярны спортивные симуляторы — теннис, бокс и др.
- 3D Action games ("стрелялки"). То же, что и экшн, но с активным использованием трехмерной графики.
- Strategy games (стратегические игры). Требуют стратегического планирования и ответственности при принятии решений, например развитие цивилизаций, соперничество миров, экономическая борьба. Особый класс стратегических игр — wargames (военные игры). В последнее время упор в 3D Action делается на многопользовательский режим (игру по сети).
- Puzzles (настольные игры-головоломки). Компьютерные реализации различных логических игр.
- Adventure/Quest (приключенческие игры). Обычно обладают разветвленным сценарием, красивой графикой и звуком. Управляя одним или несколькими персонажами, игрок

должен правильно вести диалоги, разгадывать множество загадок и головоломок, замечать и правильно использовать предметы, спрятанные в игре.

- Role-playing games RPG (ролевые игры). Распространенный жанр, берущий свое начало в старых английских настольных играх. Существуют один или несколько персонажей, обладающих индивидуальными способностями и характеристиками. Им приходится сражаться с врагами, решать загадки. По мере выполнения этих задач, у героев накапливается опыт, и по достижению определенного значения, их характеристики улучшаются...

Следует отметить, что многие игры могут быть отнесены сразу к нескольким классам, и в целом эту классификацию нельзя считать строгой. Игры иногда полезны для развлечения экспертов перед сеансом извлечения знаний. Кроме того, очевидно, что экспертные игры, сочетая элементы перечисленных выше классов, могут успешно применяться для непосредственного извлечения знаний. Однако разработка и программная реализация такой игры потребуют существенных вложений временных и денежных ресурсов.

Одна из первых отечественных экспертных компьютерных игр описана в [Гинкул, 1989]. Основной принцип игры "Зоосад" состоит в создании игровой ситуации при организации диалога с экспертом. При этом задача извлечения знаний маскируется нацеленностью на решение чисто игровой задачи: необходимо определить содержимое "черного ящика", в котором находится некое животное, при этом надо набрать наибольшее количество очков, не истратив выделенного ресурса денег. В ходе игры эксперт делает ставки на различные гипотезы, указывая при этом, какими признаками обладает то или иное животное. По ходу игры невидимо для эксперта формируются правила, отражающие знания эксперта на основании сделанных им ходов. В данной игре — это знания о том, какими признаками обладают те или иные животные. Таким образом, выявляется алфавит значимых признаков для диагностики и классификации животных.

2.6. Методы структурирования и формализации

Разделение стадий извлечения и структурирования знаний является весьма условным, поскольку хороший инженер по знаниям, уже извлекая знания, начинает работу по структурированию и формированию поля знаний.

Однако следует отметить, что и теоретический, и практический арсенал методов структурирования на сегодня проработан явно недостаточно.

Необходимость разработки теоретических основ науки о методах разработки систем, основанных на знаниях — инженерии знаний — обосновывается в работах Д. А. Поспелова, Э. В. Попова, В. Л. Стефанюка, Р. Шенка, М. Минского — ведущих специалистов в области ИИ в России и за рубежом. Первые шаги в создании методологии (работы Г. С. Осипова, В. Ф. Хорошевского, А. М. Яшина, B. Wielinga и др.) фактически являются пионерскими и чаще всего ориентированы на определенный класс задач, моделируемых в рамках конкретного программного инструментария.

В данном разделе предлагается один из новых подходов [Гаврилова, 1995], позволяющий провести стадию структурирования независимо от последующей программной реализации, опираясь на достижения в области разработки сложных систем.

2.6.1. Теоретические предпосылки

Стадия концептуального анализа или структурирования знаний традиционно является (наряду со стадией извлечения) "узким местом" в жизненном цикле разработки интеллектуальных систем [Adeli, 1994J. Методология структурирования близка к современной теории больших систем [Гиг, 1981] или сложных систем [Courtois, 1985], где традиционно акцент делается на процессе проектирования [Bertalanffy, 1950; Boulding, 1956]. Большой вклад в эту теорию внесли классики объектно-ориентированного анализа [Буч, 1992].

Разработку интеллектуальных систем с уверенностью можно отнести к данному классу задач, поскольку они обладают основными признаками сложности (иерархия понятий, внутриэлементные и межэлементные связи и пр.). Аналогичные концепции, но связанные не с общесистемными исследованиями, а рассматривающие информационные процессы в системах, таких как связь и управление, положили начало кибернетике как самостоятельной науке [Винер, 1958; Эшби, 1959]. Позднее, в 1960-х гг. были получены новые результаты по развитию математической теории систем высокого уровня общности [Месарович, Такахара, 1978]. Существенный вклад в теорию систем и основы структурирования внесли отечественные исследователи [Моисеев, 1981; Глушков, 1964; Ивахненко, 1971; Поспелов, 1986] и др.

Системный анализ тесно переплетается с теорией систем и включает совокупность методов, ориентированных на исследование и моделирование сложных систем — технических, экономических, экологических и т. п.

Проектирование сложных систем и методы структурирования информации традиционно использовали иерархический подход [Месарович, Такахара, 1978] как методологический прием расчленения формально описанной системы на уровни (или блоки, или модули). На высших уровнях иерархии используются наименее детализованные представления, отражающие только самые общие черты и особенности проектируемой системы. На следующих уровнях степень подробности возрастает, при этом система рассматривается не в целом, а отдельными блоками.

На каждом уровне вводятся свои представления о системе и элементах. Элемент *k*-го уровня является системой для *(k* — 1) уровня. Продвижение от уровня к уровню имеет строгую направленность, определяемую стратегией проектирования — дедуктивную нисходящую "сверху вниз" (top-down) или индуктивную восходящую "снизу вверх" (bottom-up).

Предлагаемый ниже объектно-структурный подход позволяет объединить две, обычно противопоставляемые, стратегии проектирования. Синтез этих стратегий, а также включение возможности итеративных возвратов на предыдущие уровни обобщений позволили создать дуальную концепцию, предоставляющую аналитику широкие возможности на стадии структурирования знаний как для формирования концептуальной структуры предметной области *Sk,* так и для функциональной структуры *Sf.*

Рис. 2.21 иллюстрирует дуальную концепцию при проектировании *Sf* для ЭС помощи оператору энергетического блока

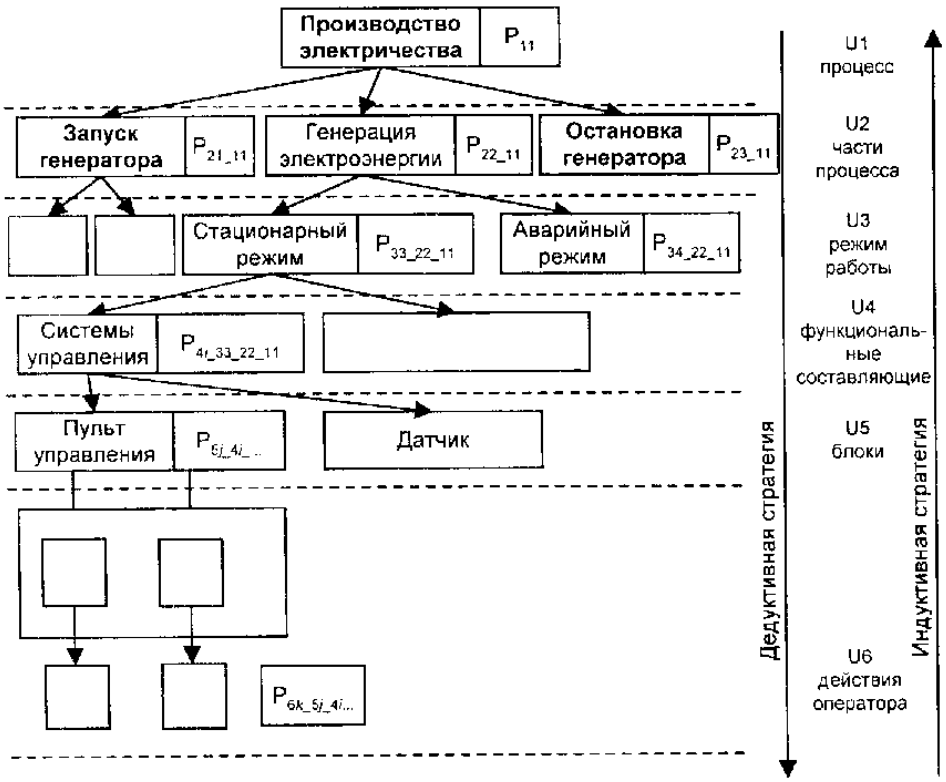

Рис. 2.21. Дуальная стратегия проектирования

Нисходящая концепция (top-down) декларирует движение от $n \Rightarrow n + 1$, где n — *n-й* уровень иерархии понятий ПО (предметной области) с последующей детализацией понятий, принадлежащих соответствующим уровням.

$$\mathrm{STR}td : P_i^n, \ldots, P_i^{n+1} \Rightarrow P_{ki}^{n+1}$$

где:

- n — номер уровня порождающего концепта;
- i — номер порождающего концепта;
- k_i — число порождающих концептов, сумма всех k_i по i составляет общее число концептов на уровне $n + 1$.

Восходящая концепция (bottom-up) предписывает движение $n => n — 1$ с последовательным обобщением понятий.

$$STR_{bu} : P_i^n, \ldots, P_{ki}^{n+1} => P_i^{n-1}$$

где:

- n — номер уровня порождающих концептов;
- i — номер порождаемого концепта;
- k_i — число порождающих концептов, сумма всех k_i по i составляет общее число концептов на уровне n.

Основанием для прекращения агрегирования и дезагрегирования является полное использование словаря терминов, которым пользуется эксперт, при этом число уровней является значимым фактором успешности структурирования.

В целом существующие подходы к проектированию сложных систем можно разделить на два больших класса:

- *структурный (системный) подход* или анализ, основанный на идее алгоритмической декомпозиции, где каждый модуль системы выполняет один из важнейших этапов общего процесса;
- *объектный подход,* связанный с декомпозицией и выделением не процессов, а объектов, при этом каждый объект рассматривается как экземпляр определенного класса.

В структурном анализе [Yourdon, 1989] разработано большое число выразительных средств для проектирования, в том числе графических [Буч, 1991]:

- диаграммы потоков данных (DFD, data-flow diagrams), структурированные словари (тезаурусы), языки спецификации систем, таблицы решений;
- стрелочные диаграммы "объект-связь" (ERD, entity-relationship diagrams), диаграммы переходов (состояний);
- деревья целей;
- блок-схемы алгоритмов (в нотации Насси— Шнейдермана, Фестля и др.);
- средства управления проектом (PERT-диаграммы, диаграммы Ганта и др.).

Множественность средств и их некоторая избыточность объясняется тем, что каждая предметная область, используя структурный подход как универсальное средство моделирования, вводила свою терминологию, наиболее подходящую для отражения специфики конкретной проблемы. Поскольку инженерия знаний имеет дело с широким классом ПО (это "мягкие" ПО), встает задача разработки достаточно универсального языка структурирования.

Объектный (объектно-ориентированный) подход (ООП) был разработан в 1979 году [Jones, 1979], а затем развит в работах [Peters, 1981; Shaw, 1984; Буч, 1993].

ООП, возникший как технология программирования больших программных продуктов, базируется на основных элементарных понятиях [Буч, 1993]:

- объекты и классы как объекты, связанные общностью структуры и свойств;
- классификации как средства упорядочения знаний;
- иерархии с наследованием свойств;
- инкапсуляции как средства ограничения доступа;
- методы и полиморфизм для определения функций и отношений.

ООП имеет систему условных обозначений и предлагает набор моделей для проектирования сложных систем. Широкое распространение объектно-ориентированных языков программирования C++, CLOS, Smalltalk и др. успешно демонстрируют жизнеспособность и перспективность этого подхода. В последнее время этот подход успешно применяется и в CASE-средствах не только для проектирования программ, но и для моделирования бизнес-процессов. Особенную популярность приобретает язык UML (Unified Modelling Language) [Буч, Рамбо, Джекобсон, 2000], и программный инструментарий на нем основанный, например Rational Rosex [Боггс, 2001].

2.6.2. Объектно-структурный подход (ОСП)

В качестве базисной парадигмы структурного анализа знаний и формирования поля знаний P_Z можно предложить обобщенный объектно-структурный подход (ОСП) [Гаврилова, 1995].

Основные постулаты этой парадигмы заимствованы из ООП и расширены:

1.Системность (взаимосвязь между понятиями).

2.Абстрагирование (выявление существенных характеристик понятия, которые отличают его от других).

3.Иерархия (ранжирование на упорядоченные системы абстракций).

4.Типизация (выделение классов понятий с частичным наследованием свойств в подклассах).

5.Модульность (разбиение задачи на подзадачи или "возможные миры").

6.Наглядность и простота нотации.

Использование пятого постулата ОСП в инженерии знаний позволяет строить глобальные БЗ с возможностью выделить локальные задачи с помощью горизонтальных и вертикальных сечений *(см. ниже)* на отдельные модули пространства-описания предметной области.

Шестой постулат внесен в список последним, но не по значимости. В инженерии знаний формирование *PZ* традиционно является критической точкой [Гаврилова, Червинская, Яшин, 1988; Гаврилова, Червинская, 1992], т. к. создаваемая неформальная модель предметной области должна быть предельно ясной и лаконичной. Традиционно языком инженерии знаний были диаграммы, таблицы и другие графические элементы, способствующие наглядности представлений. Именно поэтому предлагаемый подход к структурированию связан с возможной визуализацией процесса проектирования.

ОСП позволяет наглядно и компактно отобразить объекты и отношения предметной области на основе использования шести постулатов.

Объектно-структурный подход подразумевает интегрированное использование сформулированных выше постулатов от первой до последней стадий разработки БЗ интеллектуальных и обучающих систем. На основе ОСП предлагается алгоритм *объектно-структурного анализа* (ОСА) предметной области, позволяющего оптимизировать и упорядочить достаточно размытые процедуры структурирования знаний.

Стратификация знаний

Основы ОСА были предложены автором еще в работах [Гаврилова, 1989; Гаврилова, Красовская, 1990], и успешно применялись при разработке ЭС МИКРОЛЮШЕР [Гаврилова, Тишкин, Золотарев, 1989] и АВЭКС [Гаврилова, Минкова, Карапетян,1992].

ОСА подразумевает дезагрегацию ПО, как правило, на восемь страт или слоев (табл. 2.1 и 2.2).

Таблица 2.1. *Стратификация знаний предметной области*

Страта	Вид знаний страты	Уровни страты
s_2	ЗАЧЕМ-знания	Стратегический анализ: назначение и функции системы
s_2	КТО-знания	Организационный анализ: коллектив разработчиков системы
s 3	ЧТО-знания	Концептуальный анализ: основные концепты, понятийная структура
s_4	КАК-знания	Функциональный анализ: гипотезы и модели принятия решения
s_5	ГДЕ-знания	Пространственный анализ: окружение, оборудование, коммуникации
s_6	КОГДА-знания	Временной анализ: временные параметры и ограничения
s_7	ПОЧЕМУ-знания	Каузальный или причинно-следственный анализ: формирование подсистемы объяснений
s 8	СКОЛЬКО-знания	Экономический анализ: ресурсы, затраты, прибыль, окупаемость

Объектно-структурный анализ подразумевает разработку и использование матрицы ОСА (табл. 2.2), которая позволяет всю собранную информацию дезагрегировать последовательно по слоям-стратам (вертикальный анализ), а затем по уровням — от уровня проблемы до уровня подзадачи (горизонтальный анализ). Или наоборот — сначала по уровням, а потом по стратам.

Таблица 2.2. *Матрица объектно-структурного анализа*

Уровни страты	**Уровень области u_1**	**Уровень проблемы u_2**	**Уровень задачи u_3**	**Уровень подзадачи u_4**	**...**	**u_n**
Стратегический анализ s1	E_{11}	E_{12}	E_{13}	E_{14}		E_{1n}
Организационный анализ s_2	E_{21}					
Концептуальный анализ s_3	E_{31}					
Функциональный анализ s_4	E_{41}					
Пространственный анализ s_5	E_{51}					
Временной анализ s_6	E_{61}					
Каузальный анализ s_7	E_{71}					
Экономический анализ s_8	E_{81}					
……					E_{ij}	
s_m	E_{m1}					E_{mn}

При необходимости число страт может быть увеличено. В свою очередь знания каждой страты подвергаются дальнейшему ОСА и декомпозируются на составляющие $\|e_{mn}\|$, где m — номер уровня, n — номер страты, а e_{mn} принадлежит множеству K всех концептов (понятий) предметной области.

Алгоритм ОСА (объектно-структурного анализа)

Алгоритм ОСА предназначен для детального практического структурирования знаний предметной области (ПО). В основе ОСА заложен алгоритм заполнения ОСА-матрицы E_{mn}. Алгоритм содержит последовательность аналитических процедур, позволяющих упростить и оптимизировать процесс структурирования. Алгоритм разделяется на две составляющие:

- A_I: Глобальный (вертикальный) анализ, включающий разбиение ПО на методологические страты (Что-знания, Как-знания и т. д.) на уровне всей ПО. В результате заполняется первый столбец матрицы.
- A_II: Анализ страт (горизонтальный), включающий построение многоуровневых структур по отдельным стратам. Число уровней n определяется особенностями стратифицированных знаний ПО и может существенно отличаться для разных страт. С точки зрения методологии n<3 свидетельствует о слабой проработке ПО.

Первый уровень соответствует уровню всей ПО (предметной области). Второй — уровню проблемы, выделенной для решения. Третий — уровню конкретной решаемой задачи. Дальнейшие соответствуют подзадачам, если имеет смысл их выделять.

При этом возможно как последовательное применение восходящей (bottom-up) и нисходящей концепции (top-down), так и их одновременное применение.

Глобальный анализ

Технология глобального анализа сводится к разбиению пространства основной задачи структурирования ПО на подзадачи, соответствующие особенностям ПО. Для разработки интеллектуальных систем существует минимальный набор *s*-страт, обеспечивающий формирование БЗ. Минимальный набор включает три страты:

- s3, — формирование концептуальной структуры *Sk*;
- s4 — формирование функциональной структуры *Sf,*
- s5 — формирование подсистемы объяснений S_0.

Формирование остальных страт позволяет существенно оптимизировать процесс разработки и избежать многих традиционных ошибок проектирования. Страты s4 и s5 являются дополнительными и формируются в случаях, когда знания предметной области существенно зависят от временных и пространственных параметров (системы реального времени, планирование действий роботов и т. п.).

Алгоритм A__1 глобального анализа может быть кратко сформулирован следующим образом:

- A_1__1: Собрать все материалы, полученные по результатам извлечения знаний.
- A_1_2: Выбрать набор страт *N,* подлежащих формированию (Nmin= 3).
- A_1__3: Отобрать всю информацию по первой выбранной страте (*i*= 1, где *i* — номер из выбранного набора страт *N).*
- A_1_4: Повторить шаг A_1_3 для *i* + 1 для всех выбранных страт до *i* <= *N.*
- A_1_5: Если часть информации останется неиспользованной, увеличить число страт и повторить для новых страт шаг A_1_3; иначе перейти к последовательной реализации алгоритмов горизонтального анализа страт A_2.

Анализ страт

Последовательность шагов горизонтального анализа зависит от номера страты, но фактически сводится к реализации дуальной концепции структурирования для решения конкретной подзадачи.

Ниже предлагается алгоритм ОСА для одной из обязательных страт s_3, (ЧТО-анализ), результатом которого является формирование концептуальной структуры предметной области S_k.

- A_2_3_1: Из группы информации, соответствующей ЧТО-страте, выбрать все значимые понятия и сформулировать соответствующие концепты.
- A_2_3_2: Выявить имеющиеся иерархии и зафиксировать их графически в виде структуры.
- A_2_3_3: Детализировать концепты, пользуясь нисходящей концепцией (top-down).
- A_2__3__4: Образовать метапонятия по восходящей концепции (bottom-up).
- A_2_3_5: Исключить повторы, избыточность и синонимию.
- A_2_3_6: Обсудить понятия, не вошедшие в структуру *Sf,* с экспертом и перенести их в другие страты или исключить.
- A_2_3_7: Полученный граф или набор графов разделить на уровни и обозначить согласно матрице ОСА.

2.6.3. Практические методы структурирования

Методы извлечения знаний, рассмотренные выше, являются непосредственной подготовкой к структурированию знаний. Данный раздел посвящен изучению практических методов структурирования знаний.

Алгоритм для "чайников"

В качестве простейшего прагматического подхода к формированию поля знаний начинающему инженеру по знаниям можно предложить следующий алгоритм для "чайников" (рис. 2.22):

1.Определение входных *{X}* и выходных *{Y* данных. Этот шаг совершенно необходим, т. к. он определяет направление движения в поле знаний —от *X* к *Y.* Кроме того, структура входных и выходных данных существенно влияет на форму и содержание поля знаний. На этом шаге определение может быть достаточно размытым, в дальнейшем оно будет уточняться.

2.Составление словаря терминов и наборов ключевых слов *N.* На этом шаге проводится текстуальный анализ всех протоколов сеансов извлечения знаний и выписываются все значимые слова, обозначающие понятия, явления, процессы, предметы, действия, признаки и т. п. При этом следует попытаться разобраться в значении терминов. Важен осмысленный словарь.

3.Выявление объектов и понятий *{A}.* Производится "просеивание" словаря *N* и выбор значимых для принятия решения понятий и их признаков. В идеале на этом шаге образуется полный систематический набор терминов из какой-либо области знаний.

Рис. 2.22. Стадии структурирования знаний — алгоритм для "чайников"

4.Построение пирамиды знаний. Под пирамидой знаний мы понимаем иерархическую лестницу понятий, подъем по которой означает углубление понимания и повышения уровня абстракции (обобщенности) понятий. Количество уровней в пирамиде зависит от особенностей предметной области, профессионализма экспертов и инженеров по знаниям.

5.Определение отношений *{RA}*. Отношения между понятиями выявляются как внутри каждого из уровней пирамиды, так и между уровнями. Фактически на этом шаге даются имена тем связям, которые обнаруживаются на шагах 4 и 5, а также обозначаются причинно-следственные, лингвистические, временные и другие виды отношений.

6.Определение стратегии принятия решения *(Sf)*. Определение стратегии принятия решения, т. е. выявление цепочек рассуждений, связывает все сформированные ранее понятия и отношения в динамическую систему поля знаний. Именно стратегии придают активность знаниям, именно они "перетряхивают" модель *M* в поиске от *X* к *Y*.

7.Завершающее структурирование поля. Подразумевает упорядочивание полученной структуры, удаление дублирующих или лишних деталей, корректировку и уточнение всех конструкций.

Однако на практике при использовании данного алгоритма можно столкнуться с непредвиденными трудностями, связанными с ошибками на стадии извлечения знаний и с особенностями знаний различных предметных областей. Тогда возможно привлечение других, более "прицельных" методов структурирования. При этом на разных этапах схемы (см. рис. 2.22) возможно использование различных методик.

При этом, естественно, для таких простых и очевидных шагов, как определение входных и выходных понятий или составление словаря терминов, никаких искусственных методов предлагаться не будет.

Методы выявления объектов, понятий и их атрибутов

Понятие или концепт — это обобщение предметов некоторого класса по их специфическим признакам. Обобщенность является сквозной характеристикой всех когнитивных психических структур, начиная с простейших сенсорных образов.

Так, понятие "автомобиль" объединяет множество различных предметов, но все они имеют четыре колеса, двигатель и массу других деталей, позволяющих перевозить на них грузы и людей. Существует ряд методов выявления понятий предметной области в общем словаре терминов, который составлен на основании сеансов извлечения знаний. При этом важно выявление не только самих понятий, но и их признаков.

Возвращаясь к терминологии, введенной в *разд. 1.1,* на этом этапе определяются также интенсионалы и экстенсионалы понятий предметной области. Если задача выделения реальных объектов *A* связана только с наблюдательностью и лингвистическими способностями эксперта и инженера по знаниям, то определение метапонятий *B* требует от них умения проводить операции обобщения и классификации, которые никогда не считались тривиальными.

Д. А. Поспелов [Поспелов, 1986] предложил ряд подходов к созданию основ теории обобщения и классификации применительно к ситуационному управлению и искусственному интеллекту в целом, а также выделил ряд особенностей задач формирования понятий. Среди них особое место занимает выявление прагматически значимых признаков для формирования понятий, способствующих решению задачи.

Сложность заключается в том, что для многих понятий практически невозможно однозначно определить их признаки, это связано с различными формами репрезентации понятий в памяти человека.

Все методы выявления понятий мы разделили на :

- традиционные, основанные на математическом аппарате распознавания образов и классификации;
- нетрадиционные, основанные на методологии инженерии знаний.

Если первые достаточно хорошо освещены в литературе, то вторые пока менее известны.

Интересный эксперимент по выявлению понятий описан в [Кук, Макдональд, 1986]. Тридцати студентам, имеющим права на вождение автомобиля, предложили составить словарь терминов предметной области с помощью
четырех методов:

- формирования перечня понятий (17%);
- интервьюирования специалистов (35%);
- составление списка элементарных действий (18%);
- составления оглавления учебника (30%).

Цифры в скобках характеризуют продуктивность соответствующего метода, т. е. показывают, какой процент понятий из общего выявленного списка (702 термина) был получен соответствующим методом. Для классификации понятий были привлечены еще два участника эксперимента, которые разделили 702 выявленных понятия на семь категорий (методом сортировки карточек). Табл. 2.3 отражает численные данные концептуализации.

Таблица 2.3. *Данные концептуализации*

Категории терминов	**Процент от общего числа терминов**	**Процент от общего числа терминов, полученный соответствующим методом**			
		Перечень понятий	**Интервьюи-рование**	**Список операций**	**Составление оглавления**
Объяснение	6	5,5	7,2	7,0	4,9
Общие правила	22,0	43,6	18,9	36,8	4,9
Режимные правила	9,0	9,8	8,4	11,6	6,6
Понятия	42,0	18,4	38,9	8,5	77,7
Процедуры	9,0	5,1	9,5	25,6	1,2
Факты	9,0	15,0	12,5	8,9	1,2
Прочие понятия	3,0	2,6	4,6	1,6	3,5

В целом результаты показали, что для выявления непосредственно концептов наиболее результативными оказались методы интервьюирования и составления оглавления учебника. Однако наибольшее число общих правил было порождено в методе списка действий. Таким образом, еще раз подтвердилось утверждение о том, что нет "лучшего" метода, есть методы, подходящие для тех или иных ситуаций и типов знаний.

Интересно, что число правил — продукций "если — то" составило небольшой процент во всех четырех методах. Это говорит о том, что популярная продукционная модель вряд ли является естественной для человеческих моделей репрезентации знаний.

Методы выявления связей между понятиями

Концепты не существуют независимо, они включены в общую понятийную структуру с помощью отношений. Выявление связей между понятиями при разработке баз знаний доставляет инженеру по знаниям немало проблем. То, что знания в памяти — это некоторые связные структуры, а не отдельные фрагменты, общеизвестно и очевидно. Тем не менее основной упор в существующих моделях представления знаний делается на понятия, а связи вводят весьма примитивные (в основном причинно-следственные).

В работах по теории ИИ все больше внимания уделяется взаимосвязанности структур знаний. Так, в [Шенк, Бирнбаум, Мей, 1989] введено понятие сценария (script) как некоторой структуры представления знаний. Основу сценария составляет КОП (Концептуальная Организация Памяти) и мета-КОПЫ — некоторые обобщающие структуры.

Сценарии, в свою очередь, делятся на фрагменты — или сцены (chunks). Связи между фрагментами — временные или пространственные, внутри фрагмента — самые различные: ситуативные, ассоциативные, функциональные и т. д.

Все методы выявления таких связей можно разделить на две группы:

- формальные;
- неформальные (основаны на дополнительной работе с экспертом).

Неформальные методы выявления связей придумывает инженер по знаниям для того, чтобы вынудить эксперта указать явные и неявные связи между понятиями. Наиболее распространенным является метод "сортировка карточек" в группы [Волков, Ломнев, 1989; Rabbits, Wright, 1987], широко применяемый и для формирования понятий. Другим неформальным методом является построение замкнутых кривых. В этом случае эксперта просят обвести замкнутой кривой связанные друг с другом понятия [Olson, Reuter, 1987]. Этот метод может быть реализован как на бумаге, так и на экране дисплея. В данном случае можно говорить о привлечении элементов когнитивной графики [Зенкин, 1991].

После того как определены связи между понятиями, все понятия как бы распадаются на группы метапонятий, присвоение имен которым происходит на следующей стадии процесса структурирования.

Методы выделения метапонятий
и детализация понятий (пирамида знаний)

Процесс образования метапонятий, т. е. интерпретации групп понятий, полученных на предыдущей стадии, как и обратная процедура — детализация (разукрупнение) понятий, — видимо, принципиально не поддающиеся формализации операции. Они требуют высокой квалификации экспертов, а также наличия способностей к "наклеиванию" лингвистических ярлыков. Если на рис. 2.23 показаны схемы обобщения и детализации на тривиальных примерах, то в реальных предметных областях эта задача оказывается весьма трудоемкой. При этом независимо от того, формальными или неформальными методами были выявлены понятия или детали понятий, присвоение имен которым или интерпретация их — всегда неформальный процесс, в котором инженер по знаниям просит эксперта дать название некоторой группе понятий или отдельных признаков.

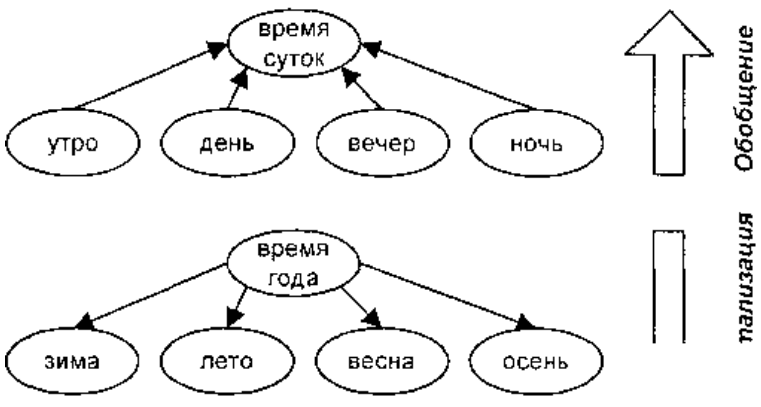

Рис. 2.23. Обобщение и детализация понятий

Это не всегда удается. Так, в системе АВТАНТЕСТ [Гаврилова, 1984] при образовании метапонятий, полученных методами кластерного анализа, интерпретация заняла несколько месяцев. Это связано с тем, что формальные методы иногда выделяют "искусственные" концепты, в то время как неформальные обычно — практически используемые и потому легко узнаваемые понятия.

Методы построения пирамиды знаний обязательно включают использование наглядного материала — рисунков, схем, кубиков. Уровни пирамиды чаще возникают в сознании инженера по знаниям именно как некоторые образы.

Построение пирамиды знаний может быть основано и на естественной иерархии предметной области, например связанной с организационной структурой предприятия или с уровнем компетентности специалистов (рис. 2.24).

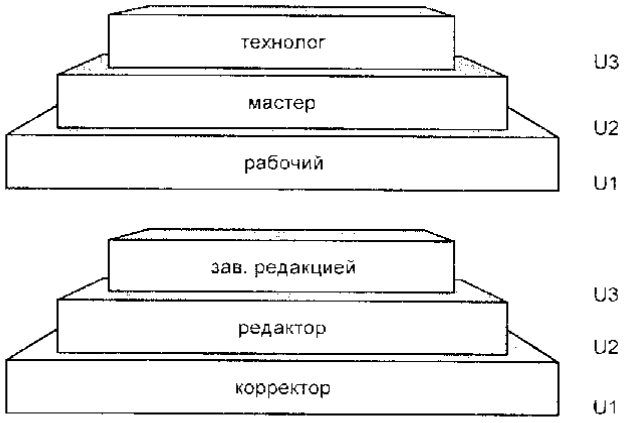

Рис. 2.24. Пирамиды знаний

Методы определения отношений

Если на стадии 4 (см. рис. 2.22) мы выявили связи между понятиями и использовали их на стадиях 5 и 6 для получения пирамиды знаний, то на стадии 7 мы даем имена связям, т. е. превращаем их в отношения.

В [Поспелов, 1986] указывается на наличие более 200 базовых видов различных отношений, существующих между понятиями. Предложены различные классификации отношений [Келасьев, 1984; Поспелов, 1986]. Следует только подчеркнуть, что помимо универсальных отношений (пространственных, временных, причинно-следственных) существуют еще и специфические отношения, присущие той или иной предметной области [Гаврилова, Червинская, Яшин, 1988].

Интересные возможности к структурированию знаний добавляют системы когнитивной графики. Так, в системе OPAL [Olton, Musen, Combs et al., 1987] эксперт может манипулировать на экране дисплея изображениями простейших понятий и строить схемы лечения заболеваний, обозначая отношения явными линиями, которые затем именуются.

Скудность методов структурирования объясняется тем, что методологическая база инженерии знаний только закладывается, а большинство инженеров по знаниям проводит концептуализацию, руководствуясь наиболее дорогими и неэффективными способами — ad hoc (применительно к случаю), или "по наитию", т. е. исходя из соображений здравого смысла.

Визуальное структурирование

Визуальные методы спецификации и проектирования баз знаний и разработка концептуальных структур являются достаточно эффективным инструментом познания [Jonassen, 1993]. Использование методов инженерии знаний в качестве дидактических инструментов и формализмов представления знаний способствует более быстрому и более полному *пониманию* структуры знаний данной предметной области, что особенно ценно для новичков на стадии изучения особенностей профессиональной деятельности.

В *разд.* 2.7.7 было предложено определение поля знаний, которое позволяет инженеру по знаниям трактовать форму представления поля достаточно широко, в частности семантические сети или понятийные карты (concept maps) являются возможной формой представления. Это означает, что сам процесс построения семантических сетей помогает осознавать познавательные структуры.

Программы визуализации являются инструментом, позволяющим сделать видимыми семантические сети памяти человека. Сети состоят из узлов и упорядоченных соотношений или связей, соединяющих эти узлы. Узлы выражают понятия или предположения, а связи описывают взаимоотношения между этими узлами (рис. 2.25). Поэтому разработка **семантических сетей** подразумевает анализ структурных взаимодействии между отдельными понятиями предметной области.

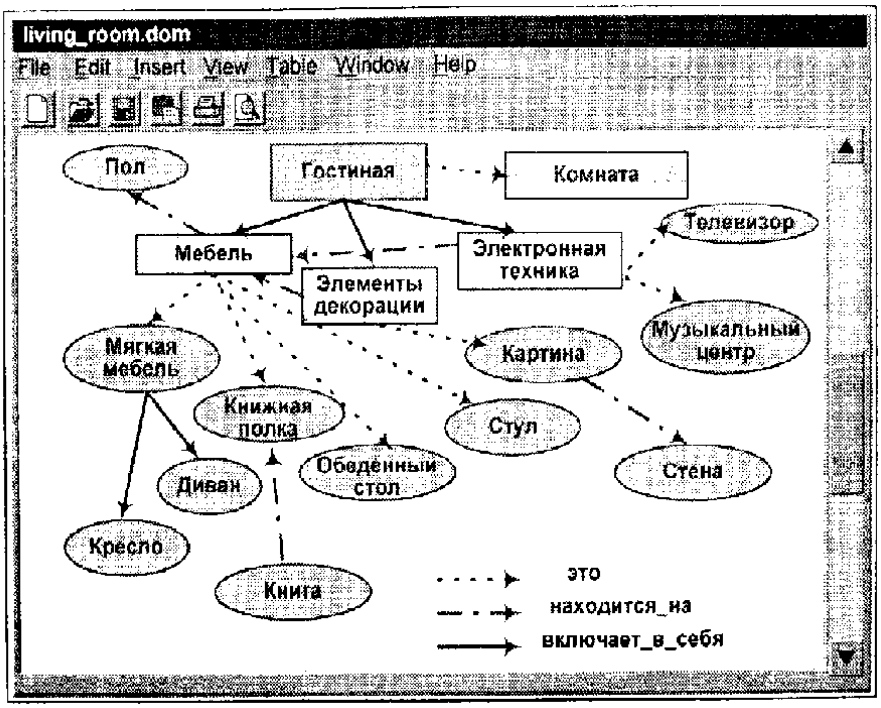

Рис. 2.25. Пример семантической сети

Нами разработано несколько версий АРМа (Автоматизированное Рабочее Место) инженера по знаниям KEW (Knowledge Engineering Workbench) [Гаврилова, 1995, Гаврилова, Воинов, 1995—1997], которые наряду с такими программами, как SemNet [Fisher, 1992], Learning Tool [Kozma, 1987], TextVision [Kommers, 1989] или Inspiration, дают возможность ученикам, экспертам или аналитикам связать между собой изучаемые ими понятия в многомерные сети представлений и описать природу связей между всеми входящими в сеть понятиями.

Одна из версий KEW, созданная совместно с А. В. Воиновым, получила первую премию на выставке программных систем IV Национальной конференции по искусственному интеллекту в 1994 году в разделе программных инструментариев разработки интеллектуальных систем. KEW демонстрирует жизнеспособность технологии автоматизированного проектирования интел6лектуальных систем (АПРИС) или CAKE (Computer Aided Knowledge Engineering), впервые описанной в работе [Гаврилова, 1992]. Последняя версия CAKE-2 создана Т. Е. Гелеверей и успешно применяется на практике (www.csa.ru/ailab).

KEW предназначен для интеллектуальной поддержки деятельности инженера по знаниям на протяжении всего жизненного цикла разработки экспертной системы, включая стадии — идентификации проблемы, получения знаний, структурирования знаний, формализации, программной реализации, тестирования.

Центральным блоком KEW является графический структуризатор знаний, который поддерживает последовательную графическую реализацию ОСА *(см. разд. 2.6.2)* и автоматическую компиляцию БЗ из графической спецификации.

Интерфейс KEW состоит из трех основных частей (рис. 2.26):

- панель концептуальной структуры;
- панель гипертекста;
- панель функциональной структуры.

Панель концептуальной структуры предназначена для графического структурирования знаний. Она позволяет определить понятия и обозначить связи между ними в форме концептуальной структуры *Sk*

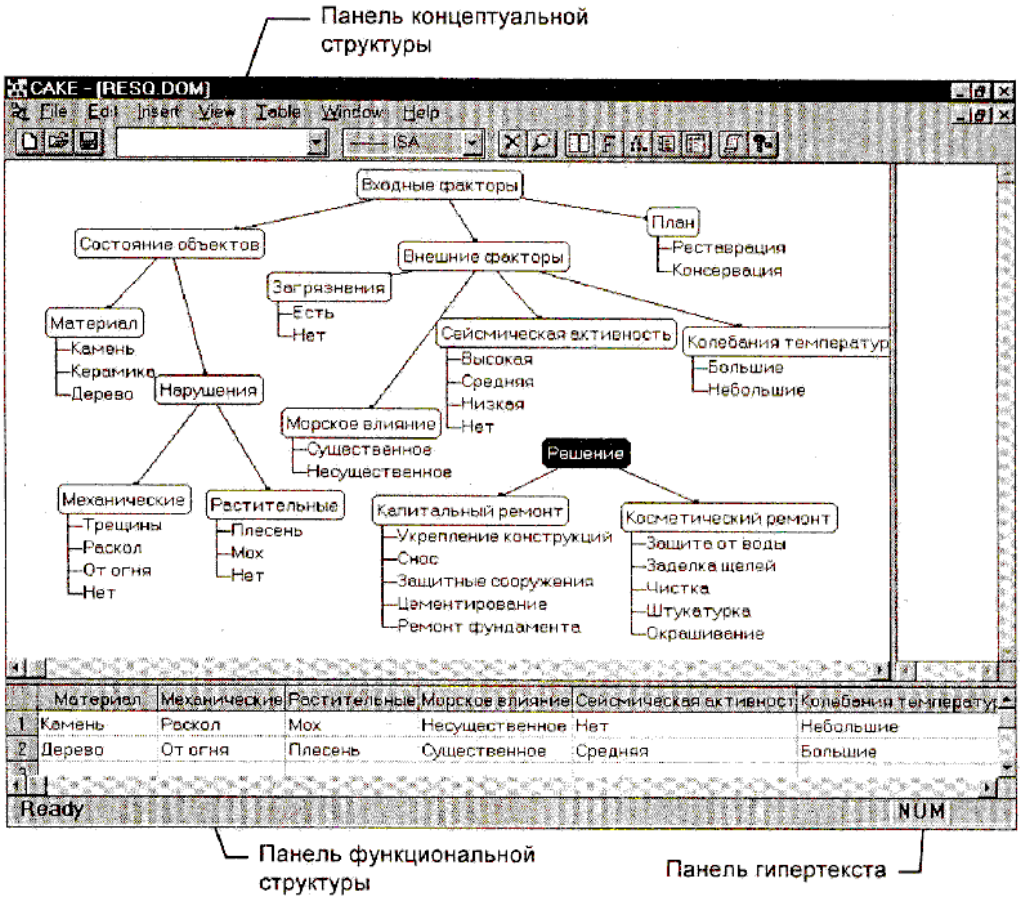

Рис. 2.26. Интерфейс АРМ инженера по знаниям.

В панель гипертекста можно поместить любой комментарий, связанный с объектом, определенным на панели концептуальной структуры понятий.

Основное назначение панели функциональной структуры S_f — представить наглядно в форме строк таблицы причинно-следственные и другие функциональные взаимосвязи между понятиями концептуальной структуры, на основании которых эксперт принимает решения. Столбцы таблицы формируются простейшей операцией drag-and-drop из понятий на панели концептуальной структуры.

После того как модели S_k и S_f созданы, KEW автоматически компилирует базу знаний на Прологе из созданной графической спецификации и моделирует работу экспертной системы. Это удобно для быстрого наглядного прототипирования ЭС и для отладки БЗ совместно с экспертом.

ЧАСТЬ II.Общие понятия.

Глава 3. Что такое CLIPS?

Глава 4. Обзор возможностей CLIPS.

ГЛАВА 3. Что такое CLIPS?.

Первоначально аббревиатура CLIPS была названием языка — C Language Integrated Production System (язык C, интегрированный с продукционными системами), удобного для разработки баз знаний и макетов экспертных систем. Теперь CLIPS представляет собой современный инструмент, предназначенный для создания экспертных систем (expert system tool). CLIPS состоит из интерактивной среды — экспертной оболочки со своим способом представления знаний, гибкого и мощного языка и нескольких вспомогательных инструментов. Сейчас, благодаря доброй воле своих создателей, CLIPS является абсолютно свободно распространяемым программным продуктом. Всем желающим доступен как сам CLIPS последней версии, так и его исходные коды. Официальный сайт CLIPS располагается по адресу: **http://www.ghg.net/clips/CLIPS.html** (рис. 3.1). Этот сайт поможет вам получить как сам CLIPS, так и всевозможный материал для его изучения и освоения (документацию, примеры, советы специалистов, исходные коды и многое другое).

Сейчас на рынке доступно не так уж много экспертных оболочек (инструментов, предназначенных для создания экспертных систем). Несмотря на то, что CLIPS распространяется бесплатно, он весьма успешно конкурирует даже с самыми известными коммерческими проектами. Количество пользователей CLIPS растет из года в год. Об этом можно судить по активности посещения сайтов, форумов и конференций, посвященных CLIPS. Если вы еще не установили CLIPS на свой компьютер — возможно, самое время сделать это. А пока, для того чтобы лучше понять философию CLIPS, его возможности и особенности, погрузимся в историю создания этой системы.

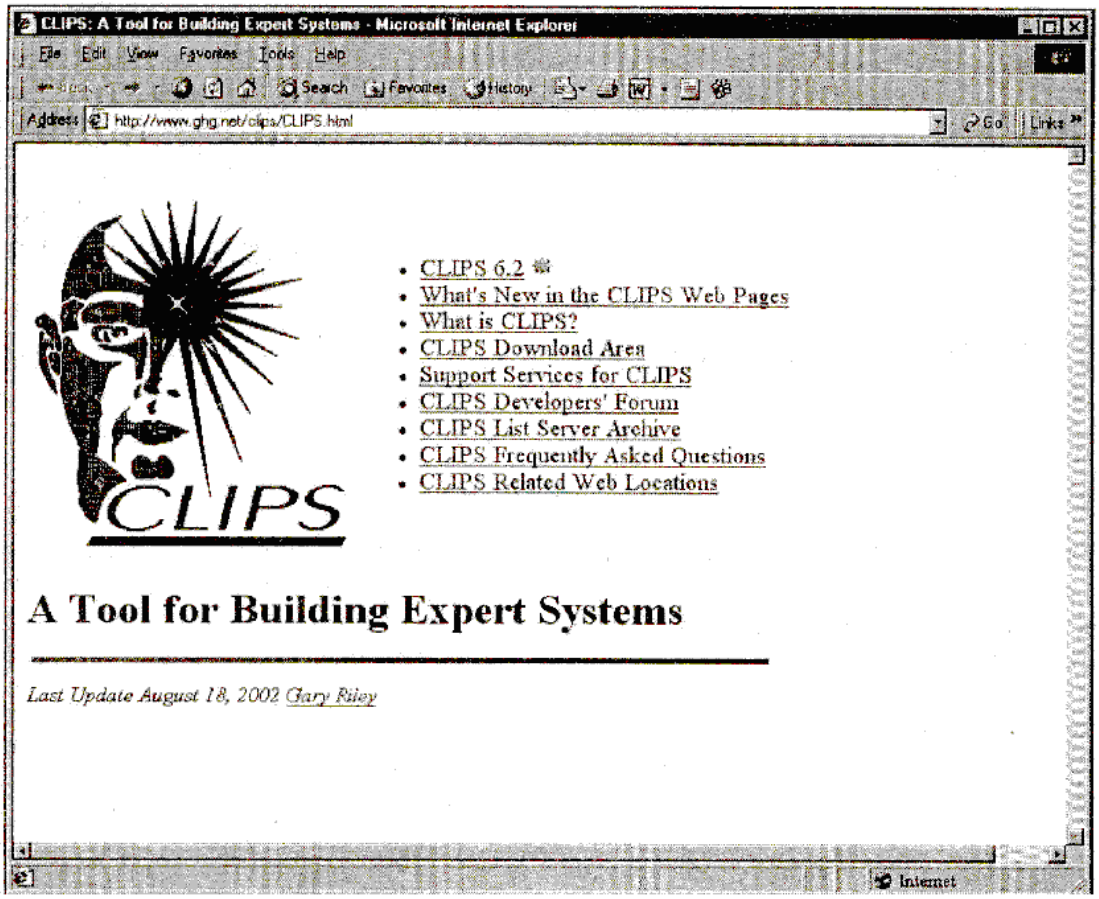

Рис. 3.1. Домашняя страница CLIPS

3.1. История создания CLIPS

Появление языка CLIPS можно датировать 1984 г., место рождения CLIPS — космический центр Джонсона NASA. Именно в это время отдел искусственного интеллекта (теперь Software Technology Branch) разработал множество прототипов экспертных систем, использующих современное программное и техническое обеспечение. Однако, несмотря на большой потенциал экспертных систем, не многие из этих приложений дошли до конечного потребителя. Эта неудача обуславливалась технологией создания экспертных систем, которой в то время оперировали в NASA. Основные ограничения накладывал язык LISP, используемый в качестве базового языка

для разработки экспертных систем. В качестве главных недостатков языка LISP можно выделить следующие три:

- недостаточная адаптируемость LISP к широкому кругу стандартных компьютеров;
- высокая цена технического и программного обеспечения, предназначенного для работы с LISP;
- низкая способность интеграции систем, написанных на LISP, с системами, написанными на других языках (производство вложенных приложений).

Сотрудники отдела искусственного интеллекта поняли, что использование традиционных языков программирования, таких как C, устранит большинство возникших проблем, и отдел начал поиски производителей и поставщиков инструментов для создания экспертных систем, оперирующих одним из традиционных языков программирования. Несмотря на то, что число подобных инструментов было достаточно велико, цена таких инструментов оказалась весьма высока. Кроме того, большинство из этих инструментов работали на очень небольшом числе платформ, а скорость их работы оставляла желать лучшего. Стало очевидно, что для получения инструмента, удовлетворяющего всем требованиям NASA, необходима разработка собственного средства для создания экспертных систем.

Прототип CLIPS был разработан весной 1985 г., немногим более чем за два месяца. Особое внимание было уделено созданию языка, совместимого с языками, использующимися в NASA в тот момент. Таким образом, синтаксис языка CLIPS был сделан очень похожим на синтаксис экспертной оболочки ART, разработанной корпорацией Inference. Несмотря на то, что ART послужил прообразом, CLIPS разрабатывался совершенно без помощи Inference или доступа к исходным кодам системы ART.

Основной целью прототипа CLIPS было создание языка, способного решать задачи, опираясь на концепцию знаний. Версия 1.0 продемонстрировала выполнимость концепций проекта. После дополнительной разработки стало очевидно, что CLIPS может стать дешевым инструментом для создания экспертных систем, моделирования и обучения. После еще одного года разработки и внутреннего использования CLIPS заметно улучшил такие свойства, как портативность, производительность и функциональность. Версия CLIPS 3.0 была выпущена летом 1986 г.

Дальнейшее усовершенствование преобразовало CLIPS из инструмента тренировки в инструмент, незаменимый при проектировании, разработке и эксплуатации экспертных систем. Версии CLIPS 4.0 и 4.1 были реализованы соответственно летом и осенью 1987 г. Их характерными особенностями были: сильно увеличенная производительность, усовершенствованные возможности интеграции с внешними языками и увеличение потенциальных возможностей. Версия CLIPS 4.2, реализованная летом 1988 г., была полностью переписанной версией CLIPS, позволяющей модульную обработку кода. Кроме того, с этой версией поставлялся справочник по архитектуре, предоставляющий детализированное описание архитектуры CLIPS, а также вспомогательная программа для верификации программ, базирующихся на правилах. Версия CLIPS 4.3 вышла летом 1989 г. и добавила системе еще большую функциональность.

Первоначально CLIPS использовал только методологию обработки данных посредством правил. CLIPS версии 5.0, вышедший весной 1991 г., ввел в CLIPS две новые парадигмы программирования: процедурное программирование (подобное используемому в языках C или Ada) и объектно-ориентированное программирование (похожее на языки Common Lisp Object System — CLOS или Smalltalk). Объектно-ориентированный язык программирования, предоставляемый системой CLIPS, называется CLIPS Object-Oriented Language (COOL). Версия CLIPS 5.1, вышедшая осенью 1991 г., улучшала поддержку разработок с использованием новых возможностей CLIPS и усовершенствовала интерфейс CLIPS для X Window, MS-DOS и Macintosh. Версия CLIPS 6.0, вышедшая 1993 г., предоставляла поддержку разработки модульных программ и тесную интеграцию объектно-ориентированных возможностей CLIPS и возможностей, базирующихся на правилах.

Так как CLIPS обладал портативностью, расширяемостью, мощностью и низкой стоимостью, он получил широкое распространение в государственных организациях, индустрии и учебных заведениях. Разработка CLIPS помогла усовершенствовать возможности технологии производства экспертных систем среди широкого диапазона приложений. Система CLIPS используется большим числом организаций, включая все отделения NASA, военные ведомства США, множество федеральных, правительственных и государственных организаций, университеты и большое число частных компаний.

CLIPS версии 6.1 был выпущен летом 1998 г. Очередная версия содержала несколько существенных улучшений. Во-первых, исходный код CLIPS стал совместим с C++. Теперь для его компиляции можно использовать любой ANSI C- или C++-компилятор. Во-вторых, в систему были добавлены несколько новых команд, предоставляющие возможность профилирования по времени выполнения конструкторов языка или определенные пользователем функций.

Последняя, доступная сейчас версия CLIPS 6.2, вышла в свет 31 марта 2002 г. Основными отличиями новой версии CLIPS являются поддержка разработки встроенных приложений, использующих CLIPS, и улучшенный интерфейс для Windows-версии, оптимизированный для использования на платформах Windows 95/98/NT.

Благодаря тому, что CLIPS является свободно распространяемым программным продуктом с доступными исходными кодами, в последнее время было выпущено множество программ и библиотек, усовершенствующих и дополняющих возможности CLIPS. Некоторые из этих продуктов являются собственностью выпустивших их компаний и предназначены для внутреннего использования или коммерческого распространения, другие, как и сам CLIPS, распространяются свободно. В качестве самых известных примеров подобных проектов можно привести DLL/OCX-библиотеку, позволяющую использовать механизм логического вывода CLIPS в ваших приложениях, FuzzyCLIPS, CLIPS++, CLIPS code generator. Большинство перечисленных продуктов, а также различная полезная информация доступна по адресу: **http://ourworld.compuserve.com/homepages/marktoml/clipstuf.htm** (рис. 3.2).

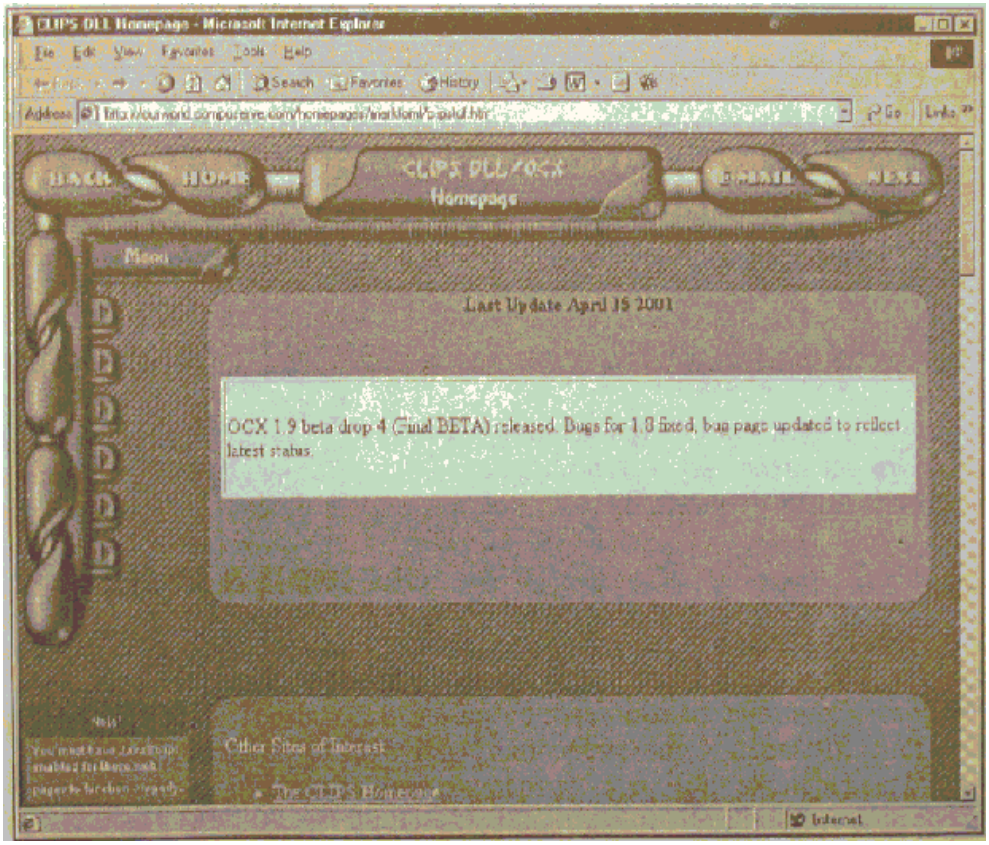

Рис. 3.2. Домашняя страница CLIPS DLL/OCX

3.2. Работа с CLIPS

Для демонстрации примеров, используемых в этой книге, будет применяться Windows-версия CLIPS 6.2. Несмотря на полную совместимость с Apple Macintosh и UNIX-версиями, при работе с данной книгой желательно использовать именно Windows-версию среды CLIPS. Внешний вид главного окна CLIPS показан на рис. 3.3.

Windows-версия среды CLIPS полностью совместима с базовой спецификацией языка. Ввод команд осуществляется непосредственно в главное окно CLIPS. Однако по сравнению с базовой Windows-версия предоставляет множество дополнительных визуальных инструментов (например, менеджеры фактов или правил, речь о которых пойдет в *гл. 5* и *6),* значительно облегчающих жизнь разработчика экспертных систем.

Экспертные системы, созданные с помощью CLIPS, могут быть запущены тремя основными способами:

- вводом соответствующих команд и конструкторов языка непосредственно в среду CLIPS;
- использованием интерактивного оконного интерфейса CLIPS (например для версий Windows или Macintosh);
- с помощью программ-оболочек, реализующих свой интерфейс общения с пользователем и использующих механизмов знаний и логического вывода CLIPS.

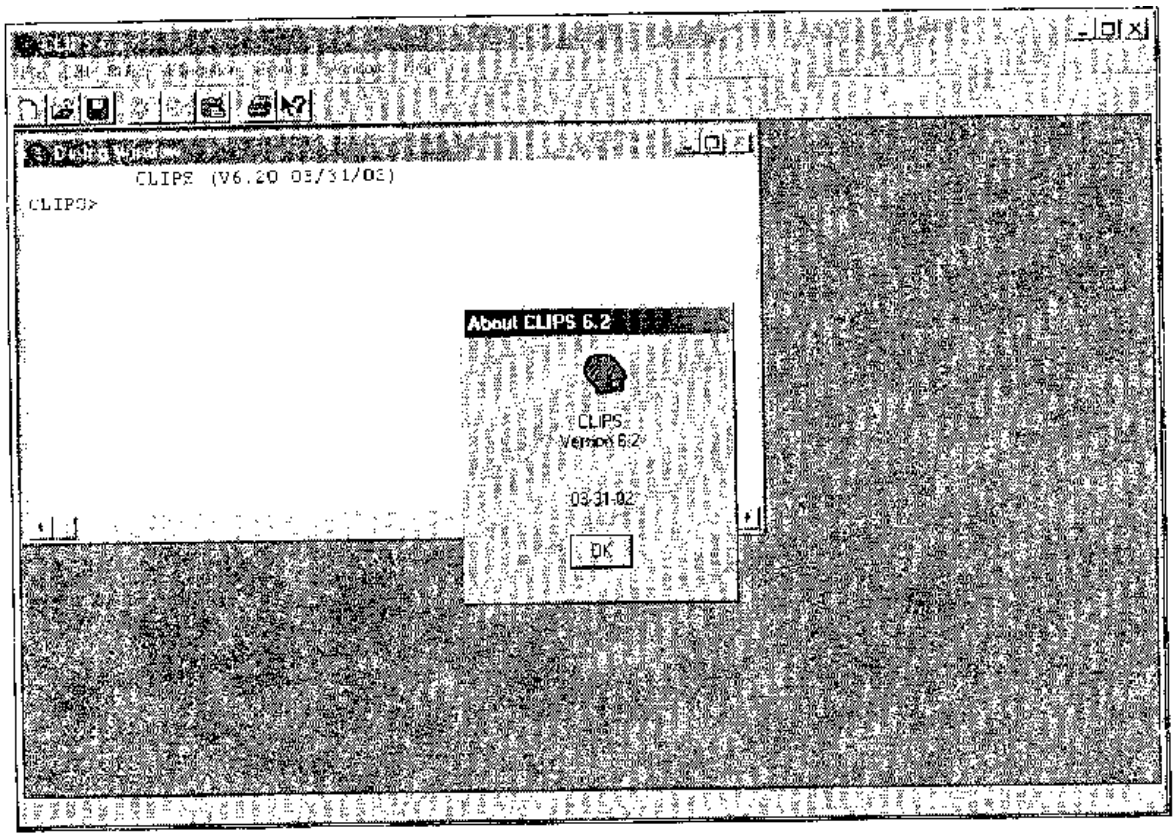

Рис. 3.3. Главное окно CLIPS

Кроме того, CLIPS при запуске позволяет выполнять командные файлы собственного формата (эта возможность также доступна при помощи команды batch). Для реализации этой возможности необходимо запустить CLIPS (в нашем случае это файл CLIPSWin.exe) с одним из трех следующих аргументов:

- -f <имя-файла>
- -f2 <имя-файла>
- -l <имя-файла>

Текстовый файл, заданный с помощью опции -f, должен содержать команды CLIPS. Если заданный файл содержит команду exit, то CLIPS завершит свою работу и пользователь вернется в операционную систему. В случае если команда exit отсутствует, то после выполнения всех команд из заданного файла пользователь попадет в главное окно CLIPS. Команды в текстовом файле должны быть набраны так же, как если бы они вводились непосредственно в командную строку CLIPS (т. е. все команды должны быть заключены в круглые скобки и разделены символом перехода на новую строку). Опция -f фактически эквивалентна запуску команды batch сразу после запуска CLIPS.

Опция -f2 идентична -f, но, в отличие от опции -f, она использует команду batch*. Файл, заданный этой опцией, также выполняется после запуска CLIPS, но результаты выполнения команд не отображаются на экране.

Опция -l задает текстовый файл, содержащий конструкторы CLIPS, которые тут же запускаются на выполнение. Использование этой опции эквивалентно использованию команды load сразу после запуска CLIPS.

Создание программ-оболочек, использующих возможности CLIPS, выходит за рамки этой книги. Желающим использовать эту возможность CLIPS можно рекомендовать обратиться к книге "CLIPS Reference Manual, Volume II, Advanced Programming Guide ".

Основным методом общения с CLIPS, используемым в данной книге, является применение командной строки. После появления в главном окне CLIPS приглашения — CLIPS > — команды пользователя могут вводиться в среду непосредственно с клавиатуры. Команды могут быть вызовами системных или пользовательских функций, конструкторами различных данных CLIPS и т. д. В случае вызова пользователем некоторой функции, она немедленно выполняется, и результат ее работы отображается пользователю. Для вызова функций или операций CLIPS

использует префиксную нотацию — аргументы всегда следуют после имени функции или операции. При вызове конструкторов CLIPS создает новый объект соответствующего типа, так или иначе представляющий некоторые знания в системе. При вводе в среду имени созданной ранее глобальной переменной CLIPS отобразит ее текущее значение. Ввод в среду некоторой константы просто приведет к ее немедленному отображению в главном окне CLIPS. В качестве примера введите в среду следующие команды (листинг 3.1).

Листинг 3.1. Пример команд CLIPS

```
(+ 3  4)
(defglobal  ?*x* = 3)
?*x*
red
```

Результат выполнения этих команд приведен на рис. 3.4.

Первой командой данного примера вызывается функция арифметического сложения двух чисел: 3 и 4. Результат работы этой функции — 7 сразу же отображается на экран. После этого, с помощью конструктора defglobal (см. гл. 7), создается глобальная переменная ?*x*, которая инициализируется значением 3. При вводе в командную строку имени глобальной переменной ?*x* CLIPS отображает ее текущее значение, равное 3. Последней командой была введена некая константа red, значение которой было тут же выведено на экран.

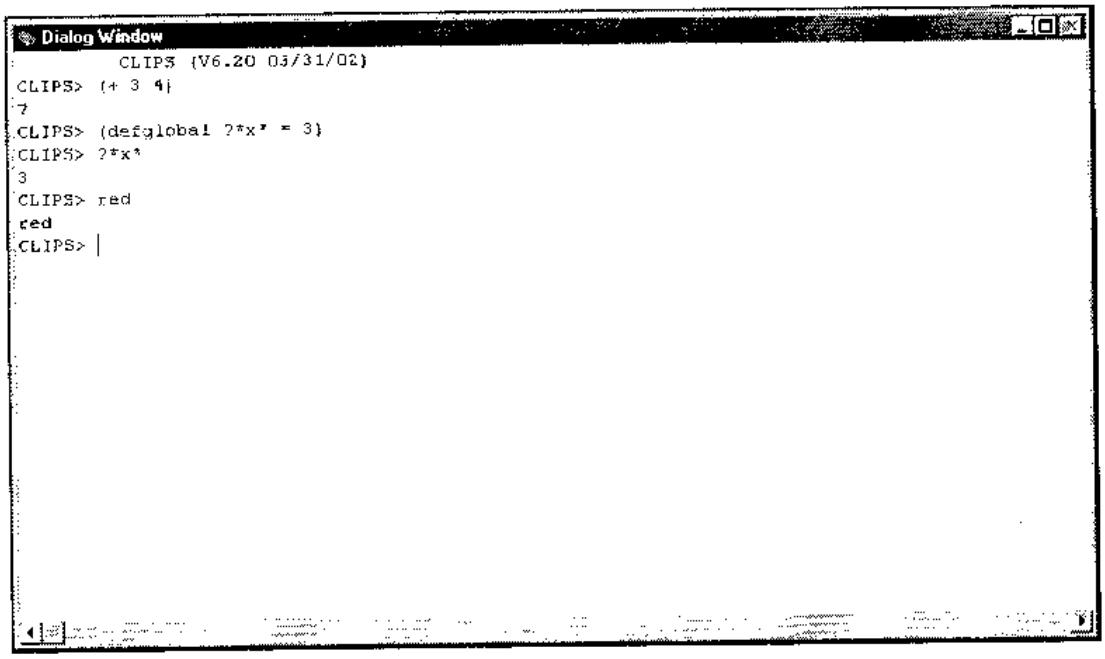

Рис. 3.4. Результат выполнения команд листинга 3.1

В заключение данной главы рассмотрим синтаксис, используемый в этой книге для определения тех или иных конструкций языка.

3.3. Синтаксис определений

В качестве базового синтаксиса для определения конструкций языка используется стандартная БНФ-нотация. Ниже приведены правила, используемые для построения определений.

Слово или выражение, заключенное в угловые скобки, называется нетерминальным символом (например, <string>). Нетерминальный символ требует дальнейшего определения. Слова или выражения, не заключенные в угловые скобки, называются терминальными символами, и представляют синтаксис описываемой конструкции языка CLIPS. Терминальные символы (особенно круглые скобки) должны вводиться в командную строку именно так, как показано в определении. Если за нетерминальным символом следует символ *, то это означает, что в данном месте может находиться список из нуля или более элементов этого типа. Если же за нетерминальным символом следует +, то в данном месте может находиться список из одного или более элементов этого типа. Символы * и +, встречающиеся сами по себе (не следующие после

нетерминальных символов), являются терминальными. Многоточие, как горизонтальное, так и вертикальное, также используется для отображения списка из одного или более элементов. Элементы, заключенные в квадратные скобки (например, [<комментарии>]), являются необязательными элементами, которые могут входить в определение. Вертикальная черта, разделяющая два или более элемента определения, указывает на то, что в конструкции необходимо использовать один из перечисленных элементов. Символ ::= используется для обозначения необходимости замены некоторого нетерминального символа. Например, определение:

<lexeme> : : = <symbol> I <string>

обозначает, что нетерминальный символ <lexeme>, встречающийся в некотором определении, должен быть заменен либо на символ <symbol>, либо на символ <string>. Пробелы, символы табуляции, переходы на другую строку используются только для логического разделения элементов определения и игнорируются CLIPS (кроме строк, заключенных в двойные кавычки).

В *приложении 1* обобщен список БНФ-определений общих конструкций языка, приведенных в книге.

ГЛАВА 4. Обзор возможностей CLIPS.

Настоящая глава посвящена обзору возможностей CLIPS, а также описанию основных концепций этой системы и терминов, применяемых в данной книге.

4.1. Основные элементы языка

Синтаксис языка CLIPS можно разбить на три основных группы элементов, предназначенных для написания программ:

- примитивные типы данных;
- функции, использующиеся для обработки данных;
- конструкторы, предназначенные для создания таких структур языка, как факты, правила, классы и т. д.

Рассмотрим каждую из этих трех групп более подробно.

4.1.1. Типы данных

CLIPS поддерживает 8 *примитивных типов данных:* float, integer, symbol, string, external-address, fact-address, instance-name, instance-address.

Для хранения численной информации предназначаются типы float и integer, для символической — symbol и string.

Число в CLIPS может состоять только из символов цифр (0—9), десятичной точки (.), знака (+ или -) и экспоненциального символа (e) с соответствующим знаком, в случае представления числа в экспоненциальной форме. Ниже приведены примеры допустимых в CLIPS представлений целых и вещественных типов:

Пример 4.1. Представление чисел в CLIPS

Целые:	237	15	+12	-32
Вещественные:	237e3	15.09	+12.0	-32.3e-7

Определение целого значения можно представить таким образом:

Определение 4.1. Представление целого числа

<целое> ::= [+ | -] <цифра>+
<цифра> : := 0 | 1 | 2 | 3 | 4| 5| 6 | 7 | 8 | 9

Вещественное значение имеет следующий синтаксис:

Определение 4.2. Представление вещественного числа

```
<вещественное>      ::= <целое> <экспонента> |
                        <целое> . [экспонента] |
                        <беззнаковое-целое> [экспонента] |
                         <целое> . <беззнаковое-целое> [экспонента]
<беззнаковое-целое>  ::= <цифра>+
<экспонента>        : : = e  | E <целое>
```

Если последовательность символов не соответствует приведенным выше определениям целого или вещественного числа, то данная последовательность воспринимается CLIPS как значение типа symbol.

Значением типа symbol может быть любая последовательность символов, начинающаяся с любого не управляющего ASCII-символа. Значение типа symbol заканчивается *ограничителем.* Ограничителями являются любые неотображаемые символы (например, пробел, символ табуляции

или перехода на другую строку), двойные кавычки, открывающая или закрывающая круглая скобка, символы &, |, < и ~. Точка с запятой (;) является символом начала комментариев и также может ограничивать значение типа symbol. Символы-ограничители не могут содержаться в значении symbol, за исключением <, который может быть первым символом значения. Значение типа symbol не может начинаться с символа ? или $?, но может содержать эти символы. CLIPS является языком, чувствительным к регистру.

Замечание

Значения типов float и integer являются частным случаем значения типа symbol. Другими словами, они удовлетворяют всем ограничениям, налагаемым на значение типа symbol.

Ниже приведены несколько примеров значений типа symbol:

Пример 4.2. Допустимые значения типа symbol

```
foo         Hello          B76-HI      bad_value
127A        456-93-039     @+=-%       2each
```

Значение типа, string представляет собой строку символов, заключенную в двойные кавычки. Символ двойных кавычек также может быть включен в строку. Для этого перед символом " необходимо поставить символ обратной косой черты (\). Для включения в строку символа обратной косой черты необходимо использовать два последовательных символа \. Примеры допустимых значений string приведены ниже:

Пример 4.3. Допустимые значения типа string

```
"foo"          "a  and b"      "1  number"      "a\"quote"
```

Замечание

Значение "abcd" типа string не эквивалентно значению abcd типа symbol. Несмотря на то, что они состоят из идентичных символов, они относятся к разным типам.

Значение типа external-address представляет собой адрес структуры данных, возвращенной внешней функцией (например, написанной на языке C или Ada), интегрированной с программой CLIPS. Значение этого типа может быть создано только посредством вызова *внешней функции.* Использование внешних функций выходит за рамки данной книги, поэтому вы не найдете примеров создания и использования этого типа. В CLIPS значения данного типа отображаются следующим образом:

Пример 4.4. Значение типа external-address

<Pointer-XXXXXX>

где XXXXXX — число, представляющее внешний адрес.

Подробную информацию о типе external-address можно найти в книге *"CLIPS Reference Manual, Volume II, Advanced Programming Guide".*

Факт в CLIPS представляет собой список атомарных значений примитивных типов, ссылаться на которые можно либо используя порядок определения этих значений, в случае *упорядоченных фактов,* либо по имени, в случае *неупорядоченных фактов* или *шаблонов.* Подробней понятие факта будет описано в *гл. 5*. Оперировать с фактом можно, используя его адрес *(см. разд. 4.2.1).* Адрес факта представляет собой значение типа fact-address.

Пример 4.5. Значение типа fact-address

<Fact-XXX>

где XXX представляет собой индекс факта.

Объект в CLIPS представляет собой экземпляр определенного пользователем *класса.* Для определения класса используется конструктор defclass *(см. разд. 11.2).* Для создания объекта используется функция make-instance. Ссылаться на объект можно либо по адресу, либо, в рамках отдельного модуля *(см. гл. 12),* по имени объекта. Тип instance-name предназначен для хранения значения имени объекта. Для представления имени используется значение типа symbol,

окруженное квадратными скобками ([и]). Ниже приведено несколько примеров допустимых значений типа instance-name:

Пример 4.6. Значения типа instance-name

[pump-l] [foo] [+++] [123-890]

Замечание

Квадратные скобки не являются частью имени объекта, а служат своеобразными ограничителями, которые позволяют системе отличать значение типа instance-name от значения типа symbol.

Тип instance-address предназначен для хранения значения, представляющего адрес объекта. Значение этого типа может быть получено посредством вызова функции instance-address или в результате выполнения операции сопоставления образцов в правиле *(см. гл. 6).* Ссылки на объект, с использованием значения типа instance-address, происходят значительно быстрее, чем ссылки по значению instance-name. Значения типа instance-address отображаются в CLIPS следующим образом:

Пример 4.7. Значение типа instance-address

<Instance-XXX>

где XXX — индекс объекта.

Место для хранения значения одного из примитивных типов в CLIPS называется *полем* или *простым полем. Константа* представляет собой неизменяемое простое поле, заданное последовательностью символов (с помощью констант нельзя задавать значения типов external-address, fact-address и instance-address — значения этих типов могут быть получены только с помощью вызовов соответствующих функций и должны храниться в переменных). Последовательность из 0 или более простых полей образует *составное поле.* Для вывода составного поля на экран CLIPS группирует данные такого поля с помощью круглых скобок. Несколько примеров составных полей приведено ниже:

Пример 4.8. Составные поля

(a) (1 bar foo) () (x 3.0 "red" 567)

Замечание

Составное поле (a) не эквивалентно простому полю a.

Составные значения создаются либо вызовом функций, возвращающих составные значения, либо с помощью специального группового аргумента в конструкторах функций, методов или обработчиков сообщений, либо в результате выполнения процесса сопоставления образцов в правилах.

Переменной является значение некоторого типа, сохраненное в простом или составном поле и имеющее некоторое имя. Переменные используются В конструкторах CLIPS (в частности в defrule, deffunction, defmethod и defmessage-handier). Описания использования переменных в конструкторах приведены в соответствующих разделах.

4.1.2. Функции

Функцией в CLIPS называется часть кода, имеющая имя и возвращающая полезный результат или выполняющая полезные действия (например, отображение информации на экране). В дальнейшем в книге функциями будут называться, как правило, только функции, возвращающие результат. Функции, не возвращающие результат и выполняющие некоторую полезную работу, как правило, называются *командами.*

CLIPS оперирует с несколькими типами функций — *определенные пользователем внешние функции, системные (внутренние) функции, функции, определенные в среде CLIPS с помощью конструктора* deffunction, *родовые функции.* Определенные пользователем внешние функции и системные функции создаются на внешних языках программирования (например, С), и затем подключаются к CLIPS на этапе компилирования или функционирования среды. Системные функции созданы

разработчиками среды CLIPS и описаны в руководстве. Описания наиболее важных системных функций приведены *в гл. 15* и *16.*

Конструктор deffunction позволяет пользователям определять новые функции непосредственно в CLIPS. Функции, созданные таким образом, действуют так же, как внешние или системные функции CLIPS, за исключением того, что вместо непосредственного выполнения (как, например, в случае вызова определенной пользователем внешней функции) вызов такой функции обрабатывается встроенным интерпретатором языка CLIPS. Подробней о функциях, созданных с помощью конструктора deffunction, будет рассказано в *гл. 8.*

Родовые функции определяются с помощью конструкторов defgeneric и defmethod. Родовые функции позволяют выполнять различные действия, в зависимости от набора аргументов, заданных при вызове функции. Таким образом функция перегружается различными реализациями (подобный механизм перегрузки функций можно встретить, например, в языке *C++).* Более подробно родовые функции описаны в *гл. 10.*

Вызов функций в CLIPS имеет *префиксную нотацию* — аргументы функции всегда следуют после имени функции. При вызове имя функции вместе со всеми аргументами заключается в круглые скобки. Аргументы отделяются друг от друга по крайней мере одним пробелом. Аргументами функций могут быть переменные примитивных типов, константы или вызовы других функций. Ниже приведены примеры использования функций + (арифметическое сложение) и * (арифметическое умножение):

Пример 4.9. Использование функций + и *

```
( + 3 4 5 )
(* 5 6.0 2)
(+ 3 (* 8 9) 4)
(* 8 ( + 3 (* 2 3 4) 9) (* 3 4))
```

Выражением в CLIPS называется неименованный отрезок кода, вызывающий функции с некоторым набором аргументов. Фактически предыдущий пример состоит из четырех выражений.

4.1.3. Конструкторы

В CLIPS определены следующие конструкторы: defmodule, defrule, deffacts, deftemplate, defglobal, deffunction, defclass, definstances, defmessage-handler, defgeneric и defmethod. Вызовы всех конструкторов заключаются в круглые скобки. Конструкторы отличаются от встроенных функций по выполняемым ими действиям. Как правило, функции не меняют состояние базы знаний среды CLIPS (за некоторым исключением, например, функций, очищающих среду или загружающих на выполнение некоторый файл). Конструкторы, наоборот, предназначены для добавления в базу знаний новых элементов. Кроме того, в отличие от функций, конструкторы не возвращают никаких значений.

Как и в любом языке программирования, *в* CLIPS хорошим тоном считается использование комментариев. Все конструкторы (за исключением defglobal) позволяют вставлять комментарии непосредственно в код определения нового элемента базы знаний. Комментарии также могут быть помещены в любое место программы с помощью символа ;. Все символы, следующие за ; до конца строки, игнорируются CLIPS. Комментарии, созданные с помощью символа ; , не сохраняются в среде CLIPS, поэтому их использование разумно только в текстовых файлах. В книге вы встретите множество примеров использования комментариев.

4.2. Абстракции данных

Для представления данных CLIPS использует три основных *абстракции данных:* факты, объекты и глобальные переменные. В данном разделе подробно описана каждая из этих форм представления информации.

4.2.1. Факты

Факты — одна из основных форм представления информации в CLIPS. Факты являются фундаментальным понятием теории экспертных систем и предназначены для использования в *правилах* системы. Каждый факт представляет фрагмент данных, помещенных в текущий *список фактов системы (рабочую память).*

Факт может быть добавлен в текущий список фактов системы (с помощью команды assert), удален из него (команда retract), изменен (modify) или продублирован (duplicate) пользователем, в процессе интерактивной работы в системе, либо из программы. Количество фактов, которые может содержать список фактов, а также количество информации, содержащейся в каждом факте, ограничено только свободной памятью вашего компьютера. В случае выполнения пользователем попытки добавить в систему факт, точно соответствующий уже существующему, данная операция будет игнорирована, хотя подобное поведение системы можно изменить *(см. гл. 5)*.

Некоторые команды, такие как retract, modify или duplicate, требуют в качестве параметра некоторого уже существующего факта. Факт может быть задан с помощью значений типа fact-index или fact-address. После создания (или изменения) факт получает уникальный индекс, называемый *индексом факта* (fact-index). Индекс фактов начинаются с 0 и увеличивается на 1 при каждом добавлении или изменении факта. При выполнении команды reset или clear текущий индекс фактов обнуляется. Для определения конкретного факта с помощью типа fact-address необходимо получить соответствующее значение от функции, возвращающей значение данного типа (например, assert, modify или duplicate), или некоторого правила.

Для удобства отображения фактов в CLIPS используется понятие *идентификатора факта.* Идентификатор факта состоит из символа f, следующего за ним знака - и индекса факта. Например, идентификатор f-io ссылается на факт с индексом 10.

Для хранения фактов используется один из двух следующих форматов: упорядоченные факты и неупорядоченные факты или шаблоны.

Упорядоченные факты

Упорядоченный факт состоит из значения типа symbol и следующей за ним последовательности из нуля или более значений типа symbol. Факт заключается в круглые скобки, а значения в последовательности отделяются друг от друга пробелами. Первое поле упорядоченного факта определяет так называемое *отношение,* или *связь* факта. Например, факт (father-of jack bill) показывает, что отцом Джека является Билл. Ниже приведено несколько примеров упорядоченных фактов:

Пример 4.10. Упорядоченные факты

```
(the pump  is  on)
(altitude  is  10000  feet)
(grocery-list bread milk eggs)
```

Поля в упорядоченном факте могут хранить данные любого примитивного типа CLIPS, за исключением первого поля, тип которого должен быть symbol. Следующие слова зарезервированы и не могут быть использованы в качестве первого поля: test, and, or, not, declare, logical, object, exist и forall.

Неупорядоченные факты

Так как упорядоченный факт хранит информацию, используя строго заданные позиции данных, то для доступа к необходимой информации пользователь должен знать не только какие данные сохранены в факте, но и какое поле содержит эти данные. *Неупорядоченные факты (или шаблоны)* предоставляют пользователю возможность задавать абстрактную структуру факта путем назначения имени каждому полю. Для создания шаблонов, которые впоследствии будут применяться для доступа к полям факта по имени, используется конструктор deftemplate. Конструктор deftemplate, по сути, аналогичен определениям записей или структур в таких языках программирования, как Pascal или C.

Конструктор deftemplate задает имя шаблона и определяет последовательность из нуля или более полей неупорядоченного факта, называемых *слотами. Слот* состоит из имени, заданного значением типа symbol, и следующим за ним списка полей. Как и факт, слот с обеих сторон ограничивается круглыми скобками. В отличие от упорядоченных фактов слот неупорядоченного факта может жестко определять тип своих значений. Кроме того, слоту могут быть заданы значения по умолчанию.

Замечание

Слоты не могут быть использованы в упорядоченных фактах, а в неупорядоченных файлах, в свою очередь, нельзя ссылаться на данные, используя порядок слотов.

CLIPS отличает неупорядоченные факты от упорядоченных по первому полю факта. Первое поле фактов любого типа должно быть значением типа symbol. Если это значение соответствует имени некоторого шаблона, то факт является неупорядоченным. Как и упорядоченные факты, неупорядоченные ограничиваются скобками.

Пример 4.11. Неупорядоченные факты

```
(client (name "Joe Brown") (id X9345A))
(point-mass (x-velocity 100) (y-velocity -200))
(class (teacher "Martha Jones") (#-students 30) (Room "37A"))
(grocery-list (#-of-items 3) (items bread milk eggs))
```

Замечание

Порядок слотов в неупорядоченном факте не важен. Например, все приведенные ниже факты считаются идентичными:

```
(class (teacher "Martha Jones") (#-students 30) (Room "37A"))
(class (#-students 30) (teacher "Martha Jones") (Room "37A"))
(class (Room "37A") (#-students 30) (teacher "Martha Jones"))
```

В отличие от фактов, приведенных выше, упорядоченные факты из следующего примера не являются идентичными:

```
(class "Martha Jones" 30 "37A")
(class 30 "Martha Jones" "37A")
(class "37A" 30 "Martha Jones")
```

Очевидными преимуществами применения шаблонов являются более высокая читабельность и независимость слотов от порядка их определения.

Так же как и упорядоченные факты, шаблоны можно добавлять в список фактов и удалять из него. Кроме того, существует возможность модификации и дублирования шаблонов.

Инициализация фактов

Конструктор deffacts позволяет создавать набор фактов, инициализирующий базу знаний CLIPS, при каждой очистке системы. При выполнении команды reset текущий список фактов CLIPS очищается, а затем в него добавляются все факты, заданные конструкторами deffacts. CLIPS содержит один предопределенный системный конструктор deffacts, который выполняет добавление в систему факта initial-fact. Более подробно особенности создания и использования фактов в CLIPS описаны в *гл. 5*.

4.2.2. Объекты

Объект CLIPS состоит из двух основных частей — свойств объекта и его поведения. Класс представляет собой своеобразный шаблон, определяющий общие свойства и поведение объектов — экземпляров этого класса. Объекты могут принадлежать классам, определенным пользователем, или некоторым системным классам (например, классам, реализующим представление объектов в виде примитивных данных). Ниже приведены несколько примеров объектов и названия их классов:

Пример 4.12. Объекты и их классы

Объекты (вид, отображаемый на экран)	Классы
Rolls-Royce	SYMBOL
"Rolls-Royce"	STRING
8.0	FLOAT
8	INTEGER
(8.0 Rolls-Royce 8 [Rolls-Royce])	MULTIFIELD
<Pointer- OOCF61AB>	EXTERNAL-ADDRESS
[Rolls-Royce]	CAR (класс, определенный пользователем)

Объекты CLIPS разделяются на две важные категории: *объекты, хранящие примитивные типы данных,* и *объекты классов, определенных пользователем.* Объекты этих двух типов отличаются способом своего создания и удаления, наборами свойств и даже методом использования.

Объекты примитивных типов неявно создаются и удаляются системой CLIPS в местах, где это необходимо. По ссылке на такой объект можно получить хранящееся в нем значение соответствующего типа. Объекты примитивных типов не имеют слотов и, как правило, не имеют имен. Классы, определяющие эти объекты, являются предопределенными классами CLIPS. Функциональность предопределенных классов, определяющих объекты примитивных типов, подобна функциональности классов, определенных пользователем, за исключением того, что к таким классам нельзя присоединить обработчики сообщений. Это делает не очень удобным использование таких классов в объектно-ориентированном программировании. Основным назначением объектов примитивных типов является использование их в определении родовых функций. Родовые функции применяют эти объекты в качестве своих аргументов и, по заданному набору, в момент вызова, определяют, какой именно метод родовой функции должен быть вызван. Более подробно данный процесс изложен в *гл. 10.*

Для ссылки на объект класса, определенного пользователем, необходимо использовать имя или адрес объекта. Такие объекты явно создаются или удаляются с помощью сообщений или специальных системных функций. Свойства объекта класса, определенного пользователем, определяются набором *слотов,* заданных при определении класса. Слот объекта имеет имя и может содержать простое или составное значение. Например, объект Rolls-Royce является объектом созданного пользователем класса CAR. Один из слотов такого объекта может, например, содержать цену автомобиля со значением 75 000. Поведение объектов определяется наличием тех или иных *обработчиков сообщений,* присоединенных к соответствующему классу. Обработчики сообщений и способы работы с объектами подробно описаны в *гл. 11.* Все объекты одного класса имеют одинаковые наборы слотов, но могут содержать в этих слотах различные значения. Однако, если два объекта имеют одинаковые наборы слотов, это еще не означает, что они принадлежат одному классу, т. к. два абсолютно разных класса, теоретически, могут иметь одинаковые наборы слотов.

Основная разница между слотами объекта и неупорядоченного факта заключается в возможности *наследования.* Наследование позволяет использовать в классе некоторые свойства и поведение, определенные в другом классе. COOL (объектно-ориентированная часть языка CLIPS) поддерживает *множественное наследование,* при котором класс может наследовать слоты и обработчики сообщений непосредственно от нескольких классов.

Инициализация объектов

Конструктор definstances позволяет создавать набор объектов, добавляющихся в базу знаний CLIPS при каждой очистке системы. При выполнении команды reset среда CLIPS очищается, а затем в список объектов добавляются все объекты, заданные конструкторами def instances. CLIPS содержит один предопределенный системный конструктор definstances, который вызывает добавление в систему объекта initial-object. К более подробному описанию особенностей создания и использования объектов мы вернемся в *гл. 11.*

4.2.3. Глобальные переменные

Конструктор defglobal предназначен для определения *глобальных переменных.* Доступ к такой переменной можно получить из любого места среды CLIPS, а значения, которые они содержат, не зависят ни от каких других конструкций языка. В отличие от этого, некоторые конструкторы (например, defruie или deffunction) позволяют создавать локальные переменные. Эти локальные переменные доступны только внутри тела соответствующего правила или функции. Глобальные переменные CLIPS подобны глобальным переменным, встречающимся в таких традиционных процедурных языках, как C или Ada. Однако, в отличие от них, глобальные переменные CLIPS являются слабо типизированными. Они способны хранить значение любого типа.

4.3. Представление знаний

CLIPS поддерживает как *эвристическую,* так и *процедурную парадигму представления знаний.* Обе эти парадигмы описаны в данном разделе. Объектно-ориентированное программирование, комбинирующее обе эти парадигмы, описано в *разд. 4.4.*

4.3.1. Эвристические знания

Одним из основных методов представления знаний в CLIPS являются *правила.* Правила используются для представления эвристик или эмпирических правил, определяющих действия, которые необходимо выполнить в случае возникновения некоторой ситуации. Разработчик экспертной системы создает набор правил, которые, работая вместе, решают поставленную задачу. Правила состоят из *предпосылок* и *следствия.* Предпосылки называются также *ЕСЛИ-частью* правила или *LHS* правила (left-hand side). Следствие называются *ТО-частью* правила или *RHS* правила (right-hand side).

Предпосылки правила представляют собой набор *условий* (или *условных элементов),* которые должны удовлетвориться, для того чтобы правило выполнилось. Предпосылки правил удовлетворяются в зависимости от наличия или отсутствия некоторых заданных фактов в списке фактов или некоторых созданных объектов, являющихся экземплярами классов, определенных пользователем. Один из наиболее распространенных типов условных выражений в CLIPS — *образцы* (patterns). Образцы состоят из набора *ограничений,* которые используются для определения того, удовлетворяет ли некоторый факт или объект условному элементу. Другими словами, образец задает некоторую маску для фактов или объектов. Процесс сопоставления образцов фактам или объектам называется *сопоставлением образцов* (pattern-matching). CLIPS предоставляет механизм, называемый *механизмом логического вывода* (inference engine), который автоматически сопоставляет образцы с текущим списком фактов и определенными объектами и ищет правила, которые применимы в настоящий момент.

Следствие правила представляется набором некоторых действий, которые нужно выполнить, в случае если правило применимо к текущей ситуации. Таким образом, действия, заданные в следствии правила, выполняются по команде механизма логического вывода, если все предпосылки правила удовлетворены. В случае, если в данный момент применимо более одного правила, механизм логического вывода использует текущую *стратегию разрешения конфликтов* (conflict resolution strategy), которая определяет, какое именно правило будет выполнено. После этого CLIPS выполняет действия, описанные вследствие выбранного правила (которые могут оказать влияние на список применимых правил), и приступает к выбору следующего правила. Этот процесс продолжается до тех пор, пока список применимых правил не опустеет.

В большинстве случаев правила CLIPS можно представить в виде операторов IF-THEN, используемых в процедурных языках программирования, например, таких как Ada или C. Однако условные выражения IF-THEN в процедурных языках проверяются только тогда, когда поток управления программы непосредственно попадает на данное выражение путем последовательного перебора выражений и операторов, составляющих программу. В CLIPS, в отличие от этого, механизм логического вывода создает и постоянно модифицирует список правил, условия которых в данный момент удовлетворены. Эти правила запускаются на выполнение механизмом логического вывода. С этой стороны правила похожи на обработчики сообщений, присутствующие в таких языках, как, например, Ada или Smalltalk.

4.3.2. Процедурные знания

Помимо эвристической, CLIPS поддерживает и *процедурную парадигму представления знаний,* используемую в большинстве языков программирования. Конструкторы deffunction и defgeneric позволяют пользователю определять новые выполняемые конструкции непосредственно в среде CLIPS, возвращающие некоторые значения или выполняющие какие-то полезные действия. Вызов этих новых функций ничем не отличается от вызова встроенных функций CLIPS. Обработчики сообщений позволяют пользователю определять поведение объектов, с помощью задания той или иной реакции на сообщения. Функции, родовые функции и обработчики сообщений представляют собой отрезки кода, заданного пользователем и выполняемого, в случае необходимости, интерпретатором CLIPS. Кроме того, механизм модулей (конструктор defmodule, *см. гл. 12)* позволяет разбивать базу знаний CLIPS на отдельные смысловые части.

Функции

Конструктор deffunction позволяет создавать новые *функции* непосредственно в CLIPS. Более ранние версии CLIPS позволяли использовать только внешние пользовательские функции, написанные на каком-нибудь языке программирования (чаще всего Си) и присоединенные к среде CLIPS.

Тело функции, определенной с помощью конструктора deffunction, представляет собой последовательность действий, подобную используемой в правой части правил. Заданные пользователем действия выполняются при вызове соответствующей функции. Значение,

возвращаемое функцией, является результатом вычисления последнего действия. Подробно тема функций освящена в *гл. 7.*

Родовые функции

Родовые функции, так же как и обычные функции, могут быть созданы непосредственно в CLIPS. Способ вызова таких функций также ничем не отличается от способа вызова обычных функций. Однако родовые функции гораздо мощнее обычных, т. к. они способны перегружаться. Благодаря механизму перегрузки родовая функция может выполнять различные действия в зависимости от типа и числа аргументов. Обычно родовая функция состоит из нескольких компонентов, называемых *методами.* Каждый метод содержит различный набор аргументов родовой функции.

Например, можно перегрузить системную функцию + (арифметическое сложение) для выполнения операции конкатенации двух строк. Однако после этого функция + все еще сможет выполнять арифметическое сложение. В данном примере у родовой функции + существует два метода: первый метод явно определен пользователем для конкатенации двух строк, второй представляет собой неявный вызов стандартной функции, выполняющей арифметическое сложение. Значение, возвращенное родовой функцией, является значением, полученным в результате вычисления последнего действия в применяемом методе.

Обработчики сообщений

Как уже упоминалось, объект CLIPS состоит из двух основных частей — свойств объекта и его поведения. Свойства объектов определяются в терминах слотов. Поведение объекта обуславливается *обработчиками сообщений,* которые являются присоединенной к классу последовательностью действий с заданным именем. Любые манипуляции с объектом можно выполнить только с помощью сообщений. Например, в случае уже упоминавшегося объекта Rolls-Royce, являющегося экземпляром пользовательского класса CAR, для того чтобы запустить двигатель, пользователь должен послать объекту сообщение start-engine с помощью функции send. То, каким образом объект Rolls-Royce прореагирует на это сообщение, зависит от определения обработчика сообщения start-engine, связанного с классом CAR. Назначение обработчиков сообщений в принципе эквивалентно назначению любой функции — возвращение результата или выполнение неких полезных действий. Более подробно тема объектов раскрывается в *гл. 11.*

Модули

Модули позволяют разбивать базу знаний на отдельные смысловые части. Каждый вызываемый конструктор помещается в определенный модуль. Программист имеет возможность контролировать возможности доступа и видимость конструкторов в тех или иных модулях. Кроме того, для модулей можно устанавливать видимость определенных фактов или объектов. Модули можно использовать для контроля или изменения потока выполнения правил. Подробное описание модулей приведено в *гл. 12.*

4.4. Объектно-ориентированные возможности CLIPS

Данный раздел дает краткий обзор элементов языка CLIPS Object-Oriented Language (COOL) — встроенного языка CLIPS, предоставляющего объектно-ориентированные возможности.

4.4.1. Отличия COOL от других объектно-ориентированных языков

В так называемых "чистых" объектно-ориентированных языках абсолютно все программные элементы являются объектами, и любые действия над ними выполняются посредством посылки сообщений. В CLIPS объектами являются только объекты классов, определенных пользователем, и объекты, представляющие данные примитивных типов CLIPS. С объектами, представляющими данные примитивных типов, можно манипулировать с помощью сообщений, а для объектов классов, определенных пользователем, это является единственно возможным способом работы с объектом. Например, в "чистых" объектно-ориентированных языках для сложения двух чисел первому из них передается сообщение add и в качестве аргумента передается второе. В CLIPS для этого достаточно просто вызвать функцию + и в качестве аргументов передать ей два числа. Однако вы можете определить соответствующий обработчик сообщения add для класса NUMBER и работать с числами в стиле "чистых" систем ООП.

Работа со всеми программными элементами CLIPS, не являющимися объектами, выполняется не в объектно-ориентированном стиле, а с помощью вызовов соответствующих функций. Например, для вывода на экран определения правила используется функция ppdefrule, которой нужное

правило передается в качестве параметра, а не посылается сообщение print, так как правило не является объектом.

4.4.2. Основные возможности ООП

Любая объектно-ориентированная система должна обладать следующими пятью характеристиками: абстрактностью, инкапсуляцией, наследованием, полиморфизмом и динамическим связыванием. *Абстракция* — это способ представления данных на определенном уровне некоторой конкретной проблемной области. *Инкапсуляция* — это процесс, позволяющий скрывать детали реализации объекта с помощью некоторого определенного для этого класса внешнего интерфейса. *Наследование* позволяет определять классы, использующие определения других классов и обладающие всеми их и некоторыми своими свойствами, если это необходимо. *Полиморфизм* — свойство, благодаря которому различные объекты по-разному реагируют на одни и те же сообщения. *Динамическое связывание* является возможностью выбирать определенный обработчик сообщения объекта во время выполнения программы. Рассмотрим теперь, как CLIPS реализует все эти основные свойства системы ООП.

Создание нового класса реализует возможность абстрактного представления нового типа данных. Слоты и обработчики сообщений этого класса определяют свойства и поведение целой группы объектов, принадлежащих этому классу.

Инкапсуляция реализуется в CLIPS требованием обязательно использовать сообщения при работе с объектами определенных пользователем классов. Обработчики сообщений класса представляют собой доступный пользователю интерфейс, скрывающий реализацию класса.

COOL поддерживает множественное наследование. Это означает, что некоторый класс может обладать всеми свойствами указанного одного или более *суперкласса*. Для установления линейного порядка наследования свойств классов при множественном наследовании COOL использует *список предшествования классов* (class precedence list), построенный с использованием иерархии наследования. Объект, представляющий собой экземпляр нового класса, наследует все свойства (слоты) и поведение (обработчики сообщений) каждого класса из списка предшествования классов. Слово "предшествование" обозначает, что свойства и поведение класса, находящегося ближе к началу списка, переопределяют конфликтующие определения ранее встретившихся классов.

Различные COOL-объекты могут реагировать на одно и то же сообщение совершенно по-разному. Это реализует свойство полиморфизма. На практике это выполняется присоединением к разным классам обработчиков одного и того же сообщения, но с разными последовательностями выполняемых действий.

CLIPS также поддерживает возможность динамического связывания, реализуемую с помощью функции send, предназначенной для посылки сообщений объекту. Вызов этой функции осуществляется именно в процессе выполнения программы, таким образом, определение обработчика выполняющегося в тот или иной момент также происходит в процессе выполнения программы. Например, функция send может получать в качестве параметра переменную, которая в разные моменты времени ссылается на различные объекты, при этом могут вызываться совершенно разные обработчики. Подробно процесс определения нужного обработчика описан в гл. 11.

4.4.3. Запросы и наборы объектов

В дополнение к возможности использовать объекты в процессе сопоставления образцов правил, COOL поддерживает гибкую систему запросов, позволяющую использовать заданные пользователем критерии для выборки некоторого набора объектов и выполнения над ним определенных действий. Запросы позволяют объединять в наборы объекты самых разных классов. Запросы можно использовать, например, для проверки существования того или иного набора объектов, выполнения действий над набором или сохранения ссылки на набор для последующего использования. Подробное описание этой возможности COOL с примерами использования приведено в гл. 11.

ЧАСТЬ III.Основные конструкции CLIPS.

ГЛАВА 5. Факты.

Для функционирования любой экспертной системы критически важным является наличие базы знаний. Об этом говорит даже тот факт, что в последнее время все чаще термин "система, основанная на знаниях" (knowledge-base system) употребляется в качестве синонима термина "экспертная система". Как правило, в любой экспертной системе знания представляются фактами и правилами, заданными на некотором языке описания знаний. CLIPS не является исключением и предоставляет возможности для приобретения, хранения и обработки фактов и правил. Данная глава посвящена способам работы с фактами в системе CLIPS. Работа с правилами будет описана в следующей главе.

5.1. Факты в CLIPS

Факты — одна из основных форм представления данных в CLIPS (существует также возможность представления данных в виде объектов и глобальных переменных, но об этом речь пойдет позже). Каждый факт представляет собой определенный набор данных, сохраняемый в текущем списке фактов — рабочей памяти системы. Список фактов представляет собой универсальное хранилище фактов и является частью базы знаний. Объем списка фактов ограничен только памятью вашего компьютера. Список фактов хранится в оперативной памяти компьютера, но CLIPS предоставляет возможность сохранять текущий список в файл и загружать список из ранее сохраненного файла.
В системе CLIPS фактом является список неделимых (или атомарных) значений примитивных типов данных. CLIPS поддерживает два типа фактов -*упорядоченные факты* (ordered facts) и *неупорядоченные факты* или *шаблоны* (non-ordered facts или template facts). Ссылаться на данные, содержащиеся в факте, можно либо используя строго заданную позицию значения в списке данных для упорядоченных фактов, либо указывая имя значения для шаблонов.
Факты можно добавлять, удалять, изменять и дублировать, вводя соответствующие команды с клавиатуры, либо из программы. Все соответствующие команды будут описаны в данной главе.
После добавления факта в список фактов ему присваивается целый уникальный идентификатор, называемый *индексом факта* (fact-index). Индекс первого факта равен нулю, в дальнейшем индекс увеличивается на единицу при добавлении каждого нового факта. CLIPS предоставляет команды, очищающие текущий список фактов или всю базу знаний, эти команды присваивают текущему значению индекса 0.
Некоторые команды, например изменения, удаления или дублирования фактов, требуют указания определенного факта. Факт можно задать либо индексом факта, либо его адресом. Адрес факта представляет собой переменную-указатель, хранящую индекс факта. Процесс создания адресов фактов будет описан ниже.
Упорядоченные факты состоят из поля, обязательно являющимся данным типа symbol и следующей за ним, возможно пустой, последовательности полей, разделенных пробелами. Ограничением факта служат круглые скобки.

Определение 5.1. Упорядоченный факт

(данное_типа_symЬo1 [поле]*)

Первое поле факта определяет так называемое *отношение,* или *связь* факта (relation). Термин "связь" означает, что данный факт принадлежит некоторому определенному конструктором или неявно объявленному шаблону. Подробней речь об этом пойдет ниже. Приведем несколько примеров фактов:

Пример 5.1 Упорядоченные факты

```
(duck is bird)
(schoolboys is Bob Mike)
(Nuke did report)
(altitude is 1000 feet)
```

Количество полей в факте не ограничено. Поля в факте могут хранить данные любого примитивного типа CLIPS, за исключением первого поля, которое обязательно должно быть типа symbol.

Следующие слова зарезервированы и не могут быть использованы в качестве первого поля: test, and, or, not, declare, logical, object, exist и forall. Эти слова могут использоваться в качестве имен слотов шаблонов, хотя это не рекомендуется.

Так как упорядоченный факт для представления информации использует строго заданные позиции данных, то для доступа к ней пользователь должен знать не только какие данные сохранены в факте, но и какое поле содержит эти данные. Неупорядоченные факты (или шаблоны) предоставляют пользователю возможность задавать абстрактную структуру факта путем назначения имени каждому полю. Для создания шаблонов, которые впоследствии будут применяться для доступа к полям факта по имени, используется конструктор deftemplate. Конструктор deftemplate аналогичен определениям записей или структур в таких языках программирования, как Pascal или C.

Конструктор deftemplate задает имя шаблона и определяет последовательность из нуля или более полей неупорядоченного факта, называемых также *слотами.* Слот состоит из имени, заданного значением типа symbol, и следующим за ним, возможно пустого, списка полей. Как и факт, слот с обеих сторон ограничивается круглыми скобками. В отличие от упорядоченных фактов слот неупорядоченного факта может жестко определять тип своих значений. Кроме того, слоту могут быть заданы значения по умолчанию.

Замечание

Слоты не могут быть использованы в упорядоченных фактах, а в неупорядоченных фактах, в свою очередь, нельзя ссылаться на данные, используя порядок слотов.

CLIPS различает неупорядоченные факты от упорядоченных по первому полю факта. Первое поле фактов любого типа является значением типа symbol. Если это значение соответствует имени некоторого шаблона, то факт -упорядоченный. Определение неупорядоченного факта, как и упорядоченного, ограничивается круглыми скобками.

Ниже приведено несколько примеров неупорядоченных фактов:

Пример *5.2.* Неупорядоченные факты

```
(client   (name  "Joe Brown")   (id X9345A))
(point-mass   (x-velocity 100)    (y-velocity -200))
(class   (teacher  "Martha Jones")    (#-students  30)    (Room "37A")
(grocery-list   (#-of-items 3)    (items bread milk eggs))
```

Замечание

Порядок слотов *в* неупорядоченном факте не важен. Например, все приведенные ниже факты считаются идентичными:

```
(class (teacher "Martha Jones") (#-students 30) (Room "37A"))
(class (#-students 30) (teacher "Martha Jones") (Room "37A"))
(class (Room "37A") (#-students 30) (teacher "Martha Jones"))
```

В отличие от фактов, приведенных выше, упорядоченные факты из следующего примера не являются идентичными:

```
(class "Martha Jones" 30 "37A")
(class 30 "Martha Jones" "37A")
(class "37A" 30 "Martha Jones")
```

С неупорядоченными фактами можно выполнять те же операции, что и с упорядоченными.

Далее рассмотрим конструкторы, операции и функции, которые предоставляет CLIPS для работы с фактами.

5.2. Работа с фактами

CLIPS предоставляет довольно богатый набор возможностей для работы с фактами с помощью соответствующих конструкторов, операций и функций. Эти возможности включают создание шаблонов с помощью конструктора deftemplate, создание, изменение, удаление, поиск фактов, просмотр, сохранение и загрузку списка фактов, определение списка предопределенных фактов с помощью конструктора deffacts и многое другое.

5.2.1. Конструктор *deftemplate*

Для создания неупорядоченных фактов в CLIPS предусмотрен специальный конструктор deftemplate. Его использование приводит к появлению в текущей базе знаний системы информации о шаблоне факта, с помощью которого в систему в дальнейшем можно будет добавлять факты, соответствующие данному шаблону. Таким образом, конструктор deftemplate аналогичен операторам record и struct таких процедурных языков программирования как Pascal или C.

Приведем простой пример использования конструктора deftemplate:

Пример 5.3. Применение конструктора deftemplate

```
(deftemplate MyObject
             (slot name)
             (slot location)
             (slot weight)
             (multislot contents))
```

Как и все конструкторы CLIPS, конструктор deftemplate не возвращает никакого значения. При вводе данной команды в CLIPS вы должны увидеть результат, приведенный на рис. 5.1.

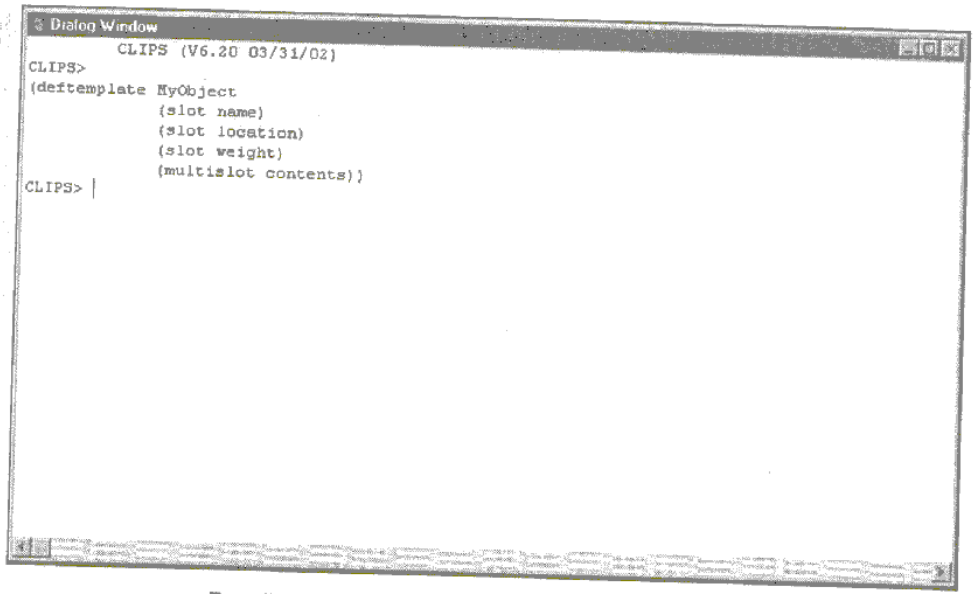

Рис. 5.1. Использование конструктора deftemplate

Подобная реакция среды говорит об удачном добавлении определения шаблона в систему. Для просмотра всех определенных в текущей базе знаний шаблонов можно воспользоваться командой get-deftemplate-list, речь о которой пойдет ниже, или специальным инструментом **Deftemplate Manager** (Менеджер шаблонов), доступным в Windows-версии среды CLIPS. Для запуска менеджера шаблонов воспользуйтесь меню **Browse** и выберите пункт **Deftemplate Manager** (рис. 5.2).

Менеджер шаблонов позволяет в отдельном окне просматривать список всех шаблонов, доступных в текущей базе знаний, удалять выбранный шаблон и отображать все его свойства (например, такие как имена и типы слотов). Внешний вид менеджера шаблонов представлен на рис. 5.3.

После выполненной нами операции в текущей базе знаний находится два шаблона, о чем сообщается в заголовке окна менеджера **(Deftemplate Manager — 2 Items).** Первый шаблон является предопределенным шаблоном initial-fact. Он не имеет слотов и всегда добавляется при запуске среды. Его нельзя удалить с помощью менеджера, или просмотреть его определение. Назначение и примеры использования факта initial-fact будут рассмотрены ниже. Вторым шаблоном является только что добавленный шаблон MyObject. Менеджер шаблонов позволяет вывести в главное окно среды его определение с помощью кнопки **Pprint** или удалить его из среды посредством кнопки **Remove.** На рис. 5.4 приведен результат последовательных операций вывода информации об определении шаблона и удалении его из текущей базы знаний.

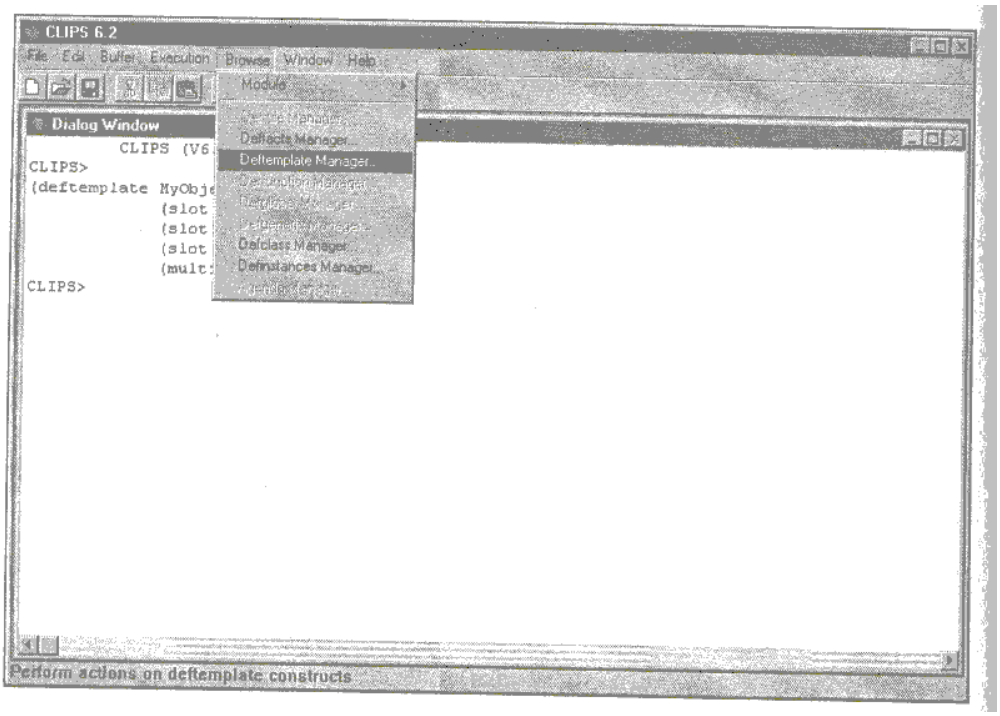

Рис. 5.2. Запуск менеджера шаблонов

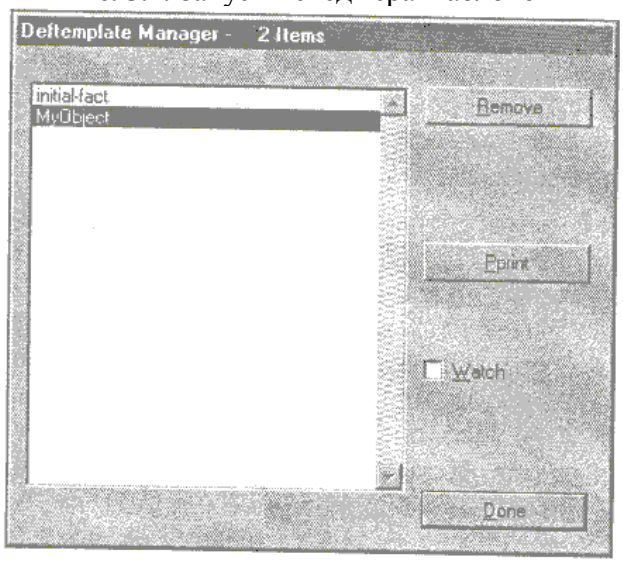

Рис. 5.3. Окно менеджера шаблонов

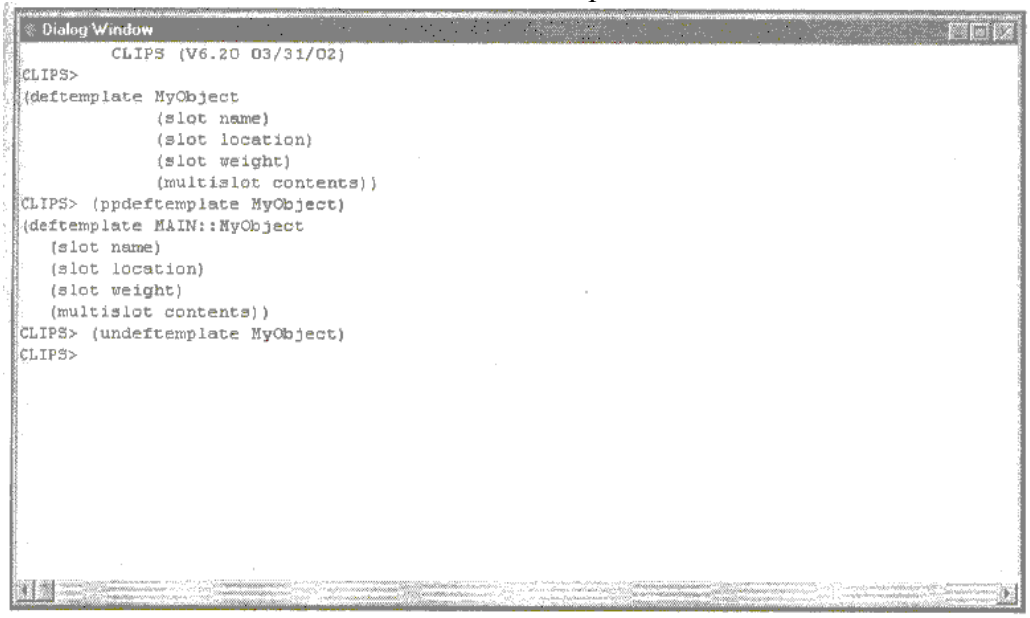

Рис. 5.4. Получение информации и удаление шаблона

Флажок **Watch** позволяет включать/выключать режим отображения сообщений об использовании шаблонов для каждого присутствующего в системе шаблона в главном окне среды CLIPS. Если этот режим включен, пользователь будет получать сообщения при добавлении и удалении неупорядоченных фактов, использующих данный шаблон.

В случае, если при добавлении нового шаблона с помощью конструктора deftemplate произошла ошибка, пользователь получит соответствующее предупреждение. Полный список сообщений об ошибках в системе CLIPS приведен в *приложениях 2* и *3*.

Переопределение уже существующего шаблона приводит к исключению предыдущего определения. Шаблон не может быть переопределен до тех пор, пока он используется (например, фактом или правилом). Шаблон может иметь любое количество простых или составных слотов. CLIPS отличает простые и составные слоты в шаблоне. Например, будет ошибкой сохранять значение составного слота в простой слот.

Рассмотрим полный синтаксис конструктора deftemplate:

Определение 5.2. Синтаксис конструктора deftemplate

(deftemplate <имя-шаблона>[<необязательные-комментарии>] [<определение-слота>*])

<определение-слота> : : = <определение-простого-слота>|<определение-составного-слота>

<определение-простого-слота> : : = (slot <имя-поля> <атрибуты-шаблона>)

<определение-составного-слота> ::= (multislot <имя-поля> <атрибуты-шаблона>)

<атрибуты-шаблона> ::= <атрибут-значение-по-умолчанию>|<атрибут-ограничения>

<атрибут-значение-по-умолчанию> ::= (default ?DERIVE I ?NONE |<Выражение>)|

(default-dynamic <Выражение>)

Заметим еще раз, что имена шаблонов и слотов должны быть значениями типа symbol, кроме того, на имена шаблонов распространяется запрет на использование некоторых слов, зарезервированных системой, перечисленных выше.

Комментарии являются необязательными и, как правило, описывают назначения шаблона. Комментарии необходимо заключать в кавычки. Кроме данного типа комментариев в конструкторе deftempiate также применимы обычные комментарии CLIPS, начинающиеся с символа ; . Отличие этих комментариев заключается в том, что комментарии, начинающиеся с символа ; , полностью игнорируются системой CLIPS, а комментарии, следующие после имени шаблона и заключенные в кавычки, сохраняются в базе знаний системы. Эти комментарии доступны при просматривании определения шаблона. Определим в среде CLIPS следующий шаблон:

Пример 5.4. Применение конструктора deftemplate

```
(deftemplate MyObject "Template for storage name and location"
                ; Slots  for storage name and location
                    (slot name) ;slot for name of object
                    (slot location) ; slot for location of object
```

После удачного добавления шаблона в систему, с помощью менеджера шаблонов, выведите в главное окно CLIPS информацию об определении шаблона MyObject. Если перечисленные действия были выполнены без ошибок, то на экране должны появиться сообщения, идентичные показанным на рис. 5.5.

Обратите внимание, что комментарии *"Template for storage name and location"* сохранены в памяти системы и отображаются вместе с определением шаблона. К сожалению, текущая версия CLIPS не воспринимает символы кириллицы даже в качестве комментариев, поэтому все комментарии придется давать на английском языке.

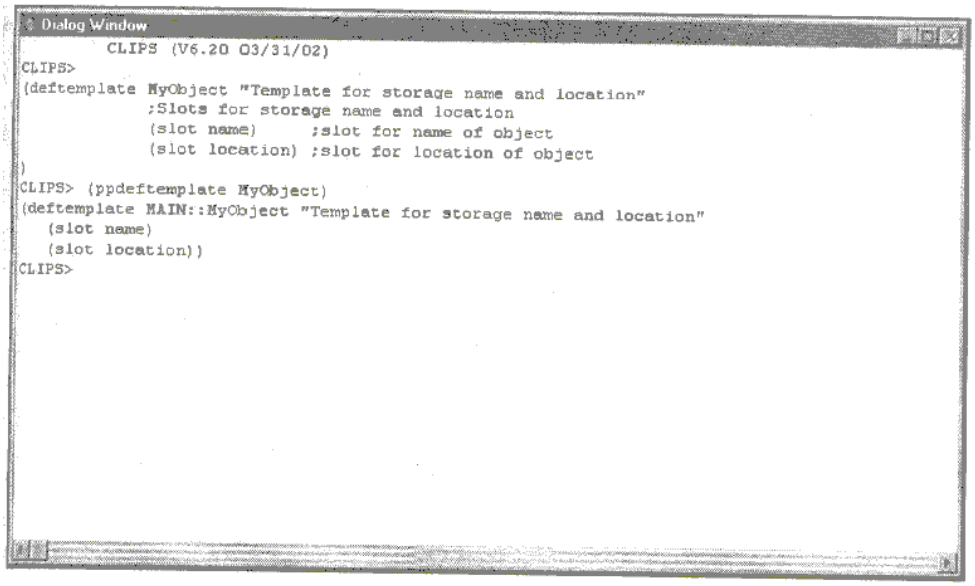

Рис. 5.5. Использование комментариев в конструкторе deftemplate

Помимо ключевого слова slot, определяющего простой слот, допустимо также применение ключевого слова multisiot, для определения составного слота. Простой слот, или слот, предназначен для хранения единицы информации одного из примитивных типов данных CLIPS. Составной слот способен хранить список подобных единиц информации неограниченного объема. Для доступа к конкретным данным, хранящимся в составном слоте, используются специальные групповые символы и функции, примеры и правила использования которых будут приведены ниже.

При создании шаблона с помощью конструктора deftemplate каждому полю можно назначать определенные атрибуты, задающие значения по умолчанию или ограничения на значение слота. Рассмотрим эти атрибуты подробней.

<Атрибут-значение-по-умолчанию> определяет значение, которое будет использовано в случае, если при создании факта не задано конкретное значение слота. В CLIPS существует два способа определения значения по умолчанию, поэтому в конструкторе deftemplate предусмотрено два различных атрибута, задающих значения по умолчанию: default и default-dynamic.

Атрибут default определяет статическое значение по умолчанию. С его помощью задается выражение, которое вычисляется один раз при конструировании шаблона. Результат вычислений сохраняется вместе с шаблоном. Этот результат присваивается соответствующему слоту в момент объявления нового факта. В случае если в качестве значения по умолчанию используется ключевое слово ?DERIVE, то это значение будет извлекаться из ограничений, заданных для данного слота. По умолчанию для всех слотов установлен атрибут default ?DERIVE.

В случае если в место выражения для значения по умолчанию используется ключевое слово ?NONE, то значение поля обязательно должно быть явно задано в момент выполнения операции добавления факта. Добавление факта без определения значений полей с атрибутом default ?NONE вызовет ошибку.

Атрибут default-dynamic предназначен для установки динамического значения по умолчанию. Этот атрибут определяет выражение, которое вычисляется всякий раз при добавлении факта по данному шаблону. Результат вычислений присваивается соответствующему слоту.

Простой слот может иметь только одно значение по умолчанию. У составного слота может быть определено любое количество значений по умолчанию (количество значений по умолчанию должно соответствовать количеству данных, сохраняемых в составном слоте).

Ниже приведен пример использования атрибута, устанавливающего значение по умолчанию:

Пример 5.5. Использование атрибутов значения по умолчанию

```
(deftemplate foo
              (slot w   (default  ?NONE))
              (slot x   (default  ?DERIVE))
              (slot y   (default   (gensym*)))
              (slot z   (default-dynamic   (gensym*))))
```

Синтаксис и функциональность <атрибута-ограничения> для простого и составного слота детально описаны в *гл. 13*. В конструкторе deftemplate поддерживается проверка статических и динамических ограничений.

Статическая проверка выполняется во время использования определения шаблона некоторой командой или конструктором. Например, для записи значений в слоты шаблона. Иначе говоря, статическая проверка выполняется до запуска программы. При несоответствии используемых значений с установленными ограничениями пользователю выводится соответствующее предупреждение об ошибке.

Ссылка на индекс факта в командах на изменение значения факта или его дублирование не связывает факт с соответствующим шаблоном явно. Это делает статическую проверку неоднозначной. Поэтому в командах, использующих индекс факта, статическая проверка не выполняется.

Статическая проверка ограничений включена по умолчанию. Эту установку среды CLIPS можно изменить с помощью функции set-static-constraint-checking.

Помимо статической, CLIPS также поддерживает динамическую проверку ограничений. Если режим динамической проверки ограничений включен, го все новые факты, созданные с использованием некоторого шаблона и имеющие определенные значения, проверяются в момент их добавления в список фактов.

В случае если нарушение заданных ограничений произойдет в момент выполнения динамической проверки в процессе выполнения программы, то выполнение программы прекращается и пользователю будет выдано соответствующее сообщение.

По умолчанию в CLIPS отключен режим динамической проверки ограничений. Эту среду установки можно изменить с помощью функции set-dynamic-constraint-checking.

Помимо описанных выше функций для изменения состояний режимов статической и динамической проверки ограничений, пользователям Windows-версии среды CLIPS доступен визуальный способ настройки этих установок. Для этого необходимо открыть диалоговое окно **Execution Options**, выбрав пункт **Options** из меню **Execution**. Внешний вид этого диалогового окна приведен на рис. 5.6.

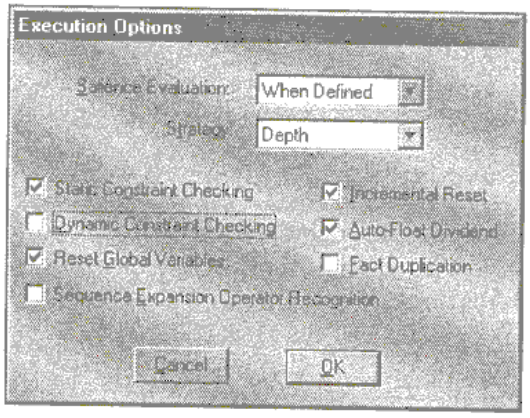

Рис. 5.6. Диалоговое окно **Execution Options**

Для включения или выключения необходимых режимов проверки ограничений атрибутов выставите в соответствующее положение флажки **Static Constraint Checking** и **Dynamic Constraint Checking** и нажмите кнопку **ОК**.

Ниже приведен пример использования атрибутов ограничения типа:

Пример 5.6. Использование атрибутов ограничения

```
(deftemplate object
        (slot name
            (type SYMBOL)
            (default  ?DERIVE))
        (slot location
            (type SYMBOL)
            (default ?DERIVE)))
```

Для полноты картины следует также упомянуть о неявно создаваемых шаблонах. При использовании факта или ссылки на упорядоченный факт (например, в правиле) CLIPS неявно создает соответствующий шаблон с одним составным слотом. Имя неявно созданного составного слота не отображается при просмотре фактов. Неявно созданным шаблоном можно манипулировать и сравнивать его с любым тождественным, определенным пользователем шаблоном, несмотря на то, что он не имеет отображаемой формы.

5.2.2. Конструктор *deffacts*

Помимо конструктора deftemplates, CLIPS предоставляет конструктор deffacts, также предназначенный для работы с фактами. Данный конструктор позволяет определять список фактов, которые будут автоматически добавляться всякий раз после выполнения команды reset, очищающей текущий список фактов. Факты, добавленные с помощью конструктора deffacts, могут использоваться и удаляться так же, как и любые другие факты, добавленные в базу знаний пользователем или программой, с помощью команды assert.

Определение 5.3. Синтаксис конструктора deffacts

(deffacts <имя-списка-фактов> [<необязательные-комментарии>]
[<факт>*])

Добавление конструктора deffacts с именем уже существующего конструктора приведет к удалению предыдущего конструктора, даже если новый конструктор содержит ошибки. В среде CLIPS возможно наличие нескольких конструкций deffacts одновременно и любое число фактов в них (как упорядоченных, так и неупорядоченных). Факты всех созданных пользователем конструкторов deffacts будут добавлены при инициализации системы.

Все замечания по поводу использования комментариев в конструкторе deftemplate применимы и к конструктору deffacts.
В поля факта могут быть включены динамические выражения, значения которых будут вычисляться при добавлении этих фактов в текущую базу знаний CLIPS.

Пример 5.7. Использование конструктора deffacts

```
(deffacts   startup "Refrigerator Status"
            (refrigerator light on)
            (refrigerator door open)
            (refrigerator temp (+ 5 10 15))
```

Обратите внимание, что третий факт содержит выражение, в данном примере сумму трех констант, но в качестве выражения, инициализирующего значение факта, могут использоваться и более сложные выражения, например, вызовы функций CLIPS или функций, определенных пользователем.]
Проверить работу конструктора deffacts можно воспользовавшись диалогом **Watch Options.** Для этого выберите пункт **Watch** меню **Execution** или используйте комбинацию клавиш <Ctrl>+<W>. В диалоговом окне **Watch Options** включите режим просмотра изменения списка фактов, поставив галочку в поле **Facts,** как показано на рис. 5.7.

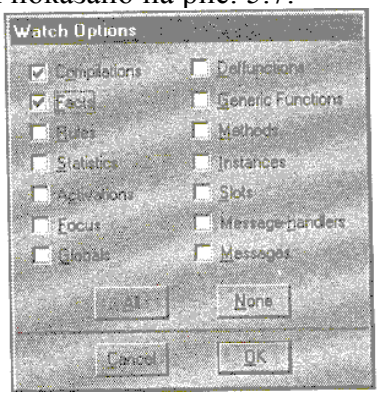

Рис. 5.7. Диалоговое окно **Watch Options**

После этого нажмите кнопку OK и введите в CKIOS приведенный выше конструктор deffacts. Затем в меню **Execution** выберите пункт **Reset** (комбинация клавиш <Ctrl>+<E>). Если пример был набран правильно, то на экране должны появиться сообщения, аналогичные приведенным на рис. 5.8.

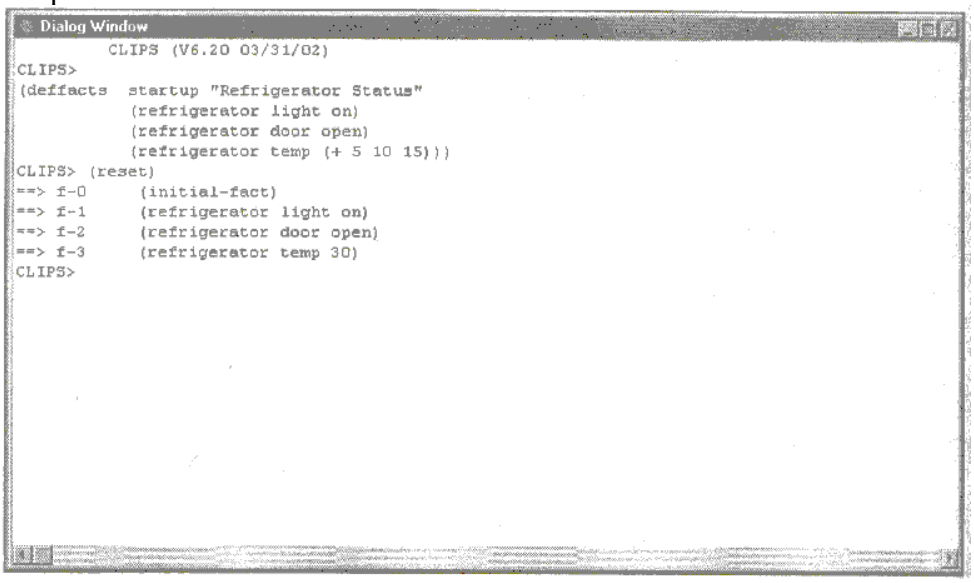

Рис. 5.8. Просмотр процесса добавления файлов

Так же, как и для конструкторов deftemplate, CLIPS предоставляет визуальный инструмент для манипуляции с определенными в данный момент в системе конструкторами deffacts -- **Deffacts Manager** (Менеджер предопределенных фактов). Для запуска **Deffacts Manager** в меню **Browse** выберите пункт **Deffacts Manager.** Внешний вид менеджера приведен на рис. 5.9.

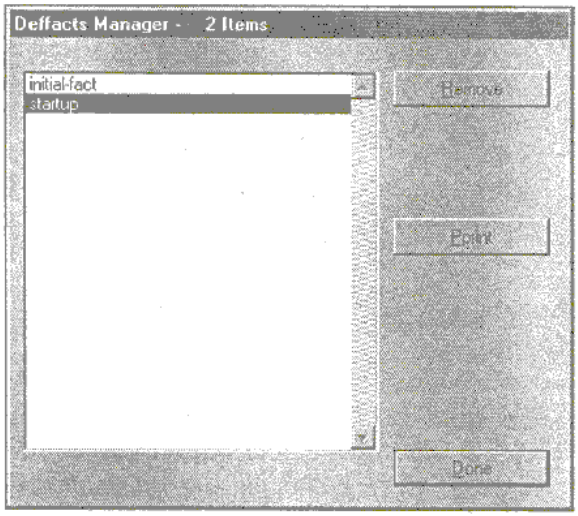

Рис. 5.9. Окно менеджера предопределенных фактов

Менеджер отображает все введенные на текущий момент в систему конструкторы deffacts. В нашем случае это initial-fact, речь о котором пойдет ниже, и только что добавленный нами startup. Менеджер позволяет выводить в основное окно CLIPS информацию об определениях существующих в данный момент в системе конструкторов deffacts с помощью кнопки **Pprint** (кроме deffacts initial-fact) и удалять любой существующий конструктор. Пример вывода информации об определении конструктора deffacts startup приведен на рис. 5.10. Обратите внимание, что комментарии, введенные после имени конструктора, сохраняются и выводятся на экран так же, как в конструкторе deftemplate.

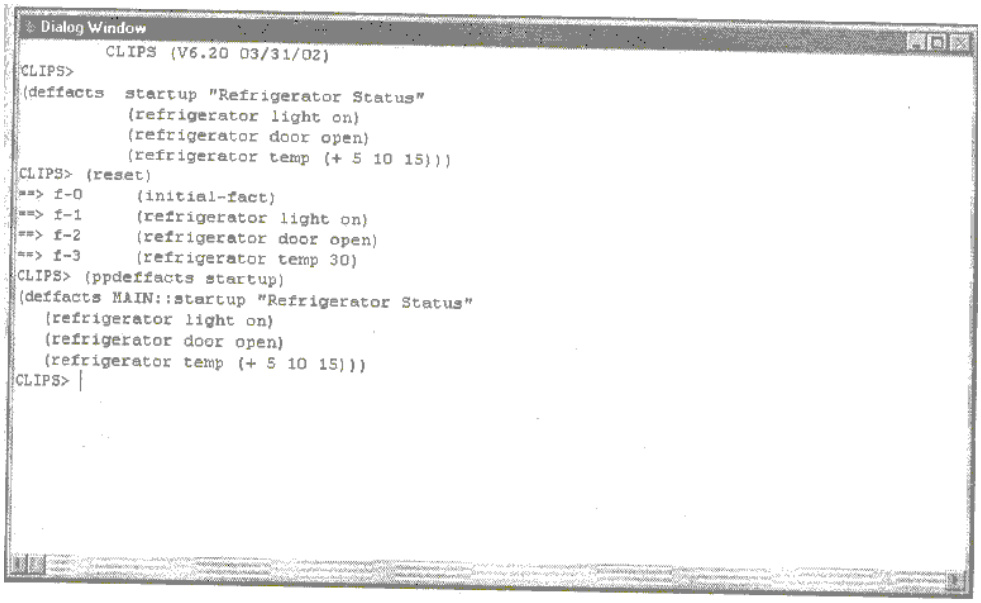

Рис. 5.10. Получение информации об определенном конструкторе

Во время запуска и после выполнения команды clear CLIPS автоматически конструирует следующие предопределенные шаблоны и факты:

Определение 5.4. Предопределенные шаблоны и факты

```
(deftemplate   initial-fact)
(deffacts      initial-fact
               (initial-fact))
```

Предопределенный факт initial-fact шаблона initial-fact предоставляет удобный способ для запуска программ на языке CLIPS — правила, не имеющие условных элементов, автоматически преобразуются в правила с условием, проверяющим наличие факта initial-fact. Факт initial-fact можно обрабатывать так же, как и все остальные факты CLIPS, добавленные пользователем или программой с помощью команды assert. Пример использования факта initial-fact будет приведен в следующей главе, сразу после первого знакомства с правилами CLIPS.

5.2.3. Функция *assert*

Функция assert - одна из наиболее часто применимых команд в системе CLIPS. Без использования этой команды нельзя написать даже самую простую экспертную систему и запустить ее на выполнение в среде CLIPS. Функции Assert, retract и modify — три рабочие лошадки, используемые большинством правил.

Функция assert позволяет добавлять факты в список фактов текущей базы знаний. Каждым вызовом этой функции можно добавить произвольное число фактов. В случае если был включен режим просмотра изменения списка фактов (как было описано в *разд. 5.2.2),* то соответствующее информационное сообщение будет отображаться в окне CLIPS при добавлении каждого факта.

Определение 5.5. Синтаксис команды assert

```
(assert        <факт>+)
```

При использовании команды assert необходимо помнить, что первое поле факта обязательно должно быть значением типа symbol. В случае удачного добавления фактов в базу знаний, функция возвращает адрес последнего добавленного факта. Если во время добавления некоторого факта произошла ошибка, команда прекращает свою работу и возвращает значение FALSE.

Слотам неупорядоченного факта, значения которых не заданы, будут присвоены значения по умолчанию *(см. разд. 5.2.1).*

Пример 5.8. Использование функции assert

```
(clear)
(assert (color red))
(assert (color blue)
        (value (+ 3 4)))
(deftemplate status
                (slot temp)
                (slot pressure
                (default low)))
(assert (status (temp high)))
```

Команда clear очищает текущий список фактов (а также все определенные конструкторы, которые уже были и еще будет рассмотрены ниже). В отличие от reset, команда clear не добавляет в список фактов initial-fact. Эту команду также можно выполнить, выбрав пункт **Clear CLIPS** в меню **Execution.** При выборе данной команды на экране появляется диалоговое окно, представленное на рис. 5.11. Это окно запрашивает подтверждение пользователя на очистку текущей базы знаний.

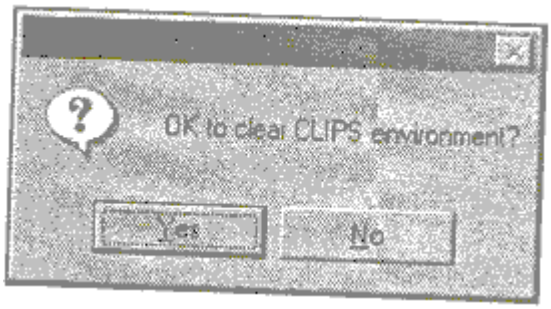

Рис. 5.11. Подтверждение очистки среды CLIPS

В случае, если команда была набрана с клавиатуры, никакого подтверждения на выполнение этой операции система не запрашивает. Если вы недавно начали работать в среде CLIPS, то для очистки системы лучше использовать меню, т. к. потеря всех текущих данных из базы знаний может оказаться весьма болезненной.

Включите режим просмотра изменения списка фактов и наберите приведенный выше пример. После этого выполните команду (facts). Если при

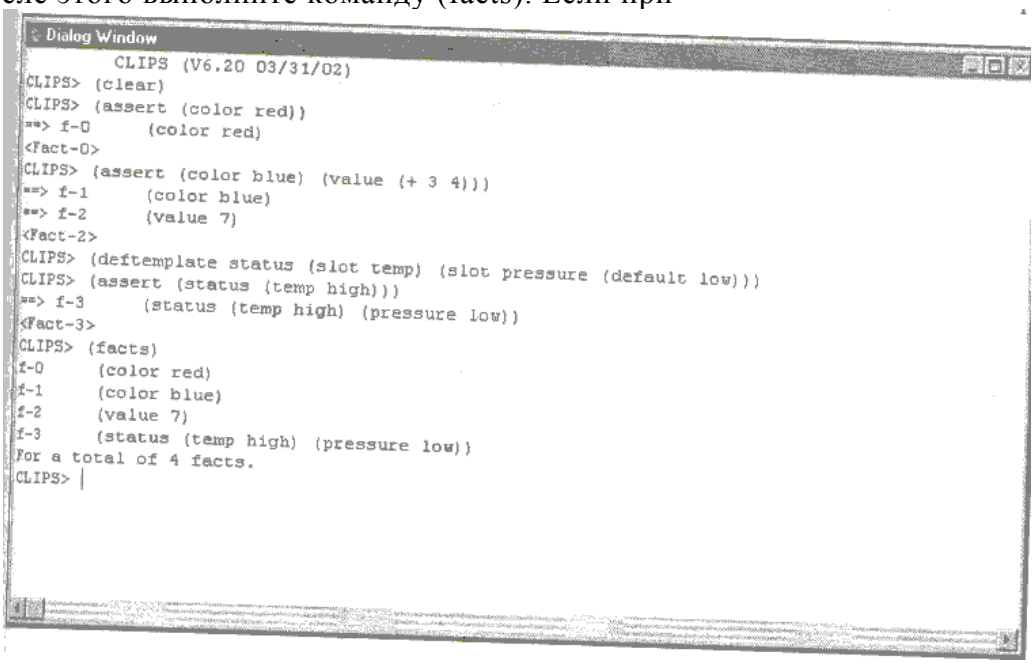

Рис. 5.12. Добавление фактов

выполнении этих действий не было допущено ошибок, то вы должны получить результат, идентичный изображенному на рис. 5.12.

Обратите внимание, что при инициализации факта value использовалось выражение, а слот pressure неупорядоченного факта status получил значение по умолчанию low.

По умолчанию CLIPS не позволяет добавлять в список фактов два одинаковых факта. Например, попытка добавить два факта color red приведет к ошибке и функция assert вернет значение FALSE. Данную установку системы можно изменить с помощью функции set-fact-duplication. Кроме того, пользователям Windows-версии CLIPS доступен еще один способ настройки. Для этого необходимо открыть диалоговое окно **Execution Options,** выбрав пункт **Options** из меню **Execution,** установить флажок **Fact Duplication.** Внешний вид этого диалогового окна приведен на рис. 5.6.

5.2.4. Функция *retract*

После добавления факта в базу знаний рано или поздно встанет вопрос о том, как его оттуда удалить. Для удаления фактов из текущего списка фактов в системе CLIPS предусмотрена функция retract. Каждым вызовом этой функции можно удалить произвольное число фактов. Удаление некоторого факта может стать причиной удаления других фактов, которые логически связаны с удаляемым. Кроме того, удаление факта вызывает удаления правил из *плана решения текущей задачи,* активированных удаляемым фактом, но об этом речь пойдет в следующих главах. В случае если был включен режим просмотра изменения списка фактов, то соответствующее информационное сообщение будет отображаться в окне CLIPS при удалении каждого факта.

Определение 5.6. Синтаксис команды retract

(retract <определение-факта>+ \ *)

Аргумент <определение-факта> может являться либо переменной, связанной с адресом факта с помощью правила (эта возможность будет описана в следующей главе), либо индексом факта без префикса (например, 3 для факта с индексом f-3), либо выражением, вычисляющим этот индекс (например, (+ 1 2) для факта с индексом f-3). Если в качестве аргумента функции retract использовался символ *, то из текущей базы знаний системы будут удалены все факты. Функция retract не имеет возвращаемого значения.

Для демонстрации работы функции retract воспользуемся еще одним визуальным инструментом, не описанным ранее. Он предназначен для просмотра содержимого списка фактов в реальном времени. Этот инструмент доступен только пользователям Windows-версии системы CLIPS. Для того чтобы активизировать просмотр списка фактов, поставьте флажок рядом с пунктом **Facts Window** меню **Windows,** как показано на рис. 5.13. Внешний вид инструмента просмотра списка фактов показан на том же рисунке. Сразу после запуска CLIPS этот список пуст.

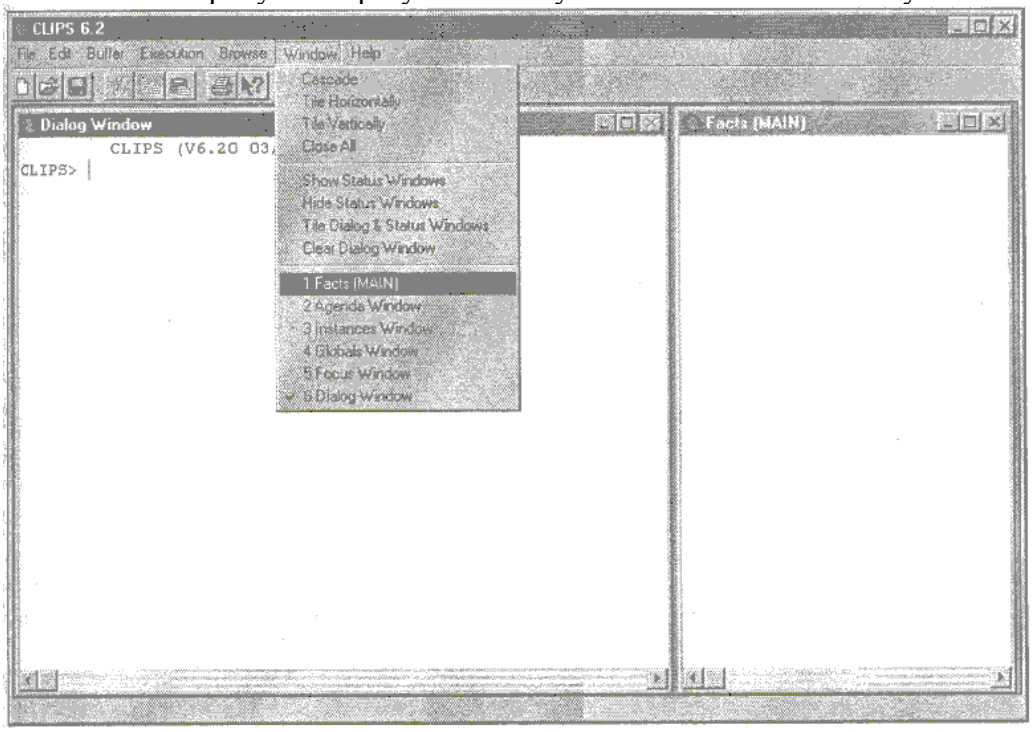

Рис. 5.13. Список фактов

Включите режим просмотра изменения списка фактов с помощью диалогового окна **Watch Options** и добавьте в список фактов следующие факты:

Пример 5.9. Добавление фактов

```
(assert  (a)  (b)  (c)  (d)  (e)  (f))
```

Обратите внимание, что в окне просмотра фактов теперь отображаются все 6 добавленных фактов.

Пример 5.10. Удаление фактов

```
(retract  0  (+  0  2)  (+ 0 2 2))
```

Эта команда удалит все факты с четными индексами, используя индекс факта непосредственно (первый аргумент) и выражение, которое вычисляет индекс факта (второй и третий аргумент). Если перечисленные команды были выполнены правильно, то результат должен соответствовать рис. 5.14.

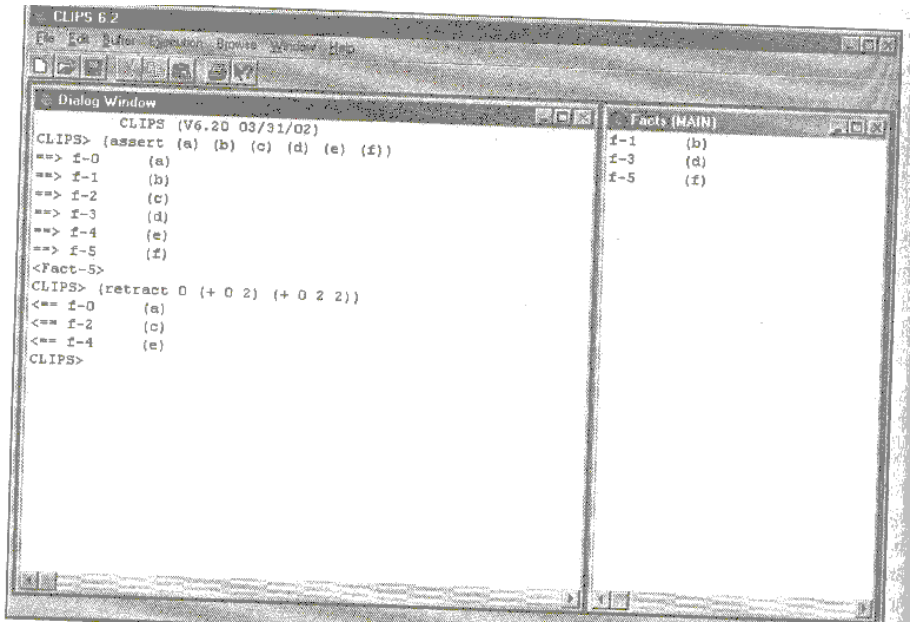

Рис. 5.14. Результат добавления и удаления фактов

В случае, если факт с указанным индексом не будет найден, CLIPS выдаст соответствующее сообщение об ошибке.

Выполните команду:

Пример 5.11. Удаление всех фактов

```
(retract *)
```

После выполнения данной команды список фактов будет очищен полностью и окно отображения текущего состояния списка фактов станет идентично изображенному на рис. 5.13.

Необходимо заметить, что функция retract не оказывает никакого воздействия на индекс следующих добавленных фактов, т. е. этот индекс не обнуляется. Если после удаления всех введенных фактов добавить в систему какой-нибудь факт, то он получит индекс f-6, несмотря на то, что список фактов в данный момент пуст.

5.2.5. Функция *modify*

Используя функции assert и retract, можно выполнять большинство необходимых для функционирования правил действий. В том числе и изменения существующего факта. Например,

если в список фактов ранее был добавлен факт (temperature is low), который получил индекс 0, то изменить его значение можно, например, следующим образом:

Пример 5.12. Изменение существующего факта

```
(clear)
(assert   (temperature is  low) )
(retract  0)
(assert   (temperature is high) )
```

Для изменения упорядоченных фактов доступен только этот способ. Для упрощения операции изменения неупорядоченных фактов CLIPS предоставляет функцию modify, которая позволяет изменять значения слотов таких фактов. Modify просто упрощает процесс изменения факта, но ее внутренняя реализация эквивалентна вызовам пар функций retract и assert. За один вызов modify позволяет изменять только один факт. В случае удачного выполнения функция возвращает новый индекс модифицированного факта. Если в процессе выполнения произошла какая-либо ошибка, то пользователю выводится соответствующее предупреждение и функция возвращает значение FALSE.

Определение 5.7. Синтаксис команды modify

```
(modify      <определение-факта>
             <новое-значение-слота>+)
```

Аргументом <определение-факта> может быть либо переменная, связанная с адресом факта с помощью правила, либо индекс факта без префикса (например, 3 для факта с индексом f-3). После определения факта следует список из одного или более новых значений слотов указанного шаблона. Для использования приведенного выше примера его необходимо переделать следующим образом:

Пример 5.13. Изменение существующего неупорядоченного факта

```
(deftemplate temperature   (slot value) )
(assert   (temperature   (value low)) )
(modify 0   (value high) )
```

Если включить режим просмотра изменения списка фактов (как было описано в *разд. 5.2.2)* и выполнить приведенные выше команды, то полученный результат должен соответствовать рис. 5.15.

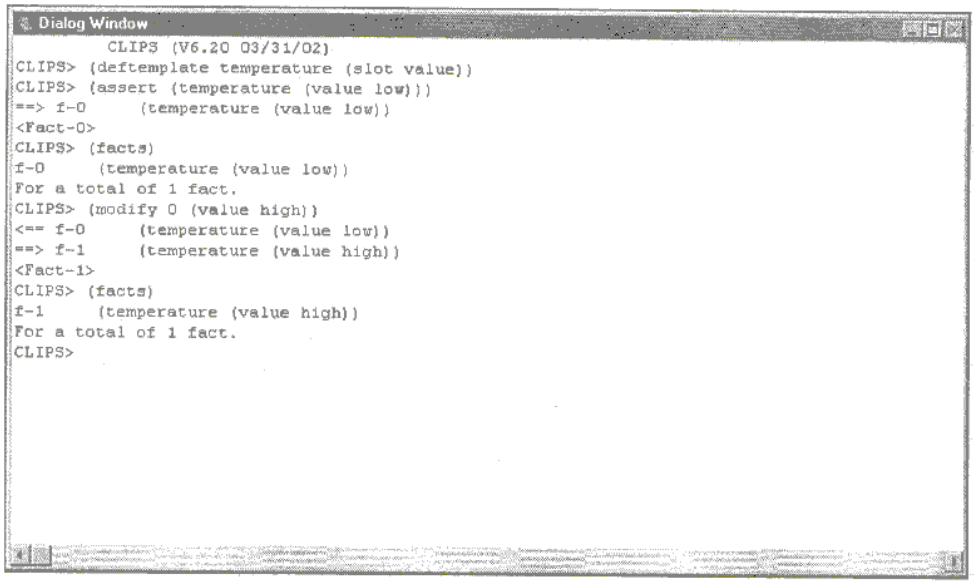

Рис. 5.15. Результат изменения существующего неупорядоченного факта

Обратите внимание на движение фактов в базе знаний CLIPS при выполнении функции modify — сначала удаляется старый факт с индексом f-o, а затем добавляется новый факт с индексом f-1, идентичный предыдущему, но с новым значением заданного слота.

Если в шаблоне заданного факта отсутствует слот, значение которого требуется изменить, CLIPS выведет соответствующее сообщение об ошибке. Если заданный факт отсутствует в списке фактов, пользователь также получит соответствующее предупреждение.

5.2.6. Функция *duplicate*

Помимо функции modify, в CLIPS существует еще одна очень полезная функция, упрощающая работу с фактами, — функция duplicate. Эта функция создает новый неупорядоченный факт заданного шаблона и копирует в него определенную пользователем группу полей уже существующего факта того же шаблона. По действиям, которые выполняет функция duplicate, аналогична modify, за исключением того, что она не удаляет старый факт из списка фактов. Одним вызовом функции duplicate можно создать одну копию некоторого заданного факта. Как и функция modify, duplicate, в случае удачного выполнения, возвращает индекс нового факта, а в случае неудачи — значение FALSE.

Определение 5.8. Синтаксис команды duplicate

```
(duplicate     <определение-факта>
               <новое-значение-слота>+)
```

Аргумент <определение-факта> может быть либо переменной, связанной с адресом факта с помощью правила, либо индексом факта без префикса. После определения факта следует список из одного или более новых значений слотов указанного шаблона. Продемонстрируем работу данной функции на следующем примере:

Пример 5.14. Создание копии существующего неупорядоченного факта

```
(deftemplate car
                    (slot name)
                    (slot producer)
                    (slot type)
                    (slot max-speed))
(assert ( car
                    (name scorpio)
                    (producer ford)
                    (type sedan)
                    (max-speed 180)))
(duplicate 0
                    (type off-road)
                    (max-speed 130))
```

В приведенном примере определяется шаблон, описывающий свойства автомобиля, и добавляется факт — автомобиль Ford Scorpio с типом кузова седан и максимальной скоростью 180 (км/ч). После этого с помощью функции duplicate добавляется факт с информацией об еще одном автомобиле с похожими характеристиками — это внедорожник Ford Scorpio с максимальной скоростью 130 (км/ч). Duplicate просто облегчает нам жизнь, избавляя от излишнего ввода значений данных совпадающих слотов.

В случае, если добавляемый с помощью duplicate факт уже присутствует в списке фактов, будет выдана соответствующая информация об ошибке и возвращено значение FALSE. Факт при этом добавлен не будет. Это поведение можно изменить, разрешив существование одинаковых фактов в базе знаний. Как это сделать, было описано в *разд. 5.2.3.*

5.2.7. Функция *assert-string*

Кроме функции assert, CLIPS предоставляет еще одну функцию, полезную при добавлении фактов, — assert-string. Эта функция принимает в качестве единственного аргумента символьную строку, являющуюся текстовым представлением факта (в том виде, в котором вы набираете его, например, в функции assert), и добавляет его в список фактов. Функция assert-string может работать как с упорядоченными, так и с неупорядоченными фактами. Одним вызовом функции assert-string можно добавить только один факт.

Определение 5.9. Синтаксис команды assert-string

(assert-string <строковое-выражение>)

Строковое выражение должно быть заключено в кавычки. Функция преобразует заданное строковое выражение в факт CLIPS, разделяя отдельные слова на поля, с учетом определенных в системе на текущий момент шаблонов. Если в строке необходимо записать внутреннее строковое выражение, представляющее, скажем, некоторое поле, то для включения в строковое выражение символа кавычек используется *обратная косая черта* (backslash). Например, факт (book-name "CLIPS user Guide") можно добавить следующим образом:

Пример 5.15. Использование кавычек внутри строки

```
(assert-string  "(book-name  V'CLIPS User Guide\")")
```

Для добавления содержащегося в поле символа обратной косой черты используйте ее дважды. Если обратная косая должна содержаться внутри подстроки, ее необходимо использовать четыре раза. Например, для помещения в текущий список факта (a\b "c\d") необходимо вызвать функцию assert-string со следующим строковым аргументом:

Пример 5.16. Использование обратной косой черты

```
(assert-string "(a\\b \"c\\\\d\")")
```

Если добавления факта прошло удачно, функция возвращает индекс только что добавленного факта, в противном случае функция возвращает сообщение об ошибке и значение FALSE. Функция assert-string не позволяет добавлять факт в случае, если такой факт уже присутствует *в* базе знаний (если вы еще не включили возможность присутствия одинаковых фактов).

5.2.8. Функция *fact-existp*

В этом разделе рассмотрим очень простую, но чрезвычайно важную функцию fact-existp. Эта функция определяет, присутствует ли в данный момент факт, заданный индексом или переменной указателем, в базе знаний системы. В случае если факт присутствует в списке фактов, функция возвращает значение TRUE, иначе — FALSE.

Определение 5.10. Синтаксис команды fact-existp

(fact-existp <определение-факта>)

Обычно эта функция применяется в правилах, описанных в следующей главе. Приведем простой пример использования данной функции:

Пример 5.17. Использование функции fact-existp

```
(clear)
(assert-string  "(a\\b  \"c\\\\d\")")
```

```
(fact-existp  0)
(retract  0)
(fact-existp  0)
```

Замечание

Не забудьте выполнить команду clear, чтобы добавленный факт имел нулевой индекс. После первого вызова функция fact-exist вернет значение TRUE, а после удаления факта с индексом 0 — FALSE.

5.2.9. Функции для работы с неупорядоченными фактами

Для работы с неупорядоченными фактами в CLIPS предусмотрен целый ряд специальных функций. К ним относятся: fact-relation, fact-slot-names и fact-slot-value. Рассмотрим эти функции по порядку.

Функция fact-relation позволяет получить *связь* (relation) существующего факта с шаблоном. Связь факта с шаблоном, определенным с помощью конструктора deftemplate или неявно созданным шаблоном, определяется по первому полю факта. Это поле всегда является простым полем и используется CLIPS в качестве имени шаблона, с которым связан факт. Таким образом, функция fact-relation просто возвращает первое поле факта, или значение FALSE, если указанный факт не найден.

Определение 5.11. Синтаксис команды fact-relation

(fact-relation <определение-факта>)

В качестве определения факта, как и в описанных выше функциях, нужно использовать или переменную указатель, содержащую адрес факта, или индекс факта.

Пример 5.18. Использование функции fact-relation

```
(clear)
(assert (car Ford))
(fact-relation 0)
(retract 0)
(fact-relation 0)
```

В первом случае функция fact-relation вернет значение car, а во втором —FALSE.

Для получения имен всех слотов заданного факта в CLIPS предназначена функция fact-slot-names.

Определение 5.12. Синтаксис команды fact-slot-names

(fact- slot-names <определение-факта>)

Данная функция возвращает список имен слотов в составном поле. Для упорядоченных фактов функция возвращает значение implied (подразумеваемый), т. к., если вы помните, CLIPS представляет упорядоченные факты как неявно заданные неупорядоченные с одним составным слотом. В случае если заданный факт не найден, функция возвращает значение FALSE.

Пример 5.19. Использование функции fact-slot-names

```
(clear)
(deftemplate  car
                    (slot name)
                    (slot producer)
                    (slot type)
```

```
                (slot max-speed))
(assert (   car
                (name scorpio)
                (producer ford)
                (type sedan)
                (max-speed 180)))
(fact-slot-names 0)
```

Если приведенный пример был набран без ошибок, то функция fact-slot-names вернет значение (name producer type max-speed).

Последней из рассмотренных в данной главе функций для работы с неупорядоченными фактами будет функция fact-slot-value.

Определение 5.13. Синтаксис команды fact-slot-value

(fact-slot-value <определение-факта> <имя-слота >)

Данная функция позволяет получать значения слота некоторого заданного факта. Если факт является упорядоченным, то для получения значения неявно определенного составного слота используется значение implied. В случае если указанный факт не существует, или имя слота указано не верно, функция возвращает значение FALSE.

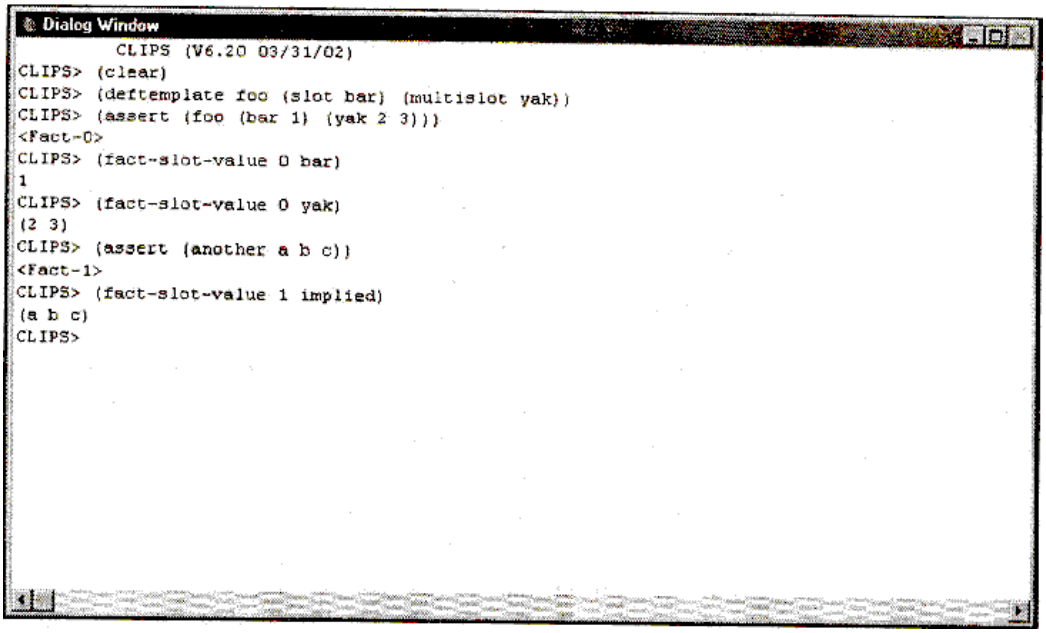

Рис. 5.16. Результат использования функции fact-slot-value

Выполните в среде CLIPS следующий пример:

Пример 5.20. Использование функции fact-slot-value

```
(clear)
(deftemplate foo
                (slot bar)
                (multislot yak)
(assert (foo (bar 1) (yak 23)))
(fact-slot-value 0 bar)
(fact-slot-value  0  yak)
(assert   (another a b c))
(fact-slot-value  1  implied)
```

Если предыдущий пример был выполнен без ошибок, то полученный результат должен соответствовать приведенному на рис. 5.16.

5.2.10. Функции сохранения и загрузки списка фактов

Как можно заметить, наполнение списка фактов в CLIPS довольно кропотливое и длительное занятие. Если фактов достаточно много, этот процесс может растянуться на несколько часов, или даже дней. Так как список фактов хранится в оперативной памяти компьютера, теоретически, из-за сбоя компьютера или, например, неожиданного отключения питания, список фактов можно безвозвратно потерять. Чтобы этого не произошло, а так же для того чтобы сделать работу по наполнению базы знаний фактами более удобной, CLIPS предоставляет команды сохранения и загрузки списка фактов в файл — save-facts и load-facts соответственно.

Определение 5.14. Синтаксис команды save-facts

(save-facts <имя-файла> [<границы-видимости> <список-шаблонов >])
<границы-видимости> ::= visible|local

Команда save-facts сохраняет факты из текущего списка фактов в текстовый файл. На каждый факт отводится одна строчка. Неупорядоченные факты сохраняются вместе с именами слотов. В функции существует возможность ограничить область видимости сохраняемых фактов. Для этого используется аргумент <границы-видимости>. Он может принимать значение local или visible. В случае если этот аргумент принимает значение visible, то сохраняются все факты, присутствующие в данный момент в системе. Если в качестве аргумента используется ключевое слово local, то сохраняются только факты из текущего модуля. О модулях речь пойдет в *гл. 12*. По умолчанию аргумент <границы-видимости> принимает значение local. После аргумента <границы-видимости> может следовать список определенных в системе шаблонов. В этом случае будут сохранены только те факты, которые связаны с указанными шаблонами.

Пример 5.21. Использование функции save-facts

```
(clear)
(deftemplate template
                        (slot a)
                        (slot b))
(assert  (template  (a  1)  (b  2)))
(assert  (simple-factl)  (simple-fact2))
(save-facts  fl  local  template simple-factl)
```

Последовательность действий, приведенная в данном примере, сохраняет в файл fl, находящийся в текущем каталоге, все факты, видимые в текущем модуле и связанные с шаблонами template и simple-factl (как вы помните, после добавления факта simple-factl CLIPS определяет неявно созданный шаблон simple-factl). В результате будет получен текстовый файл со следующим содержанием:

Пример 5.22. Содержание файла fl

```
(template  (a  1)  (b  2) } (simple-factl)
```

В случае успешного выполнения, команда возвращает значение TRUE, а в случае неудачи — соответствующее сообщение об ошибке. Если указанный файл уже существует, он будет перезаписан.
Для загрузки сохраненных ранее файлов используется функция load-facts. Функция имеет следующий формат:

Определение 5.15. Синтаксис команды load-facts

(load-facts <имя-файла>)

Здесь <имя-файла> — имя текстового файла, сохраненного ранее с помощью команды save-facts, содержащего список фактов. Файл со списком фактов можно также создать в любом текстовом редакторе, если вы хорошо разобрались с представлением фактов в CLIPS. Для загрузки сохраненного в предыдущем примере файла выполните:

Пример 5.23. Использование функции load-facts

(load-facts fl)

В случае успешного выполнения команда возвращает значение TRUE, а в случае неудачи — FALSE и соответствующее сообщение об ошибке. Обратите внимание, что если в файле содержатся факты, связанные с явно созданными с помощью конструктора deftemplate шаблонами, то в момент загрузки все необходимые шаблоны должны быть уже определены в системе. Если это условие не будет выполнено, то загрузка фактов закончится неудачно. К счастью, CLIPS также позволяет и загрузку конструкторов из текстового файла, но об этом мы поговорим в следующей главе, после рассмотрения конструктора defrule.

Кроме описанных выше функций и команд работы с шаблонами, фактами и предопределенными фактами, в CLIPS существует еще несколько полезных функций и команд, служащих той же цели. Полное описание этих функций и команд приведено в *гл. 15* и *16*. Далее, имея достаточный объем знаний о фактах в CLIPS и способах работы с ними, приступим к изучению правил.

ГЛАВА 6. Правила.

CLIPS поддерживает эвристическую и процедурную парадигму представления знаний. Для представления знаний в процедурной парадигме CLIPS предоставляет такие механизмы, как глобальные переменные, функции и родовые функции, речь о которых пойдет в *гл. 7, 8 и 10* соответственно. В этой главе мы рассмотрим такой способ представления знаний, как *правила.* Правила в CLIPS служат для представления эвристик или так называемых "эмпирических правил", которые определяют набор действий, выполняемых при возникновении некоторой ситуации. Разработчик экспертной системы определяет набор правил, которые вместе работают над решением некоторой задачи. Правила состоят из *предпосылок* и *бедствия.* Предпосылки называются также *ЕСЛИ-частью правила, левой частью правила* или *LHS правила* (left-hand side of rule). Следствие называется *ТО-частью правила, правой частью правила* или *RHS правила* (right-hand side of rule).
Предпосылки правила представляют собой набор условий (или условных элементов), которые должны удовлетвориться, для того чтобы правило выполнилось. Предпосылки правил удовлетворяются в зависимости от наличия или отсутствия некоторых заданных фактов в списке фактов (о котором было рассказано в предыдущей главе) или некоторых созданных объектов, являющихся экземплярами классов, определенных пользователем (о которых будет рассказано в *гл. 11).* Один из наиболее распространенных типов условных выражений в CLIPS — *образцы* (patterns). Образцы состоят из набора ограничений, которые используются для определения того, удовлетворяет ли некоторый факт или объект условному элементу. Другими словами, образец задает некоторую маску для фактов или объектов. Процесс сопоставления образцов фактам или объектам называется *процессом сопоставления образцов* (pattern-matching). CLIPS предоставляет механизм, называемый *механизмом логического вывода* (inference engine), который автоматически сопоставляет образцы с текущим списком фактов и определенными объектами в поисках правил, которые применимы в данный момент.

Следствие правила представляется набором некоторых действий, которые необходимо выполнить, в случае если правило применимо к текущей ситуации. Таким образом, действия, заданные вследствие правила, выполняются по команде механизма логического вывода, если все предпосылки правила удовлетворены. В случае если в данный момент применимо более одного правила, механизм логического вывода использует так называемую *стратегию разрешения конфликтов* (conflict resolution strategy), которая определяет, какое именно правило будет выполнено. После этого CLIPS выполняет действия, описанные вследствие выбранного правила (которые могут оказать влияние на список применимых правил), и приступает к выбору следующего правила. Этот процесс продолжается до тех пор, пока список применимых правил не опустеет.

Чтобы лучше понять сущность правил в CLIPS, их можно представить в виде оператора IF-THEN, используемого в процедурных языках программирования, например, таких как Ada или C. Однако условия выражения IF-THEN в процедурных языках вычисляются только тогда, когда поток управления программы непосредственно попадает на данное выражение путем последовательного перебора выражений и операторов, составляющих программу. В CLIPS, в отличие от этого, механизм логического вывода создает и постоянно модифицирует список правил, условия которых в данный момент удовлетворены. Эти правила запускаются на выполнение механизмом логического вывода. С этой стороны правила похожи на обработчики сообщений, присутствующие в таких языках программирования, как, например, Ada или Smalltalk.

Без правил не обойдется ни одна экспертная система, так что правила и язык их представления в экспертной системе можно смело назвать самой важной частью любой экспертной оболочки. Успех экспертной системы во многом определяется тем, насколько удачен способ представления знаний в виде правил, и насколько хорошо им владеет разработчик экспертной системы. Вся данная глава посвящена правилам, их синтаксису и способам построения, их функционированию и назначению, а также приемам их применения.

6.1. Создание правил. Конструктор *defrule*

Для добавления новых правил в базу знаний CLIPS предоставляет специальный конструктор defrule. В общем виде синтаксис данного конструктора можно представить следующим образом:

Определение 6.1. Синтаксис конструктора defrule

```
(defrule
        <имя-правила>
        [<комментарии>]
        [<определение-свойства-правила>]
        <предпосылки >                    ; левая часть правила
        =>
        <следствие>                       ; правая часть правила
              )
```

Имя правила должно быть значением типа symbol. В качестве имени правила нельзя использовать зарезервированные слова CLIPS, которые были перечислены ранее. Повторное определение существующего правила приводит к удалению правила с тем же именем, даже если новое определение содержит ошибки.

Комментарии являются необязательными, и, как правило, описывают назначения правила. Комментарии необходимо заключать в кавычки. Эти комментарии сохраняются и в дальнейшем могут быть доступны при просматривании определения правила.

Определение правила может содержать объявление свойств правила, которое следует непосредственно после имени правила и комментариев. Более подробно свойства правила будут рассмотрены ниже.

В справочной системе и документации по CLIPS для обозначения предпосылок правила чаще всего используется термин "LHS of rule", а для обозначения следствия "RHS of rule", поэтому в дальнейшем мы будем использовать аналогичную терминологию — левая и правая часть правила.

Левая часть правила задается набором условных элементов, который обычно состоит из условий, примененных к некоторым образцам. Заданный набор образцов используется системой для сопоставления с имеющимися фактами и объектами. Все условия в левой части правила объединяются с помощью неявного логического оператора and. Правая часть правила содержит список действий, выполняемых при активизации правила механизмом логического вывода. Для разделения правой и левой части правил используется символ =>. Правило не имеет ограничений на количество условных элементов или действий. Единственным ограничением является свободная память вашего компьютера. Действия правила выполняются последовательно, но тогда и только тогда, когда все условные элементы в левой части этого правила удовлетворены.

Если в левой части правила не указан ни один условный элемент, CLIPS автоматически подставляет условие-образец initial-fact или initial-object. Таким образом, правило активизируется всякий раз при появлении в
базе знаний факта initial-fact ИЛИ Объекта initial-object. О факте initial-fact было рассказано в предыдущей главе, об объекте initial-object вы узнаете в *гл. 11*. Если в правой части правила не определено ни одно действие, правило может быть активировано и выполнено, но при этом ничего не произойдет.

Более подробно синтаксис левой и правой части правил будет описан далее в этой главе, а пока, в качестве демонстрации применения правил, напишем простейшую CLIPS-программу, которая по традиции здоровается со всем миром, сразу после своего рождения. Вы наверно знакомы с текстом подобных программ для процедурных языков программирования, таких как, например, C. На языке CLIPS такая программа будет выглядеть следующим образом:

Пример 6.1. Программа "Hello-World!"

```
(clear)
(defrule
        Hello-World
        "My FirstCLIPS Rule"
        =>
        (printout  t  crlf crlf)
        (printout  t    ********************  crlf)
        (printout  t   "* HELLO WORLD!!!   *"  crlf)
        (printout  t    ********************  crlf)
```

```
        (printout t crlf crlf)
(reset)
(run)
```

Так как это первая наша программа на языке CLIPS, разберем подробно все ее действия и то, каким образом они выполняются.

Функция clear полностью очищает систему, т. е. удаляет все правила, факты и прочие объекты базы знаний CLIPS, добавленные конструкторами, приводит систему в начальное состояние, необходимое для каждой новой программы.

Затем, с помощью конструктора defrule в систему добавляется новое правило с именем Hello-World и соответствующими комментариями. Левая часть правила в данном случае отсутствует, поэтому CLIPS автоматически формирует предпосылки, состоящие из единственного условного выражения (initial-fact). Это выражение является образцом простейшего типа. При запуске программы на выполнение механизм логического вывода CLIPS будет искать в списке фактов факт (initial-fact) и если он там будет найден — активизирует правило. Правая часть нашего правила состоит из нескольких вызовов функции printout. Подробно данная функция будет рассмотрена в *гл. 15.* Сейчас же вам необходимо знать только то, что эта функция выводит текстовое выражение в один из потоков вывода. Параметр t задает стандартный поток вывода — экран. Он аналогичен, например, стандартному потоку cout в C++. Выражение clrf служит для перехода на новую строку.

Функция reset, как уже упоминалось ранее, очищает список фактов и заносит в него факт (initial-fact), что очень важно для нормального функционирования нашей программы. И, наконец, функция run запускает механизм логического вывода и приводит нашу программу в движение. Если описанные выше действия были выполнены правильно, то вы должны увидеть результат, аналогичный приведенному на рис. 6.1 — фразу "HELLO WORLD!!!" в красивой рамочке из звездочек:

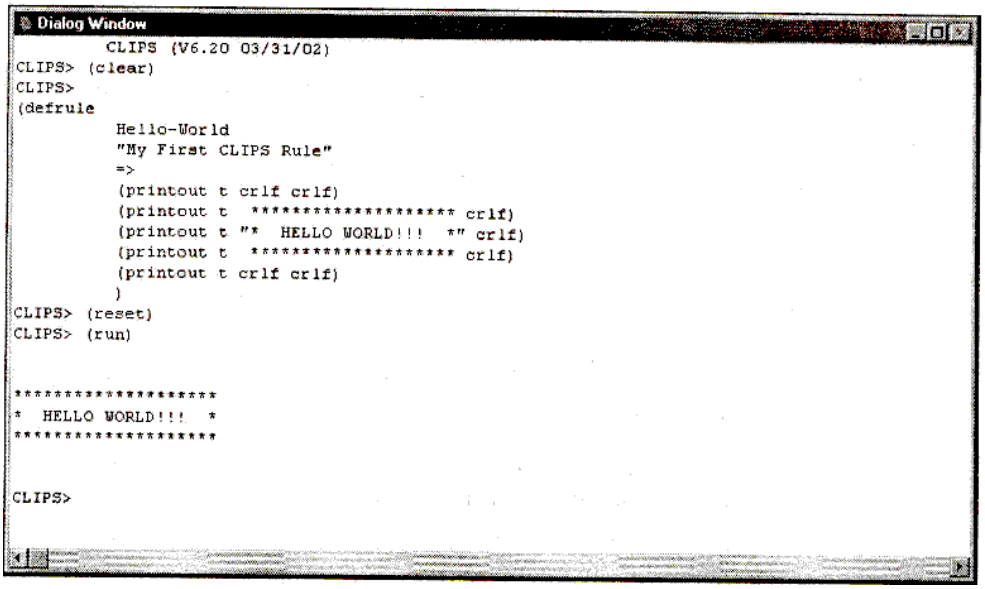

Рис. 6.1. Результат работы программы "Hello-World!"

Чтобы запустить нашу программу на выполнение еще раз, достаточно вызвать функции reset и run. Эти функции можно вводить с клавиатуры, кроме того, они доступны в меню **Execution** и имеют "горячие" клавиши <Ctrl>+<E> и <Ctrl>+<R> соответственно.

CLIPS поддерживает ряд функций, команд и визуальных средств, необходимых для эффективной работы с правилами. Самые основные из них будут рассмотрены в данной главе по мере необходимости. Полный список функций и команд, предназначенный для работы с правилами, будет приведен в *гл. 15* и *16* соответственно. Сейчас же рассмотрим визуальный инструмент, доступный пользователям Windows-версии среды CLIPS — **Defrule Manager** (Менеджер правил). Для запуска менеджера правил в меню **Browse** выберите пункт **Defrule Manager.** Внешний вид этого инструмента показан на рис. 6.2.

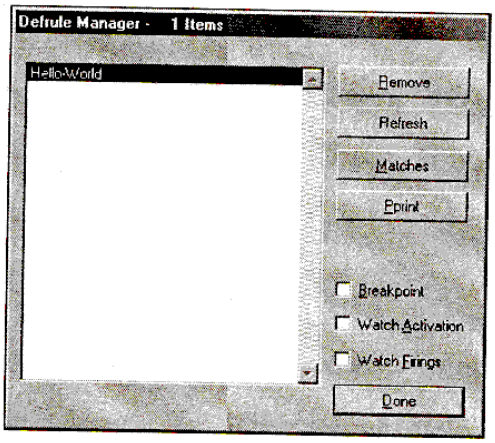

Рис. 6.2. Окно менеджера правил

Менеджер отображает список правил, присутствующих в системе в данный момент, и позволяет выполнять над ними ряд операций. Например, с помощью кнопки **Remove** можно удалить выбранное правило из системы, а с помощью **Ppprint** вывести в окне CLIPS определение выделенного правила вместе с введенными комментариями. Общее количество правил отображается в заголовке окна менеджера — **Defrule Manager — 1 Items.** Другие возможности менеджера правил будут рассмотрены позже, по мере необходимости.

Как вы уже могли убедиться, разбирая приведенный выше пример несложной программы, довольно тяжело создать даже минимально полезную программу на языке CLIPS, не представляя себе, что именно происходит при выполнении вашей программы. При создании экспертных систем необходимо точно знать, как происходит сопоставление образцов, заданных в левой части правила, каким именно образом выбирается правило для выполнения и т. д. Именно поэтому в последующих разделах мы остановимся на внутренних алгоритмах представления и обработки правил в CLIPS, а уже затем перейдем к описанию функций и команд, связанных с правилами.

6.2. Основной цикл выполнения правил

После того как в систему добавлены все необходимые правила и приготовлены начальные списки фактов и объектов, CLIPS готов выполнять правила. В традиционных языках программирования точка входа, точка остановки и последовательность вычислений явно определяются программистом. В CLIPS поток исполнения программы совершенно не требует явного определения. Знания (правила) и данные (факты и объекты) разделены, и механизм логического вывода, предоставляемый CLIPS, применяет данные к знаниям, формируя *список применимых правил,* после чего последовательно выполняет их. Этот процесс называется *основным циклом выполнения правил* (basic cycle of rule execution). Рассмотрим последовательность действий (шагов), выполняемых системой CLIPS в этом цикле в момент выполнения нашей программы:

1.Если был достигнут предел выполнения правил или не был установлен текущий фокус, выполнение прерывается. В противном случае, для выполнения выбирается первое правила модуля, на котором был установлен фокус. Если в текущем плане выполнения нет удовлетворенных правил, то фокус перемещается по стеку фокусов и устанавливается на следующий модуль в списке. Если стек фокусов пуст, выполнение прекращается. Иначе шаг 1 выполняется еще один раз. Более подробно о модулях и фокусе будет рассказано в *гл. 12.*

2.Выполняются действия, описанные в правой части выбранного правила. Использование функции return может менять положение фокуса в стеке фокусов. Число запусков данного правила увеличивается на единицу, для определения предела выполнения правила.

3.В результате выполнения шага 2 некоторые правила могут быть активированы или дезактивированы. Активированные правила (т. е. правила, условия которых удовлетворяются в данный момент) помещаются в план решения задачи модуля, в котором они определены. Размещение в плане определяется приоритетом правила (salience) и текущей стратегией раз решения конфликтов (эти понятия будут описаны ниже). Дезактивированные правила удаляются из текущего плана решения задачи. Если для правила установлен режим просмотра активаций, то пользователь получит соответствующее информационное сообщение при каждой активации или дезактивации правила (режим просмотра активаций можно установить с помощью диалогового окна **Watch Options.** Для этого выберите пункт **Watch** в меню **Execution** и установите флажок **Activations).**

4.Если установлен режим динамического приоритета (dynamic salience), то для всех правил из текущего плана решения задачи вычисляются новые значения приоритета. После этого цикл повторяется с шага 1.

6.3. Свойства правил

Для более полного понимания материала, изложенного далее в этой главе, необходимо разобраться с таким понятием, как свойства правил. Свойства правил позволяют задавать характеристики правил до описания левой части правила, как было в *разд. 6.1.* Для задания свойства правила используется ключевое слово declare. Одно правило может иметь только одно определение свойства, заданное с помощью declare.

Определение 6.2. Синтаксис свойств правил

<определение-свойства-правила> ::= (declare <свойство-правила>)
<свойство-правила> ::= (salience <целочисленное выражение>) | (auto-focus TRUE | FALSE)

6.3.1. Свойство *salience*

Свойство правила salience позволяет пользователю назначать приоритет для своих правил. Объявляемый приоритет должен быть выражением, имеющим целочисленное значение из диапазона от —10 000 до +10 000. Выражение, представляющее приоритет правила, может использовать глобальные переменные и функции (которые будут описаны в *гл. 7* и *8* соответственно). Однако старайтесь не указывать в этом выражении функций, имеющих побочное действие. В случае если приоритет правила явно не задан, ему присваивается значение по умолчанию — 0.

Значение приоритета может быть вычислено в одном из трех случаев: при добавлении нового правила, при активации правила и на каждом шаге основного цикла выполнения правил. Два последних варианта называются *динамическим приоритетом* (dynamic salience). По умолчанию значение приоритета вычисляется только во время добавления правила. Для изменения этой установки можно использовать команду set-salience-evaluation.
Кроме того, пользователи Windows-версии среды CLIPS могут изменить эту настройку с помощью диалогового окна **Execution Options.** Для этого выберите пункт **Options** в меню **Execution,** в появившемся диалоговом окне укажите необходимый режим вычисления приоритета с помощью раскрывающегося списка **Salience Evaluation,** как показано на рис. 6.3.

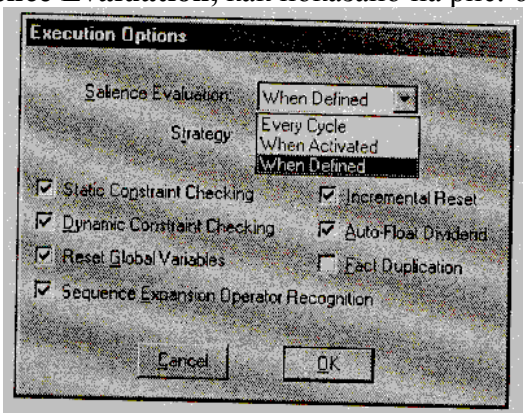

Рис. 6.3. Установка способа вычисления приоритетов правил

Замечание

Каждый метод вычисления приоритета содержит в себе предыдущий (т. е. если приоритет вычисляется на каждом шаге основного цикла выполнения правил, то он также вычисляется и при активации правила, а так же при его добавлении в систему).

6.3.2. Свойство *auto-focus*

Свойство auto-focus позволяет автоматически выполняться команде focus (о которой будет рассказано в *гл. 12)* при каждой активации правила. Если свойство auto-focus установлено в значение TRUE, то команда focus в модуле, в котором определено данное правило, автоматически выполняется всякий раз при активации правила. Если свойству auto-focus присвоено значение FALSE, то при активации правила не происходит никаких действий. По умолчанию это свойство установлено в FALSE.

6.4. Стратегия разрешения конфликтов

План решения задачи — это список всех правил, имеющих удовлетворенные условия при некотором, текущим состоянием списка фактов и объектов (и которые еще не были выполнены). Каждый модуль имеет свой собственный план решения задачи. Выполнение плана подобно стеку (верхнее правило плана всегда будет выполнено первым). Когда активируется новое правило, оно размещается в плане решения задачи руководствуясь следующими факторами:

1.Только что активированное правило помещается выше всех правил с меньшим приоритетом и ниже всех правил с большим приоритетом.

2.Среди правил с одинаковым приоритетом используется текущая стратегия разрешения конфликтов для определения размещения среди других правил с одинаковым приоритетом.

3.Если правило активировано вместе с несколькими другими правилами, добавлением или исключением некоторого факта и с помощью шагов 1 и 2 нельзя определить порядок правила в плане решения задачи, то правило произвольным образом упорядочиваются вместе с другими правилами, которые были активированы. Заметьте, что в этом случае порядок, в котором правила были добавлены в систему, оказывает произвольный эффект на разрешения конфликта (который в высшей степени зависит от текущей реализации правил). Старайтесь не использовать произвольное упорядочивание правил при решении задач, в которых требуются точные результаты или объяснения полученных решений.

CLIPS поддерживает семь различных стратегий разрешения конфликтов: *стратегия глубины* (depth strategy), *стратегия ширины* (breadth strategy), *стратегия упрощения* (simplicity strategy), *стратегия усложнения* (complexity strategy), *LEX* (LEX strategy), *MEA* (MEA strategy) и *случайная стратегия* (random strategy). По умолчанию в CLIPS установлена стратегия глубины. Текущая стратегия может быть установлена командой set-strategy (которая переупорядочит текущий план решения задачи, базируясь на новой стратегии). Кроме того, пользователи Windows-версии среды CLIPS могут указать необходимую стратегию поиска с помощью диалогового окна **Execution Options** (см. рис. 6.3). Для этого выберите пункт **Options** в меню **Execution,** в появившемся диалоговом окне выберите необходимую стратегию с помощью раскрывающегося списка **Strategy.**

6.4.1. Стратегия глубины

Только что активированное правило помещается выше всех правил с таким же приоритетом. Например, допустим, что факт-А активировал правило-1 и правило-2 и факт-Б активировал правило-3 и правило-4, тогда, если факт-А добавлен перед фактом-Б, в плане решения задачи правило-3 и правило-4 будут располагаться выше, чем правило-1 и правило-2. Однако позиция правила-1 относительно правила-2 и правила-3 относительно правила-4 будет произвольной.

6.4.2. Стратегия ширины

Только что активированное правило помещается ниже всех правил с таким же приоритетом. Например, допустим, что факт-А активировал правило-1 и правило-2 и факт-Б активировал правило-3 и правило-4, тогда, если факт-А добавлен перед фактом-В, в плане решения задачи правило-1 и правило-2 будут располагаться выше, чем правило-3 и правило-4. Однако позиция правила-1 относительно правила-2 и правила-3 относительно правила-4 будет произвольной.

6.4.3. Стратегия упрощения

Между всеми правилами с одинаковым приоритетом только что активированные правила размещаются выше всех активированных правил с равной или большей *определенностью* (specificity). Определенность правила вычисляется по числу сопоставлений, которые нужно сделать в левой части правила. Каждое сопоставление с константой или заранее связанной с фактом переменной добавляет к определенности единицу. Каждый вызов функции в левой части правила, являющийся частью условных элементов :, = или test, также добавляет к определенности единицу. Логические функции and, or и not не увеличивают определенность правила, но их аргументы могут сделать это. Вызовы функций, сделанные внутри функций, не увеличивают определенность правила.

Например, следующее правило имеет определенность, равную 5.

Пример 6.2. Вычисление определенности правила

```
(defrule    example
          (item ?x  ?y  ?x)
          (test  (and  (numberp  ?x)  (>  ?x  (+  10  ?y))  (<  ?x  100)))
          =>)
```

И сравнение заранее связанной переменной ?x с константой, и вызовы функций numberp, < и > добавляют единицу к определенности правила. В итоге получаем определенность, равную 5. Вызовы функций and и + не увеличивают определенность правила.

6.4.4. Стратегия усложнения

Между правилами с одинаковым приоритетом, только что активированные правила размещаются выше всех активированных правил с равной или меньшей определенностью.

6.4.5. Стратегия LEX

Между правилами с одинаковым приоритетом только что активированные правила размещаются с использованием одноименной стратегии, впервые использованной в системе OPS5. Для определения места активированного правила в плане решения задачи используется "новизна" образца, который активировал правило. CLIPS маркирует каждый факт или объект временным тегом для отображения относительной новизны каждого факта или объекта в системе. Образцы, ассоциированные с каждой активацией правила, сортируются по убыванию тегов для определения местоположения правила. Активация правила, выполненная более новыми образцами, располагается перед активацией, осуществленной более поздними образцами. Для определения порядка размещения двух активаций правил, поодиночке сравниваются отсортированные временные теги для этих двух активаций, начиная с наибольшего временного тега. Сравнение продолжается до тех пор, пока не останется одна активация с наибольшим временным тегом. Эта активация размещается выше всех остальных в плане решения задачи.

Если активация некоторого правила выполнена большим числом образцов, чем активация другого правила и все сравниваемые временные теги одинаковы, то активация с большим числом временных тегов помещает перед активацией с меньшим. Если две активации имеют одинаковое количество временных тегов и их значения равны, то правило с большей определенностью *(см. разд. 6.4.3)* помещается перед активацией с меньшей. В отличие от системы OPS5, условный элемент not в CLIPS имеет псевдовременной тег, который также используется в данной стратегии разрешения конфликтов. Временной тег условного элемента not всегда меньше, чем временной тег образца.

В качестве примера рассмотрим следующие шесть активаций правил, приведенные в LEX-порядке (запятая в конце строки активации означает наличие логического элемента not). Учтите, что временные теги фактов не обязательно равны индексу, но если индекс факта больше, то больше и его временной тег. Для данного примера примем, что временные теги равны индексам.

Пример 6.3. Правила, отсортированные стратегией LEX

```
rule-б:     f-l,f-4
rule-5:     f-l,f-2,f-3,
rule-1:     f-l,f-2,f-3
rule-2:     f-3, f-1
rule-4:     f-l,f-2
rule-3:     f-2, f-1
```

В примере 6.4 показаны те же активации с индексами фактов в том порядке, в котором они сравниваются стратегией LEX.

Пример 6.4. Порядок сравнения стратегией LEX

```
rule-6:     f-4,f-l
rule-5:     f-3,f-2,f-1,
rule-1:     f-3,f-2,f-1
```

rule-2: f-3,f-1
rule-4: f-2,f-1,
rule-3: f-2,f-1

6.4.6. Стратегия MEA

Между правилами с одинаковым приоритетом только что активированные правила размещаются *с* использованием одноименной стратегии, впервые использованной в системе OPS5. Основное отличие стратегии MEA от LEX в том, что в стратегии MEA не производится сортировка образцов, активировавших правило. Сравниваются только временные теги первых образцов двух активаций. Активация с большим тегом помещается в план решения задачи перед активацией с меньшим. Если обе активации имеют одинаковые временные теги, ассоциированные с первым образцом, то для определения размещения активации в плане решения задачи используется стратегия LEX. Так же, как и в стратегии LEX, условный элемент not имеет псевдовременной тег.

В качестве примера рассмотрим следующие шесть активаций, приведенные в MEA-порядке (запятая на конце активации означает наличие логического элемента not).

Пример 6.5. Правила, отсортированные стратегией MEA

rule-2: f-3,f-l
rule-3: f-2,f-l
rule-6: f-l,f-4
rule-5: f-1,f-2,f-3,
rule-1: f-1,f-2,f-3
rule-4: f-1, f-2,

6.4.7. Случайная стратегия

Каждой активации назначается случайное число, которое используется для определения местоположения среди активаций с одинаковым приоритетом. Это случайное число сохраняется при смене стратегий, таким образом, тот же порядок воспроизводится при следующей установки случайной стратегии (среди активаций в плане решения задачи, когда стратегия заменена на исходную).

6.5. Синтаксис LHS правила

Этот раздел описывает синтаксис, используемый в левой части правил. Левая часть правил содержит список *условных элементов* (conditional elements или CEs), которые должны удовлетворяться, для того чтобы правило было помещено в план решения задачи. Существует восемь типов условных элементов, используемых в левой части правил: *CEs-образцы, test CEs, and CEs, or CEs, not CEs, exists CEs, forall CEs* и *logical CEs*. Образцы — наиболее часто используемый условный элемент. Он содержит ограничения, которые служат для определения, удовлетворяет ли какой-нибудь элемент данных (факт или объект) образцу. Условие test используется для оценки выражения, как части процесса сопоставления образов. Условие and применяется для определения группы условий, каждое из которой должно быть удовлетворено. Условие or — для определения одного условия из некоторой группы, которое должно быть удовлетворено. Условие not — для определения условия, которое не должно быть удовлетворено. Условие exists — для проверки наличия, по крайней мере одного, совпадения факта (или объекта) с некоторым заданным образцом. И наконец, условие logical позволяет выполнить добавление фактов и создание объектов в правой части правила, связанных с фактами и объектами, совпавшими с заданным образцом в левой часта правила (поддержка достоверности фактов в базе знаний).

Синтаксис условного элемента можно формализовать следующим образом:

Определение 6.3. Синтаксис условного элемента

<условный-элемент> ::= <pattern-CE> |
<assigned-pattern-CE> |
<not-CE> |

<and-CE> |
<or-CE> |
<logical-CE> |
<test-CE> |
<exists-CE> |
<forall-CE>

В последующих разделах будет подробно рассмотрен синтаксис каждого условного элемента.

6.5.1. Образец (pattern CE)

Этот условный элемент состоит из списка *ограничений полей, групповых символов* (wildcards) и *переменных,* которые используются для поиска множества фактов или объектов, которые соответствуют заданному образцу. Таким образом, образец как бы определяет маску, которой должны соответствовать данные. Такой условный элемент удовлетворяется любым фактом или объектом, соответствующим заданным ограничениям.

Ограничения полей — это набор ограничений, которые используются для проверки простых полей или слотов объектов. Ограничения полей могут состоять только из одного символьного ограничения, однако, несколько ограничений можно соединять вместе. В дополнение к символьным ограничениям, CLIPS поддерживает три других типа ограничений: *объединяющие ограничения, предикатные ограничения* и *ограничения, возвращающие значения (см. гл. 13).*

Групповые символы используются при сопоставлении образцов в ситуации, когда простое поле или группа полей могут принимать любые значения.

Переменные применяются для хранения значения поля, которое может быть впоследствии использовано в левой части правила для другого условного элемента или в правой части, как аргумент действия.

Первое поле любого образца обязательно должно быть значением типа symbol и не может принимать значения других типов. CLIPS использует первое поле для определения: является ли данный образец упорядоченным фактом, шаблоном или объектом. Ключевое слово object зарезервировано для создания образцов, предназначенных для сопоставления с объектами. Любое другое значение типа symbol должно соответствовать имени шаблона, созданного с помощью конструктора deftemplate или неявно созданного шаблона. Для задания имен слотов также должны использоваться значения типа symbol.

В слотах простых полей образцов, предназначенных для объектов и шаблонов, может содержаться только одно ограничение поля, и не могут присутствовать групповые символы или переменные. В составных слотах может содержаться любое количество ограничений поля.

Далее будут показаны синтаксис и примеры использования образцов. *В подразделе "Сопоставление образцов с объектами" разд. 6.5.1* будут объяснены отличия между образцами для шаблонов и образцами для объектов. Для обеспечения наглядности примеров в последующих разделах будут использоваться факты и шаблоны, приведенные в примере 6.6.

Пример 6.6. Необходимые для дальнейшей работы шаблоны и факты

```
(deffacts      data-facts
               (data 1.0 blue "red")
               (data 1 blue)
               (data 1 blue red)
               (data 1 blue RED)
               (data 1 blue red 6.9))
(deftemplate   person
               (slot name)
               (slot age)
               (multislot friends))
(deffacts      people
               (person (name Joe)   (age 20) )
               (person (name Bob)   (age 20) )
               (person (name Joe)   (age 34))
```

(person (name Sue) (age 34))
(person (name Sue) (age 20))

Символьные ограничения

Основные ограничения, использующиеся в образцах, — это ограничения, определяющие точное соответствие между полями факта и образцом. Эти ограничения называются *символьными*. Символьное ограничение полностью состоит из констант, таких как вещественные и целые числа, значения типа symbol, строки или имена объектов. Они не могут содержать групповых символов или переменных. Все символьные ограничения при сопоставлении образцов должны точно совпадать по всем указанным полям, иначе факт не будет считаться подошедшим данному образцу.
Условный элемент, представляющий собой образец для неупорядоченного факта, в котором присутствуют только символьные ограничения, имеет следующий синтаксис:

Определение 6.4. Синтаксис символьных ограничений для неупорядоченного факта

(<ограничение-1> … <ограничение-n>)

Условный элемент, представляющий собой образец для шаблона, в котором присутствуют только символьные ограничения, выглядит так:

Определение 6.5. Синтаксис символьных ограничений для шаблона

(<имя—шаблона > (<имя-слота-1> <ограничение-1>)
…
(<имя-слота-n> <ограничение-n>))

Рассмотрим пример правил, использующих в качестве образца — образец фактов (как упорядоченных, так и шаблонов) с символьными ограничениями. Для нормальной работы этого примера (как впрочем и всех примеров, приведенных в *подразделе "Адрес образца" разд. 6.5.1)* необходимо ввести в CLIPS все конструкторы предопределенных фактов и шаблонов, представленные в *разд. 6.5.1.* После этого следует выполнить команду reset для инициализации списка фактов. Для проверки правильности выполненных операций откройте окно **Facts,** как было рассмотрено в *разд. 5.4.2.* Если все описанные действия были выполнены без ошибок, вы должны увидеть результат, приведенный на рис. 6.4.

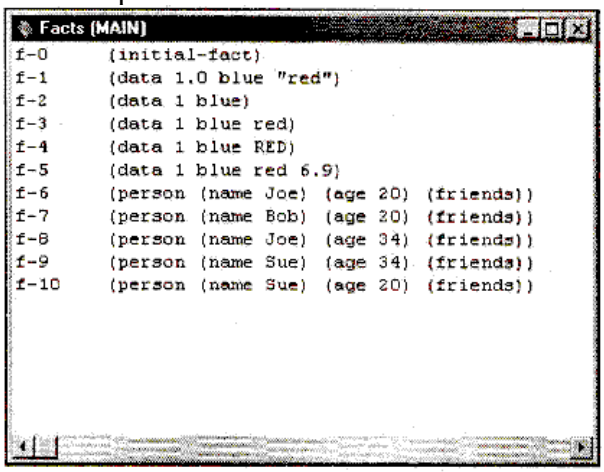

Рис. 6.4. Список необходимых фактов

Для нормальной работы примеров не забывайте выполнять команду reset перед каждым запуском правил. Введите в CLIPS определения следующих правил:

Пример 6.7. Правила с символьными ограничениями

```
(defrule  Find-data
          (data  1  blue  red)
          =>
          (printout t crlf "Found data (data 1 blue red)" crlf))
(defrule Find-Bob-20
          (person (name Bob) (age 20))
          =>
          (printout t crlf "Found Bob-20 (person (name Bob) (age 20))" crlf))
(defrule Find-Bob-30
          (person (name Bob) (age 30))
          =>
          (printout t crlf "Found Bob-30 (person (name Bob) (age 30))" crlf))
```

Выполните команды reset и run. Вы должны получить результат, приведенный на рис. 6.5.

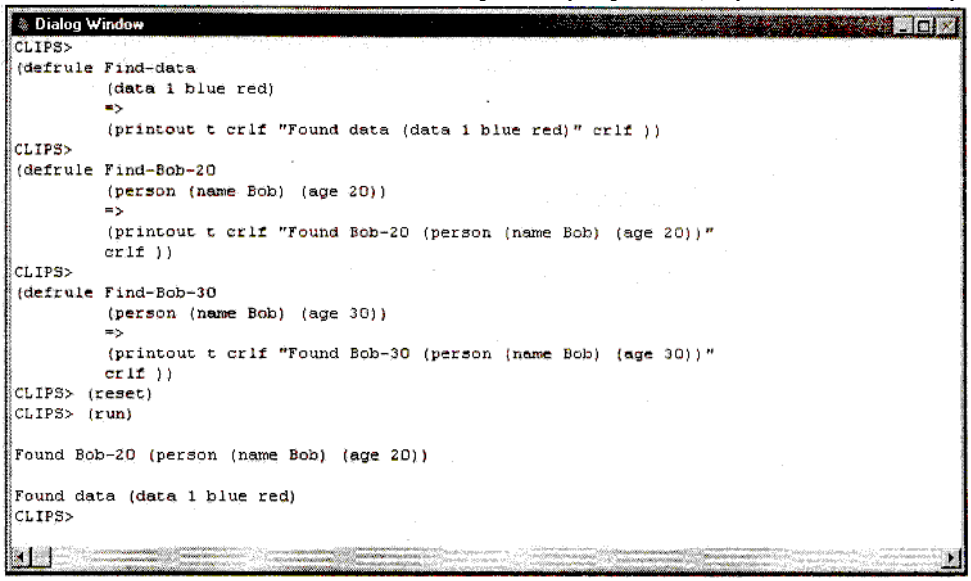

Рис. 6.5. Выполнение правил с символьными ограничениями

Как мы видим, были активированы и выполнены два правила: Find-data и Find-Bob-20. Это произошло потому, что образцы, заданные в левой части этих правил, нашли в списке фактов данные, полностью соответствующие заданным символьным ограничениям.

Групповые символы для простых и составных полей

В CLIPS имеется два различных групповых символа, которые используются для сопоставления полей в образцах. CLIPS интерпретирует эти групповые символы как место для подстановки некоторых частей данных, удовлетворяющих образцам. Групповой символ для простого поля записывается с помощью знака ?, который соответствует одному любому значению, сохраненному в заданном поле. Групповой символ составного поля записывается с помощью знака $? и соответствует, возможно, пустой последовательности полей, сохраненной в составном поле. Групповые символы для простых и составных полей могут комбинироваться в любой последовательности. Нельзя использовать групповой символ составного поля для простых полей. По умолчанию не заданный в образце простой слот шаблона или объекта сопоставляется с неявно заданным групповым символом для простого поля. Аналогично не заданный в образце составной слот сопоставляется с неявно заданным групповым символом для составного поля.

Условный элемент, представляющий собой образец для неупорядоченного факта, в котором присутствуют только символьные ограничения и групповые символы, будет иметь следующий вид:

Определение 6.6. Синтаксис ограничений для неупорядоченного факта

(<ограничение-1> ... <ограничение-n>)
<ограничение> ::= <символьное-ограничение > | ? | $?

Соответственно для шаблона образец примет вид:

Определение 6.7. Синтаксис ограничений для шаблона

(<имя-шаблона > (<имя-слота-1> <ограничение-1>)
…
(<имя-слота-n> <ограничение-n>))

В качестве примера можно привести следующее правило:

Пример 6.8. Правило Find-data

```
(defrule      Find-data
              (data   ? blue  red $?)   =>)
```

В нашем списке фактов присутствуют два факта, подходящие заданному шаблону и способные активировать данное правило:

Пример 6.9. Факты, активирующие правило Find-data

```
(data  1  blue  red)
(data  1  blue  red  6.9))
```

Рассмотрим еще одно правило:

Пример 6.10. Правило match-all-persons

```
(defrule      match-all-persons
              (person)
              =>)
```

Поскольку person является шаблоном, а в образце данного правила не определен ни один слот шаблона, CLIPS автоматически поставит в соответствие каждому простому слоту групповой символ для простого поля, а составному слоту — символ для составного. Таким образом, правило преобразуется в следующее:

Пример 6.11. Преобразованное правило match-all-persons

```
(defrule      match-all-persons
              (person
                    (name  ?)
                    (age   ?)
                    (friends  $?))
              =>)
```

Это правило будут активировать все факты шаблона person.
Групповые символы для составного поля можно комбинировать с символьными ограничениями, что приводит к получению более мощных возможностей сопоставления образцов. Образец,

который сопоставляется со всеми фактами, имеющими значение YELLOW в любом поле (включая первый), может быть записан так:

Пример 6.12. Образец со значением YELLOW в любом поле

```
(data $? YELLOW $?)
```

Вот несколько фактов, соответствующих этому образцу:

Пример 6.13. Факты со значением yellow в любом поле

```
(data   YELLOW  blue  red  green)
(data   YELLOW  red)
(data   red YELLOW)
(data   YELLOW)
(data   YELLOW  data  YELLOW)
```

Последний факт будет соответствовать образцу дважды, т. к. yellow присутствует в нем дважды. Использование группового символа для составного поля позволяет создавать гораздо более общие образцы, чем те, которые можно сформировать с помощью групповых символов для простого поля. Однако подобная общность приводит к тому, что процесс сопоставления образцов, использующих групповые символы, иногда занимает гораздо больше времени, чем аналогичный процесс с образцами, использующими только групповые символы для простых полей.

Переменные, связанные с простыми и составными полями

Групповые символы заменяют любые поля образца и могут принимать какие угодно значения этих полей. Значение поля может быть связано с переменными для последующего сопоставления, отображения и других действий. Это выполняется с помощью применения имени переменной следующим непосредственно после группового символа.
Таким образом, синтаксис ограничения, применяемого в образце, примет следующий вид:

Определение 6.8. Синтаксис ограничений

```
<ограничение> ::=                 <символьное-ограничение > |
                                  ? |
                                  $? |
                                  <переменная-простого-поля>|
                                  <переменная-составного-поля>
<переменная-простого-поля>     ::=  ?<имя-переменной>
<переменная-составного-поля>   ::=  $?<имя-переменной>
```

Имя переменной должно быть значением типа symbol и обязательно начинаться с буквы. В имени переменной не разрешается использовать кавычки, т. е. строка не может использоваться как имя переменной или ее часть.

Правила сопоставления образцов при использовании переменных в ограничениях образца аналогичны правилам, использующимся для групповых символов. В момент первого появления имени переменной она ведет себя так же, как и соответствующий групповой символ. В этот момент CLIPS связывает значения поля с заданной переменной. Эта связь будет действовать только в рамках правила, в котором она возникла. Каждое правило имеет свой собственный список имен переменных со значениями, связанными с ними, эти переменные локальны для правил.
Связанные переменные могут быть использованы во внешних функциях. Символ $ имеет особое значение в левой части правил — этот оператор отображает, что некоторая, возможно пустая, последовательность полей требует сопоставления. В правой части правила символ $ ставится перед переменной для обозначения того, что перед использованием переменной в качестве аргумента функции необходимо раскрыть последовательность полей, содержащихся в переменной. Таким образом, при использовании переменных в качестве параметров функций (как в левой, так и правой части правил) перед именем переменной, содержащей значение составного поля, не должен стоять символ $ (за исключением случаев, когда требуется раскрыть последовательность полей). При использовании переменной, содержащей значение составного

поля, в других случаях, перед ее именем должен стоять символ $. Нельзя применять переменную составного поля при операциях с простым полем образца шаблона или объекта.
В качестве примера введите в среду CLIPS следующее правило:

Пример 6.14. Правило Find-data

```
(defrule      Find-data
              (data  ?  blue  ?x  $?y)   =>
              (printout  t "Found  data  (data  ?  blue " ?x " " ?y ")" crlf))
```

Выполните команды reset и run. Если правило было введено в систему без ошибок, то на экране появится следующий результат:

Пример 6.15. Результат работы правила Find-data

```
Found  data    (data ?  blue  red       (6.9))
Found  data    (data ?  blue  RED       ())
Found  data    (data ?  blue  red       ()}
Found  data    (data ?  blue  red       ()}
```

Образцу, заданному в правиле, удовлетворяют четыре факта с индексами 1, 3, 4, 5. В результате активации правило выводит на экран свойства фактов, активировавших правило. Значение переменной, содержащей значение из составного поля, выводится в скобках. Кроме первого случая (факта с индексом 5), переменная содержит пустое значение. Переменную составного поля не обязательно использовать в качестве последнего ограничения. Рассмотрим следующее правило:

Пример 6.16. Модифицированное правило Find-data

```
(defrule   Find-data
             (data ?x $?y ?z) =>
             (printout  t   "x="   ?x   "  y="   ?y   "   z="   ?z  crlf))
```

Заданному образцу удовлетворяют все факты data, но обратите внимание, каким образом связываются значения с переменной y в разных случаях:

Пример 6.17. Результат работы модифицированного правила Find-data

```
x=1.0          y=(blue)          z=red
x=1            y=()              z=blue
x=1            y=(blue)          z=red
x=1            y=(blue)          z=RED
x=1            y=(blue red)      z=6.9
```

После того как произошло связывание переменной со значением, все ссылки на эту переменную возвращают значение, с которым переменная была связана. Это действительно как для переменных, связанных с составными полями, так и для переменных, связанных с простыми полями. Кроме того, допустимы ссылки между образцами в одном правиле.

Пример 6.18. Правило Find-2-Coeval-Person

```
(defrule       Find-2-Coeval-Person
               (person (name ?x) (age ?z))
               (person (name ?y) (age &z))
               =>
               (printout  t "name=" ?x " name=" ?y " age=" ?z crlf))
```

Приведенное выше правило Find-2-Coevai-person выведет на экран всевозможные пары имен людей (все перестановки) одинакового возраста. Как научить это правило не выводить

эквивалентные по смыслу или бессмысленные пары одинаковых имен (Bob-Bob), мы увидим в следующих разделах.

Связывающие ограничения

CLIPS предоставляет 3 связывающих ограничения, предназначенных для объединения отдельных ограничений и переменных в единое целое: & (логическое И), | (логическое ИЛИ) и ~ (логическое НЕ). Ограничение & удовлетворяется, если два соседних ограничения удовлетворяются. Ограничение | удовлетворяется, если любое из двух соседних ограничений удовлетворяется. Ограничение ~ удовлетворяется, если следующее за ним ограничение не удовлетворяется. Связывающие ограничения могут комбинироваться почти произвольным образом и в любом количестве. Ограничение ~ имеет наивысший приоритет, далее следуют & и |. В случае одинакового приоритета ограничение вычисляется слева направо. Существует одно исключение из правил приоритета, которое применяется при связывании переменных. Если первое ограничение — это переменная и за ней следует &, то переменная является отдельным ограничением. Ограничение ?x&red|blue вычисляется как ?x& (red|blue), в то время как по правилам приоритета оно должно было вычисляться как (?x&red) (blue.

Связанные ограничения имеют следующий синтаксис:

Определение 6.9. Синтаксис связывающих ограничений

<элемент-1>& <элемент-2> ... & Элемент -n>

<элемент-1>| <элемент-2> ... | Элемент -n>

~ <элемент>

Здесь <элемент> должен быть переменной, связанной с простым или составным полем, ограничением или связанным ограничением.

Таким образом, определения ограничений, приведенные в предыдущих разделах, можно расширить так:

Определение 6.10. Синтаксис ограничений

<ограничение> ::= ? |
$? |
<связанное-ограничение>

<связанное-ограничение> ::= <простое-ограничение> |
<простое-ограничение>&<связанное-ограничение>|
<простое-ограничение>|<связанное-ограничение>|

<простое-ограничение> ::= <элемент>| ~ <элемент>

<элемент> ::= <константа> |
<простая-переменная>|
<составная-переменная>

Ограничение & обычно служит только для объединения с другими ограничениями или связывания переменных. Заметьте, что связывающие ограничения могут использовать связанные переменные и в то же время сами производить связывание переменной со значением некоторого поля. Если имя переменной встретилось в первый раз, то для ограничения будут использоваться остальные члены условного элемента, а переменная будет связана с соответствующим значением поля. Если переменная уже была связана, то ее значение работает как дополнительное ограничение для данного поля.

В качестве примера приведем улучшенный вариант правила Find-2-coevai-Person из предыдущего раздела.

Пример 6.19. Улучшенное правило Find-2-Coeval-Person

```
(defrule      Find-2-Coeval-Person
              (person  (name ?x)  (age ?z))
              (person  (name ?y&~?x)  (age &z))
              =>
              (printout t  "name="  ?x  "  name="  ?y  "  age="  ?z  crlf))
```

Ограничение ?y&~?x запрещает выводить бессмысленные пары одинаковых имен (Bob-Bob). Однако данное правило все еще выводит эквивалентные по смыслу пары имен (например, Bob-Sue и Sue-Bob). Дальнейшее совершенствование правила продолжится в *разд. 6.5.5.*

Предикатные ограничения

Иногда необходимо ограничить поле, основываясь на истинности некоторого логического выражения. CLIPS позволяет использовать предикатные ограничения. Предикатные ограничения позволяют вызывать предикатные функции (функции, которые возвращают значение FALSE при не соответствии условиям и не-FALSE, если значение удовлетворяет условиям) в течение процесса сопоставления образцов. Если предикатная функция возвращает значение НЕ-FALSE, ограничение удовлетворяется. Если предикатная функция возвращает значение FALSE, то ограничение не удовлетворяется. Предикатные ограничения записываются с помощью двоеточия и следующего за ним вызова соответствующей предикатной функции. Обычно предикатные ограничения используются совместно со связывающими ограничениями и при связывании переменных (т. е. если вы имеете переменную, которую нужно связать с некоторым полем и хотите одновременно ее протестировать, объедините ее с предикатным ограничением).
Предикатные ограничения имеют следующий синтаксис:

Определение 6.11. Синтаксис предикатного ограничения

:<вызов-функции>

Таким образом, определение понятия "элемент", приведенное в предыдущем разделе, можно расширить следующим образом:

Определение 6.12. Синтаксис понятия "элемент"

```
<элемент>      ::= <константа> |
                   <простая-переменная>|
                   <составная-переменная>|
                   :<вызов-функции>
```

CLIPS предоставляет несколько готовых предикатных функций (см. гл. 15). Кроме этого, пользователь также может создавать свои собственные предикатные функции.

Пример 6.20. Еще один вариант правила Find-data

```
(defrule   Find-data
           (data ?x&: (floatp ?x)&:(> ?x 0) $?y ?z&:(stringp ?z) )
           =>
           (printout t "x=" ?x " y=" ?y " z=" ?z crlf ) )
```

Выше приведен еще один вариант правила Find-data. В данном случае ищется факт неявно созданного шаблона data, первое поле которого — вещественное число больше нуля, а последнее — строка. В нашем списке фактов такому правилу удовлетворяет только факт с индексом 1 — (data 1.0 blue "red") .

Ограничения, возвращающие значения

В ограничениях возможно использование значений, возвращенных некоторыми функциями (в том числе и внешними). Вызов функции записывается с помощью знака = и указанной за ним функцией.

> **Замечание**
> Функция сравнения также использует знак =. Разница между ними может быть определена по контексту.

Возвращаемое значение должно быть одним из простых типов данных CLIPS. Это значение, возвращенное функцией, объединяется с образцом так, как если бы оно было символьным ограничением. Заметьте, что функция вычисляется при каждом сопоставлении образцов, а не один раз при определении правила.
Ограничения, возвращающие значения, имеют следующий синтаксис:

Определение 6.13. Синтаксис ограничения, возвращающего значение

=<вызов-функции>

Определения понятия "элемент", приведенные в предыдущем разделе, примут такой вид:

Определение 6.14. Синтаксис понятия "элемент"

```
<элемент>      ::= <константа> |
                   <простая-переменная>|
                   <составная-переменная>|
                   :<вызов-функции>
                   =<вызов-функции>
```

Правило из примера 6.21 выводит на экран такие факты data, в которых значение второго поля в два раза больше, чем значение первого. В нашем случае это факты (data 1 2) И (data 2 4).

Пример 6.21. Использование ограничения, возвращающего значение

```
(assert  (data I 2)
         (data 2 3)
         (data 24))
(defrule  Find-data
         (data ?x ?y&=(* 2 ?x))
         =>
         (printout  t  "x="   ?x  "  y="   ?y crlf))
```

Сопоставление образцов с объектами

Во всех приведенных выше примерах образцы сопоставлялись с фактами из списка фактов. Кроме этого, образцы можно сопоставлять с экземплярами объектов — экземпляров, определенных пользователем классов на языке COOL (см. гл. 11). Такие образцы называются образцами объектов. Образцы могут сопоставляться с объектами, спецификация которых определена до создания образца и которые находятся в границах видимости текущего модуля. Любой класс, который имеет объекты, соответствующие образцу, не может быть удален или изменен, пока не будет удален образец. Даже если правило удалено с помощью действий, выполняемых в собственной правой части, класс, связанный с образцом, не может быть изменен до тех пор, пока правая часть правила не закончит работу.

При создании или удалении объекта все образцы, подходящие этому объекту, обновляются.
Однако в случае изменения слота объекта обновляются только те образцы, которые явно сопоставляются по этому слоту. Таким образцом можно использовать логические зависимости для обработки изменений некоторых слотов.
Изменение неактивных слотов или объектов неактивных классов не оказывает никакого воздействия на правила.

Определение 6.15. Синтаксис образцов объектов

<образец объекта> ::= (object <атрибуты-ограничения>)
<атрибуты-ограничения> ::= (is-a <ограничение>)|
(name <ограничение>)|
(slot <ограничение>)

Ограничение is-a (является) используется для определения ограничений класса, таких как "Является ли этот объект экземпляром заданного класса?". Ограничение is-a также определяет, является ли объект экземпляром класса, который является наследником класса, заданного в ограничении, в случае если это не будет явно запрещено образцом.
Ограничение name используется для определения конкретного объекта с заданным именем. Имя, задаваемое в данном ограничении, должно быть значением типа instance-name, а не значением типа symbol, как обычно. Ограничения для составных полей (такие как $?) не могут использоваться с ограничениями is-a и name. Эти ограничения применяются в работе со слотами объектов так же, как и при работе со слотами шаблонов. Как и в случае образцов для шаблонов, имена слотов для образца объекта должны быть значениями типа symbol.
Приведем несколько примеров использования образцов объектов.

Пример 6.22. Использование образцов объектов

```
(defrule example-1
        (object (is-a MyObj1 | MyObj2) )
        =>)
(defrule example-2
        (object (is-a ?x) )
        (object (is-a ~?x) }
        =>)
(defrule example-3
        (object (width ?x&:(> ?x 20)))
        =>)
(defrule example-4
        (object (width ?x) (height ?x))
        =>)
```

Первое правило удовлетворяет любой объект класса MyObj1 или MyObj2. Второе правило активируется любой парой объектов, принадлежащей разным классам. Третье правило выполняется в случае, если будет найден объект активного класса, содержащий активный слот width, значение которого больше 20. Последний приведенный пример удовлетворяется любым объектом активного класса, содержащим активные слоты width и height, значения которых должны быть равны.

Адрес образца

Некоторые действия в правой части правил, такие как retract и unmake-instance, оперируют с фактами или объектами, участвующими в левой части. Для того чтобы определить, какой факт или объект будет изменяться, необходимо присвоить переменной адрес конкретного факта или объекта. Присваивание адресов происходит в левой части правила и полученное значение называется *адресом образца* (pattern-address).

Определение 6.16. Синтаксис адреса образца

<адрес-образца> ::= ?<имя-переменной> <- <образец>

Стрелка влево (<-) — необходимая часть синтаксиса. Переменная, связанная с адресом факта или объекта, может сравниваться с другой переменной или использоваться внешней функцией. Переменная, связанная с адресом факта или объекта, может быть также использована для последующего ограничения полей в образце условного выражения. Однако нельзя связывать переменную в условном выражении not.

В качестве примера приведем простое правило, которое удаляет все факты data.

Пример 6.23. Правило del-data-facts

```
(defrule  del-data-facts
          ?data-facts <-  (data  $?)
          =>
          (retract  ?data-facts))
```

На этом рассмотрение синтаксиса и способов использования условного элемента *образец (pattern CE)* можно считать завершенным. Как вы уже успели убедиться, это довольно сложная конструкция языка CLIPS. Утешением может послужить то, что остальные условные элементы (test, and, or, not, exists, forall и logical) гораздо проще используют образцы в качестве основы. Их рассмотрением мы и займемся в следующих разделах.

6.5.2. Условный элемент *test*

Условный элемент test предоставляет возможность наложения дополнительных ограничений на слоты фактов или объектов. Элемент test удовлетворяется, если вызванная в нем функция возвращает значение НЕ-FALSE. Как и в случае предикатных ограничений образца в условном элементе test, можно использовать переменные, уже связанные со своими значениями. Внутри элемента test могут быть выполнены различные логические операции, например сравнения переменных.

Определение 6.17. Синтаксис условного элемента test

<условный-элемент-test > ::= (test <вызов-функции>)

Выражение test вычисляется каждый раз при удовлетворении других условных элементов. Это означает, что условный элемент test будет вычислен больше одного раза, если обрабатываемое выражение может быть удовлетворено более чем одной группой данных. Использование условного элемента test может стать причиной автоматического добавления правилу некоторых условных выражений. Кроме того, CLIPS может автоматически переупорядочивать условные элементы test *(см. разд. 6.5.9).*

Приведенное ниже правило находит пару фактов data, причем разница между значениями первых полей этих фактов должна быть больше или равной 3.

Пример 6.24. Применение условного элемента test

```
(defrule  example
          (data ?x)
          (data ?y)
          (test (>= (abs (- ?y ?x) ) 3))
          =>)
```

Условный элемент test может привести к автоматическому добавлению образцов initial-fact или initial-object в левую часть правила. Поэтому не забывайте использовать команду reset (которая

создает initial-fact и initial-object), чтобы быть уверенным в корректной работе условного элемента test.

6.5.3. Условный элемент *or*

Условный элемент or позволяет активировать правило любым из нескольких заданных условных элементов. Если какой-нибудь из условных элементов, объединенных с помощью or, удовлетворен, то и все выражение or считается удовлетворенным. В этом случае, если все остальные условные элементы, входящие в левую часть правила (но не входящие в or), также удовлетворены, правило будет активировано. Условный элемент or может объединять любое количество элементов.

Замечание

Правило будет активировано для каждого выражения в условном элементе or, которое было удовлетворено. Таким образом, условный элемент or производит эффект, идентичный написанию нескольких правил с похожими посылками и следствиями.

Определение 6.18. Синтаксис условного элемента or

<условный-элемент-or > : : = (or <условный-элемент>+)

Пример 6.25. Применение условного элемента or

```
(defrule system-fault
        (error-status unknown) (or (temp high)
            (valve broken)
            (pump off))
            =>
            (printout t "The system has a fault." crlf))
```

Данное правило сообщит о поломке системы, если в списке фактов будет присутствовать факт error-status unknown и один из фактов temp high, valve broken или pump off. В случае если будут присутствовать два из этих трех фактов, например temp high и pump off, то сообщение будет выведено два раза. Заметьте, что приведенный пример — точный эквивалент следующих трех отдельных правил:

Пример 6.26. Эквивалент правилу system-fault

```
(defrule system-fault-1
        (error-status unknown)
        (pump off)
        =>
        (printout t "The system has a fault." crlf))
(defrule  system-fault-2
        (error-status  unknown) (valve broken)
        =>
        (printout t "The system has a fault." crlf))
(defrule system-fault-3
        (error-status unknown) (temp high)
        =>
        (printout t "The system has a fault." crlf))
```

6.5.4. Условный элемент *and*

Все условные элементы в левой части правил CLIPS объединены неявным условным элементом and. Это означает, что все условные элементы, заданные в левой части, должны удовлетвориться, для того чтобы правило было активировано. С помощью явного применения условного элемента and можно смешивать различные условия and и or и группировать элементы так, как этого требует логика правил. Условие and удовлетворяется, только если все условия внутри

явного and удовлетворены. В случае, если остальные условия в левой части правила также истинны, правило будет активировано. Элемент and может объединять любое число условных элементов.

Определение 6.19. Синтаксис условного элемента and

<условный-элемент-and> ::= (and <условный-элемент>+)

Пример 6.27. Применение условного элемента and

```
(defrule system-flow
        (error-status confirmed)
        (or   (and  (temp high)
                    (valve closed))
(and  (temp low)
                     (valve open)))
=>
(printout t "The system is having a flow problem. " crlf))
```

Если условный элемент and содержит условные элементы test или not в качестве первого элемента, то перед ними автоматически добавляется образец initial-fact или initial-object. Помните, что левая часть любого правила содержит неявный элемент and, поэтому приведенное в примере 6.28 правило будет автоматически преобразовано (см. пример 6.29).

Пример 6.28. Правило nothing-to-schedule

```
(defrule  nothing-to-schedule
        (not   (schedule   ?))
        =>
        (printout  t   "Nothing to  schedule."  crlf))
```

Пример 6.29. Преобразованное правило nothing-to-schedule

```
(defrule nothing-to-schedule
        (and (initial-fact)
             (not (schedule ?)))
        =>
        (printout t "Nothing to schedule." crlf))
```

6.5.5. Условный элемент *not*

Иногда важнее отсутствие информации, а не ее присутствие, т. е. возникают ситуации, когда необходимо запустить правило, если образец или другой условный элемент не удовлетворяется (например, факт не существует). Условный элемент not предоставляет эту возможность. Элемент not удовлетворяется, только если условный элемент, который он содержит, не удовлетворяется.

Определение 6.20. Синтаксис условного элемента not

<условный-элемент-not> ::= (not <условный-элемент>)

Условный элемент not может отрицать только одно выражение. Несколько условных элементов нужно отрицать с помощью нескольких элементов not. Тщательно следите за комбинациями not с or или and; результат не всегда очевиден!

Пример 6.30. Применение условного элемента not

```
(defrule high-flow-rate
        (temp high)
        (valve open)
        (not (error-status confirmed))
        =>
        (printout t "Recommend closing of valve due to high temp" crlf))
```

В логическом элементе not можно использовать связанные переменные, так же как и в других условных элементах:

Пример 6.31. Правило check-value

```
(defrule check-valve
        (check-status  ?valve)
        (not   (valve-broken  ?valve))
        =>
        (printout  t  "Device   "   ?valve   "   is  OK"  crlf))
```

С помощью условного элемента not можно, наконец, довести до совершенства наше правило Find-2-coeval-Person, последняя версия которого была приведена в *разд. 6.5.1*. Если вы помните, это правило выводит всевозможные пары персон одинакового возраста. Чтобы данное правило не выводило эквивалентные по смыслу пары имен (например, Bob-Sue и Sue-Bob), преобразуем нашу программу следующим образом:

Пример 6.32. Улучшенное правило Find-2-coeval-Person

```
(deftemplate person
            (slot name)
            (slot age))
(deftemplate person-pair
            (slot namel)
            (slot name2)
            (slot age))
(deffacts   people
            (person (name Joe) (age 20))
            (person (name Bob) (age 20))
            (person (name Joe) (age 34))
            (person (name Sue) (age 34))
            (person (name Sue) (age 20)))
(defrule    Find-2-Coeval-Person
            (person (name ?x) (age ?z))
            (person (name ?ys~?x) (age ?z))
            (not (person-pair (namel ?x) (name2 ?y) (age ?z)))
            (not (person-pair (namel ?y) (name2 ?x) (age ?z)))
            =>
            (printout t "name=" ?x " name=" ?y " age=" ?z crlf)
            (assert (person-pair (namel ?x) (name2 ?y) (age ?z))))
```

Обратите внимание на произведенные изменения. Во-первых, с помощью конструктора deftemplate был Добавлен дополнительный шаблон person-pair. В фактах, соответствующих данному шаблону, будет храниться информация об уже найденных парах ровесников. Кроме того, было сильно изменено и само правило. В его левой части было добавлено два условия:

```
(not (person-pair (namel ?x) (name2 ?y) (age ?z)))
(not (person-pair (namel ?y) (name2 ?x) (age ?z)))
```

Эти условные элементы проверяют наличие фактов типа person-pair и, тем самым отслеживают, была ли уже обработана данная пара или ее перестановка. Если эти факты отсутствуют, то это означает, что обработка еще не была выполнена. В этом случае правило активируется, и выполняются действия, описанные в правой части правила. А именно выводится на экран сообщение о найденной паре ровесников и добавляется соответствующий факт person-pair, утверждающий, что данная пара уже была обработана. Для запуска программы выполните команды reset и run. Программа выведет на экран следующую информацию:

Пример 6.33. Результат работы правила Find-2-Coeval-Person

```
name=Sue name=Bob  age=20
name=Sue name=Joe  age=20
name=Sue name=Joe  age=34
name=Bob name=Joe  age=20
```

Если вы внимательно посмотрите на полученный результат и исходные данные, то обнаружите, что это именно то, что нам было нужно. Это список всевозможных ровесников без повторений и с исключением того факта, что все люди являются ровесниками сами себе. Теперь наше правило достигло полного совершенства! Обратите внимание на тот факт, что если вы повторно попробуете выполнить команду run, то ничего не увидите. Это происходит потому, что в списке фактов содержится информация обо всех обработанных парах, оставшаяся после первого запуска. Для того чтобы повторно запускать данный пример, выполняйте команду reset перед каждой командой run.

Условный элемент not, так же как и test, может привести к автоматическому добавлению образцов initial-fact или initial-object в левой части правил. Поэтому не забывайте использовать команду reset (которая создает initial-fact и initial-object), чтобы быть уверенным в корректной работе условного элемента not.

В условный элемент not, содержащий элемент test, автоматически преобразуется в элемент not, содержащий and с initial-fact и исходным элементом test. Например, следующий условный элемент из примера 6.34 преобразуется в элемент из примера 6.35.

Пример 6.34. Условный элемент not, содержащий элемент test

```
(not   (test   (>  ?time-l   ?time-2)))
```

Пример 6.35. Преобразованный условный элемент not, L содержащий элемент test

```
(not     (and   (initial-fact)
                (test   (>  ?time-l  ?time-2))))
```

Замечание

Заметьте, что наиболее простым и правильным способом записи данного выражения будет: (test (not (> ?time-1 ?time-2))).

6.5.6. Условный элемент *exists*

Условный элемент exists позволяет определить, существует ли хотя бы один набор данных (фактов или объектов), которые удовлетворяют условным элементам, заданным внутри элемента exists.

Определение 6.21. Синтаксис условного элемента exists

<условный-элемент-exists> ::= (exists <условный-элемент>+)

CLIPS автоматически заменяет exists двумя последовательными условными элементами not. Например, следующее правило (пример 6.36) будет преобразовано в правило из примера 6.37.

Пример 6.36. Правило example

```
(defrule example
        (exists   (a  ?x)    (b  ?x))=>)
```

Пример 6.37. Преобразованное правило example

```
(defrule example
          (not (not (and (a ?x) (b ?x))))=>)
```

Так как внутренний способ реализации exists использует условный элемент not, то для exists справедливы все замечания и ограничения, приведенные в предыдущем разделе.

Рассмотрим следующий пример:

Пример 6.38. Использование условного элемента exists

```
(deftemplate     hero
                 (multislot name)
                 (slot status  (default unoccupied)))
(deffacts        goal-and-heroes
                 (goal save-the-world)
                 (hero   (name Death Defying Man))
                 (hero   (name  Stupendous Man))
                 (hero   (name  Incredible Man)))
(defrule         save-the-world
                 (goal  save-the-world)
                 (exists   (hero   (status unoccupied)))
                 =>
                 (printout t  "The day is  saved."  crlf))
```

Данная программа определяет шаблон — героя, имеющего составное поле с именем героя и простое поле, содержащее статус "не занят" по умолчанию. Конструктор deffacts определяет трех ничем не занятых героев и текущую цель — спасение мира. Правило проверяет, есть в данный момент эта цель, и в случае положительного ответа проверяет, если ли какой-нибудь еще не занятый герой. Если все условные элементы правила удовлетворены, оно сообщает, что мир спасен. Обратите внимание: несмотря на то, что у нас все три героя не заняты, правило будет активировано только один раз.

Так как способ реализации exists использует условный элемент not, то условный элемент exists может привести к автоматическому добавлению образцов initial-fact или initial-object в левую часть правила. Поэтому не забывайте использовать команду reset (которая создает initial-fact и initial-object), чтобы быть уверенным в корректной работе условного элемента exists.

6.5.7. Условный элемент *forall*

Условный элемент forall позволяет определить, что некоторое заданное условие выполняется для всех заданных условных элементов.

Определение 6.22. Синтаксис условного элемента forall

<условный-элемент fora11> ::= (forall <условный-элемент>
<условный-элемент>+)

CLIPS автоматически заменяет f o rail комбинацией условных элементов not и and. Например, следующее правило (пример 6.39) будет преобразовано так, как показано в примере 6.40.

Пример 6.39. Правило example

```
(defrule  example
          (forall   (a  ?x)    (b ?x)    (c  ?x))=>)
```

Пример 6.40. Преобразованное правило example

```
(defrule example
         (not     (and   (a ?x)
                         (not   (and       (b ?x)
                                           (c  ?x)))))=>)
```

Рассмотрим следующий пример. Правило all-students-passed определяет, прошли ли все студенты чтение, письмо и арифметику, используя условие forall:

Пример 6.41. Правило all-students-passed

```
(defrule all-students-passed
         (forall (student ?name)
                 (reading ?name)
                 (writing ?name)
                 (arithmetic ?name))
         =>
         (printout t "All students passed." crlf))
```

Заметьте, что данное правило удовлетворяется, пока нет ни одного студента. При добавлении факта (student Bob) правило перестает удовлетворяться, т. к. нет фактов, подтверждающих, что Bob прошел все необходимые предметы. Правило не начнет удовлетворяться и после добавления фактов (reading Bob) и (writing Bob). А вот после добавления факта (arithmetic Bob) правило будет активировано и сможет вывести на экран соответствующую запись. Если добавить факт (student John), правило опять перестанет удовлетворяться, т. к. один из студентов (John) не прошел все необходимые предметы. Используя условный элемент exists, вы без труда сможете изменить это правило так, чтобы оно не выполнялось в случае отсутствия студентов.
Так как реализация forall использует условный элемент not, то forall, так же как и not, test и exists, может привести к автоматическому добавлению образцов initial-fact или initial-object в левую часть правила. **Не** забывайте использовать команду reset для корректной работы этого условного элемента.

6.5.8. Условный элемент *logical*

Условный элемент logical предоставляет механизм поддержки достоверности для созданных правилом данных (фактов или объектов), удовлетворяющих образцам. Данные, созданные в правой части правила, могут иметь логическую зависимость от данных, удовлетворивших образцы в левой части правила. Такая зависимость называется *логической поддержкой.* Данные могут зависеть от группы данных или нескольких групп данных, удовлетворивших одно или несколько правил. Если удаляются данные, которые поддерживают некоторые другие данные, то зависимые данные также автоматически удаляются.

Если некоторые данные созданы без логической поддержки (например, с помощью конструкторов deffacts, definstance или команды assert, введенной пользователем или вызванной в правой части правила), то считается, что они имеют безусловную поддержку. Безусловная поддержка удаляет все присутствующие в данный момент условные поддержки этих данных (но не удаляет сами данные). Дальнейшая логическая поддержка для данных с безусловной поддержкой игнорируется. Удаление правила, которое вызвало логическую поддержку для данных, удаляет логическую поддержку,

сгенерированную этим правилом (но не удаляет данные, если у них еще есть логическая поддержка, сгенерированная другим правилом).

Определение 6.23. Синтаксис условного элемента logical

<условный-элемент-1од!са1> ::= (logical <условный-элемент>+)

Условный элемент logical группирует образцы, так же как это делает and. Данное свойство можно использовать при объединении элементов and, or и not. Однако только первые *n* образцов правила могут использоваться в условном элементе logical. Например, следующее правило записано верно:

Пример 6.42. Правильный вариант использования условного элемента logical

```
(defrule      ok
              (logical (a))
              (logical (b))
              (c)
              =>
              (assert (d)))
```

А такое объявление правил недопустимо:

Пример 6.43. Неправильные варианты использования условного элемента logical

```
(defrule not-ok-1
        (logical (a))
        (b)
        (logical (c))
        =>
        (assert (d)))
(defrule not-ok-2
        (a)
        (logical (b))
        (logical (c))
        =>
        (assert (d)))
(defrule not-ok-3
        (or (a)
        (logical (b)))
        (logical (c))
        =>
        (assert (d)))
```

Рассмотрим следующий пример. Включите просмотр списка фактов с помощью пункта **Facts Window** меню **Windows,** это поможет следить за тем, что происходит в момент выполнения программы.

Пример 6.44. Использование условного элемента logical

```
(clear)
(reset)
(defrule example
        (logical   (a))
        (b)
```

```
        =>
        (assert (c)))
(assert (a) (b) )
(run)
(retract 2)
(retract 1)
```

По команде run правило example, активированное фактами a и b, добавляет новый факт c, который имеет логическую поддержку (зависит) от факта a.

После удаления факта b с помощью команды (retract 2) ничего особенного не происходит, но если мы удалим факт a, то увидим, что это тут же приведет к удалению связанного с ним факта c.

Как упоминалось в *подразделе "Сопоставление образцов с объектами"разд. 6.5.1,* условный элемент logical может быть использован для создания данных, которые будут логически связаны с изменениями некоторых отдельных слотов объекта, а не от всего объекта целиком. Данную возможность можно использовать только при работе с объектом. При работе с шаблонами фактов данную возможность использовать нельзя, т. к. изменения слота факта, как было рассмотрено в *разд. 5.2.5,* фактически приводит к удалению старого факта и добавления нового с измененными слотами и индексом. В отличие от фактов изменения слотов объекта выполняются без удаления объекта. Это поведение иллюстрируется приведенным ниже примером:

Пример 6.45. Использование условного элемента logical с объектами

```
(clear)
(defclass A (is-a USER)
            (role concrete)
            (pattern-match reactive)
            (slot foo (create-accessor write))
            (slot bar (create-accessor write)))
(defrule  match-A
            (logical (object (is-a A) (foo ?)))
            =>
            (assert (new-fact)))
(make-instance a of A)
run)
send [a] put-foo 100)
```

После выполнения команды run правило match-A добавляет факт new-fact, логически связанный с конкретным значением слота foo объекта a. При изменении значения данного слота факт new-fact автоматически удаляется из списка фактов.

6.5.9. Автоматическое добавление и перегруппировка условных элементов

В некоторых ситуациях CLIPS автоматически добавляет дополнительные образцы к левой части правил (обычно для улучшения алгоритма сопоставления образцов, используемого системой CLIPS). Существует два образца, применяемых CLIPS по умолчанию: образец факта initial-object и образец объекта initial-object.

Ниже приводится определение этих данных:

Определение 6.24. Синтаксис предопределенного факта и объекта

```
(initial-fact)
(object (is-a INITIAL-OBJECT) (name [initial-object]))
```

Безусловные правила

Если правило не содержит условных элементов в своей левой части, то к предпосылкам правила автоматически добавляется образец initial-fact (конфигурацию CLIPS можно настроить таким образом, чтобы вместо образца факта добавлялся образец объекта initial-object). Например, следующее правило из примера 6.46 будет преобразовано так, как показано в примере 6.47:

Пример 6.46. Правило без условий

```
(defrule  example
          => )
```

Пример 6.47. Преобразованное правило без условий

```
(defrule  example
          (initial-fact)
           =>)
```

Использование элементов *test» not* перед *and*

Условные элементы test и not, стоящие перед and, добавляют образец initial-fact или initial-object непосредственно перед собой. Образец initial-fact добавляется, если в первом условном элементе используется образец факта. Образец initial-object добавляется, если в первом условном элементе используется образец объекта. Если в первом условном элементе нет образцов, то тип добавляемого образца определяется по следующему условному элементу таким же методом. Если во всем текущем условном выражении нет образцов, то система использует предопределенный факт initial-fact (хотя конфигурацию CLIPS можно настроить таким образом, чтобы вместо образца initial-fact добавлялся образец initial-object). Например, следующие правила из примера 6.48 будут изменены так, как в примере 6.49.

Пример 6.48. Правила с условиями test и not перед and

```
(defrule      example1
              (test   (>  80   (startup-value)))
              =>
(defrule  example2
              (test (> 80 (startup-value)))
              (object (is-a MACHINE))
              =>)
(defrule  example3
              (machine ?x)
              (not (and (not (part ?x ?y))
                                (inventoried ?x)))
              =>
```

Пример 6.49. Преобразованные правила с условиями test и not перед and

```
(defrule  example1
              (initial-fact)
              (test (> 80 (startup-value)))
              =>)
(defrule  example2
              (object (is-a INITIAL-OBJECT) (name [initial-object]))
              (test (> 80 (startup-value)))
              (object (is-a MACHINE))
              =>)
```

```
(defrule  example3
          (machine ?x)
          (not (and (initial-fact)
                    (not (part ?x ?y))
                    (inventoried ?x)))
          =>)
```

Использование элемента *not* перед *test*

Если сразу перед условным элементом test использовался условный элемент not, то CLIPS автоматически перемещает условный элемент not на место первого условия непосредственно следующего за test. Например, правило из примера 6.50 изменится на эквивалентное (пример 6.51):

Пример 6.50. Правило с условиями not перед элементом test

```
(defrule      example
              (a   ?x)
              (not   (b   ?x))
              (test   (>  ?x  5)}
              =>)
```

Пример 6.51. Преобразованное правило с условиями not перед test

```
(defrule      example
              (a  ?x)
              (test   (>   ?x  5))
              (not   (b  ?x) )
               =>)
```

Использование элемента *not* перед or

Если сразу перед условным элементом or использовался условный элемент not, то CLIPS автоматически заменяет комбинацию not/or на эквивалентную комбинацию and/not. Например, следующее правило (пример 6.52) будут изменено так, как показано в примере 6.53.

Пример 6.52. Правило с условиями not перед or

```
(defrule  example
          (a ?x)
          (not (or (b ?x)
                   (c ?x)))
=>)
```

Пример 6.53. Преобразованное правило с условиями not перед or

```
(defrule      example
              (a  ?x)
              (and   (not   (b  ?x))
                     (not   (c  ?x)))
              =>)
```

Замечания об автоматическом добавлении и перегруппировке условных элементов

В завершение описания синтаксиса левой части правил CLIPS обратим внимание на следующие важные особенности:

1. Полная версия левой части правила содержит неявный условный элемент and.
2. Преобразование условных элементов forall и exists к эквивалентным выражениям с помощью not и and выполняется перед добавлением соответствующих образцов в левую часть правила.
3. Условный элемент test обычно не используется в качестве первого элемента в условии and.
4. Команды, выводящие информацию об условных элементах в левой части правила, отображают информацию об определении правила в виде, в котором ее задал пользователь. Информация о перегруппировке и добавлении образцов initial-fact и initial-object не выводится.

6.6. Команды и функции для работы с правилами

После того как мы полностью разобрались с представлением правил в CLIPS, рассмотрели внутренние алгоритмы обработки правил, стратегии разрешения конфликтов и синтаксис левой части правил, можно смело переходить к изучению функций и команд, предоставляемых CLIPS для работы с правилами. Полная спецификация этих функций будет дана в *гл. 15* и *16,* в данной главе мы рассмотрим лишь основные из них с примерами использования.

6.6.1. Просмотр и удаление существующих правил

После создания правил с помощью конструктора defrule вполне естественно возникает желание сделать что-нибудь с уже существующим правилом. CLIPS поддерживает множество различных команд, оперирующих с правилами. В данном разделе мы рассмотрим наиболее часто используемые команды: ppdefrule,list-defrules и undefrule.

С помощью команды ppdefrule можно просмотреть определение правила в том виде, в котором оно было создано с помощью конструктора defrule.

Определение 6.25. Синтаксис команды ppdefrule

(ppdefrule <имя-правила>)

Для того чтобы получить полный список правил, присутствующих в CLIPS в данный момент, используется команда list-def rules.

Определение 6.26. Синтаксис команды list-defrules

(list-defrules <имя-модуля>)

Полный синтаксис этой команды содержит необязательный аргумент <имя-модуля> (о понятии модуля будет рассказано в *гл. 12).* Если данный аргумент не задан, то будет выведен список правил, определенных в текущем модуле. В случае явного задания модуля будет список правил, принадлежащих конкретному модулю. Данный аргумент может принимать значение *. В этом случае на экран будет выведен список всех правил из всех модулей.

Для удаления правила используется команда undefrule.

Определение 6.27. Синтаксис команды undefrule

(undefrule <имя-правила>)

В качестве параметра команда undefruie принимает имя правила, которое нужно удалить. Если в качестве имени правила был задан символ *, то будут удалены все правила.

Для демонстрации работы команд, приведенных в этом и последующих разделах, будем использовать следующие правила:

Пример 6.54. Необходимые для дальнейшей работы правила

```
(defrule      MakeC
              (a)
              (b)
              =>
              (assert (c)))
(defrule      MakeD
              (c)
              (or (a)
              (b))
              =>
              (assert (d)))
(defrule       MakeE
              (d)
              (or (a)
              (b)
              (c))
              =>
              (assert (e)))
```

Введите эти правила в среду CLIPS, а затем выполните следующую последовательность команд:

Использование команд ppdefrule, list-defrules и undefrule

```
(ppdefrule MakeE)
(list-defrules)
(undefrule MakeD)
(list-defrules)
(undefrule *)
(list-defrules)
```

Если приведенные выше действия были выполнены правильно, то полученный результат должен соответствовать рис. 6.6.

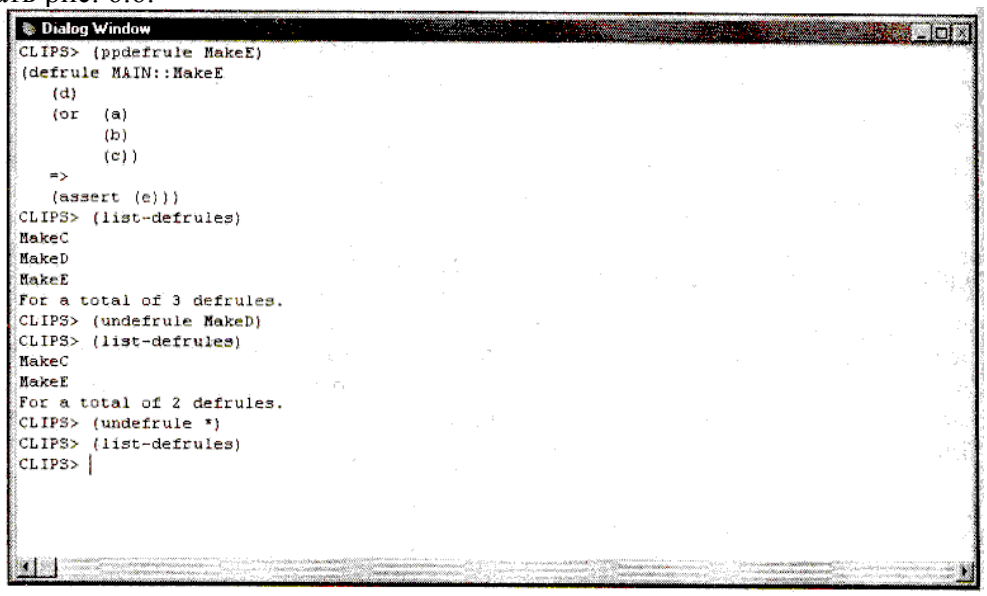

Рис. 6.6. Результат применения команд ppdefrule, list-defrules и undefrule

Как уже упоминалось в *разд. 6.1,* пользователям Windows-версии CLIPS доступен инструмент под названием **Defrule Manager** (Менеджер правил). Если в данный момент в среде CLIPS отсутствуют правила, то пункт **Defrule Manager** меню **Browse** не будет доступен. Если вы повторно заведете приведенные выше правила и откроете менеджер правил, то должны будете увидеть результат, приведенный на рис. 6.7. Менеджер отображает список всех правил, доступных в данный момент. Общее количество правил отображается в заголовке окна менеджера, в данный момент это **Defrule Manager — 3 Items.** С помощью кнопок **Remove и Pprint** можно удалять и выводить определение выбранного правила соответственно. Вся информация, получаемая от менеджера правил, отображается непосредственно в главном окне CLIPS.

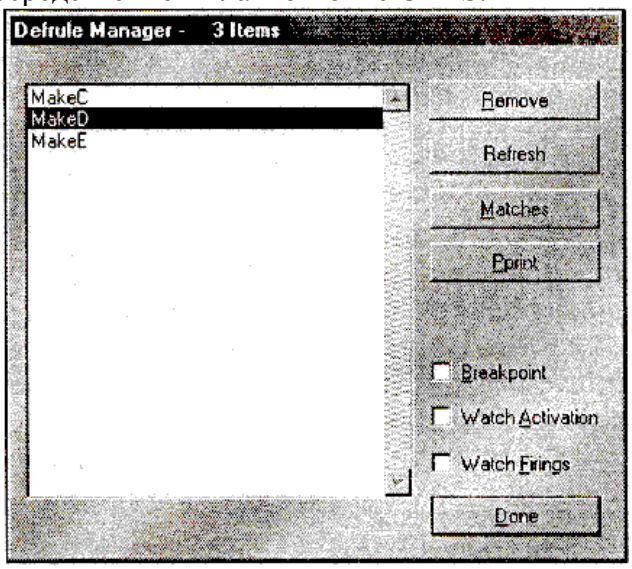

Рис. 6.7. Просмотр списка правил с помощью менеджера правил

CLIPS не содержит специальных команд для изменения существующих правил. Чтобы изменить существующее правило, пользователю необходимо заново определить такое правило с помощью конструктора def rule. При этом существующее определение правила будет автоматически удалено из системы, даже если новый конструктор содержал ошибки, и новое правило добавлено не было.

6.6.2. Сохранение правил

Как вы уже успели убедиться, создавать правила конструктором defrule каждый раз, по мере необходимости используя для этого среду CLIPS, довольно неудобно. Для облегчения участи пользователя CLIPS позволяет загружать конструкторы правил (как, впрочем, и все остальные конструкторы) из текстового файла. Для этого используется следующая команда:

Определение 6.28. Синтаксис команды load

(load <имя-файла>)

Имя файла должно быть строкой, т. е. заключаться в кавычки. Имя файла может содержать полный путь к файлу. В противном случае система будет искать файл в текущем каталоге. Для создания файла в принципе можно использовать любой ASCII-редактор, но лучше применять встроенный редактор, предоставляемый средой CLIPS. Встроенный редактор поддерживает несколько дополнительных функций, чрезвычайно полезных при разработке программ. Во-первых, он способен проверять синтаксис функций, баланс открывающих и закрывающих скобок, помогает в расстановке и удалении комментариев и т. д. Если вы будете использовать встроенный редактор для создания серьезной экспертной системы, вы по достоинству оцените эти возможности. Во-вторых, встроенный редактор позволяет быстро загружать в среду отдельные конструкторы и команды. Эта возможность помогает проверять и тестировать большую экспертную систему. И, наконец, в-третьих, редактор предоставляет помощь по среде и языку, которая бывает чрезвычайно полезной, даже при наличии большого опыта работы в CLIPS. По умолчанию файлы, созданные во встроенном редакторе CLIPS, получают расширение clp. Для начала работы с редактором просто выберите пункт New меню **File.**

Создайте в CLIPS файл example1.CLP с тремя приведенными выше правилами. После чего очистите CLIPS с помощью команды clear и выполните команду (load "example1.CLP"). Полученный результат должен соответствовать рис. 6.8.

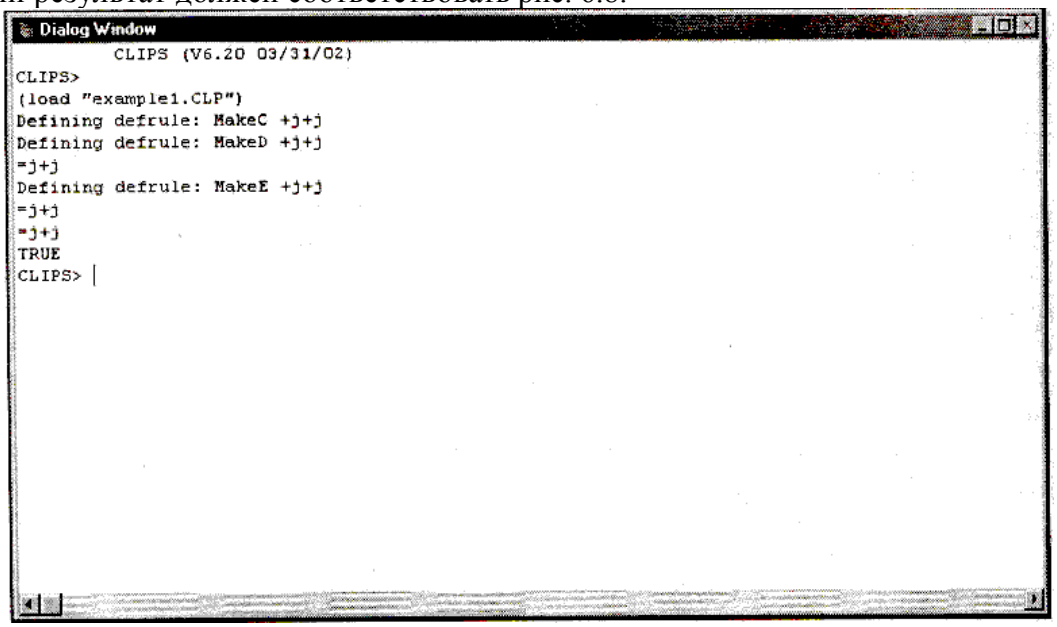

Рис. 6.8. Результат загрузки файла example1.CLP

Команда load отображает процесс загрузки каждого конструктора. В случае успешной загрузки всех определенных в файле конструкторов команда возвращает значение TRUE, в противном случае — информацию об ошибке. В случае если была найдена ошибка, процесс загрузки файла прекращается.

CLIPS поддерживает также команду load*. Эта команда полностью идентична load за исключением того, что она не отображает процесса загрузки конструкторов.

Определение 6.29. Синтаксис команды load*

(load* <имя-файла>)

CLIPS предоставляет также команду save, которая позволяет сохранять в текстовый файл все конструкторы, определенные в данный момент в системе. Синтаксис этой команды идентичен синтаксису команд load и load*.

Определение 6.30. Синтаксис команды save

(save <имя-файла>)

Текстовый формат не единственный способ хранения конструкторов CLIPS. Команды bsave и bload позволяют сохранять и загружать конструкторы в двоичном виде. Двоичные файлы загружаются гораздо быстрее, чем текстовые, но занимают больше места (т. к. кроме конструкторов они хранят полную информацию о текущем состоянии среды). Еще одним неудобством использования двоичных файлов является то, что создавать их можно только непосредственно в среде CLIPS.

Большинство описанных выше команд для работы с файлами (а именно load, save, bsave и bload) доступны в меню **File** Windows-версии среды CLIPS. Это команды **Load, Save, Save Binary и Load Binary** соответственно. Все используют стандартные Windows-диалоги для выбора файлов.

6.6.3. Запуск и остановка программы

Как было замечено, для запуска CLIPS-программ используется команда run.

Определение 6.31. Синтаксис команды run

(run <целочисленное-выражение>)

Целочисленное выражение является необязательным аргументом команды run. В простейшем случае в качестве этого аргумента можно использовать любую целую константу. Если данный аргумент задан и он положителен, то CLIPS запустит на выполнение заданное число правил из плана решения задачи. Если данное число больше числа правил в плане решения задачи, то будет запущены все правила. В случае если аргумент не задан или является отрицательным, план решения задачи также будет выполнен полностью.

В Windows-версии CLIPS в меню **Execution** доступны две версии команды run — **Run** и **Step.** Первая команда использует версию команды run без аргументов и запускает все правила из плана решения задачи. Программа Step позволяет трассировать программу и выполнять заданное число правил. По умолчанию это число равно 1, но эту установку среды можно изменить с помощью диалогового окна **Preferences.** Для запуска этого диалогового окна выберите пункт **Preferences** меню **Execution.** Общий вид диалогового окна показан на рис. 6.9. Количество правил, запущенных за один шаг трассировки, отображается в поле **Step Rule Firing Increment.**

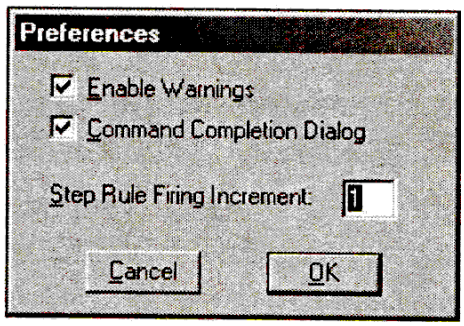

Рис. 6.9. Диалоговое окно **Preferences**

Кроме выполнения программы по шагам, CLIPS позволяет установку *точек останова* (breakpoints) на отдельных правилах.

Определение 6.32. Синтаксис команды set-break

(set-break <имя-правила>)

Если точка останова определена для заданного правила, то выполнение программы прекратится перед запуском этого правила. Точка останова не останавливает правило, если это первое правило в плане решения задачи.

Удалить точки останова можно с помощью команды remove-break.

Определение 6.33. Синтаксис команды remove-break

(remove-break <имя-правила>)

В случае выполнения команды remove-break без параметров CLIPS удалит все определенные ранее точки останова.

Устанавливать и снимать точки останова также можно с помощью менеджера правил, внешний вид которого представлен на рис. 6.7. Для этого выберите правило и установите флажок **Breakpoint.**

В случае если выполнение программы необходимо остановить, используйте команду halt без аргументов.

Определение 6.34. Синтаксис команды halt

(halt)

Диалоговое окно **Watch Options** (пункт **Watch** меню **Execution)** позволяет установить флажок **Statistics,** как показано на рис. 6.10.

В этом случае после выполнения каждой команды run CLIPS будет выводить статистическую информацию о количестве запущенных правил, полном и среднем времени выполнения правил, количестве добавленных фактов и т. д.

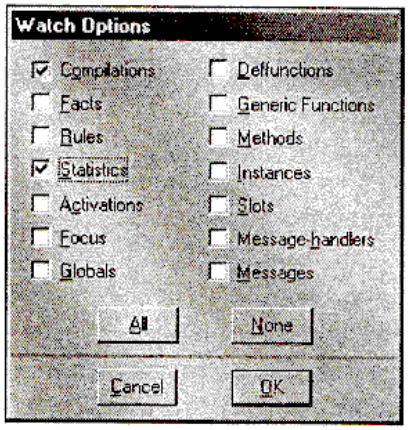

Рис. 6.10. Установка режима с выводом статистической информации

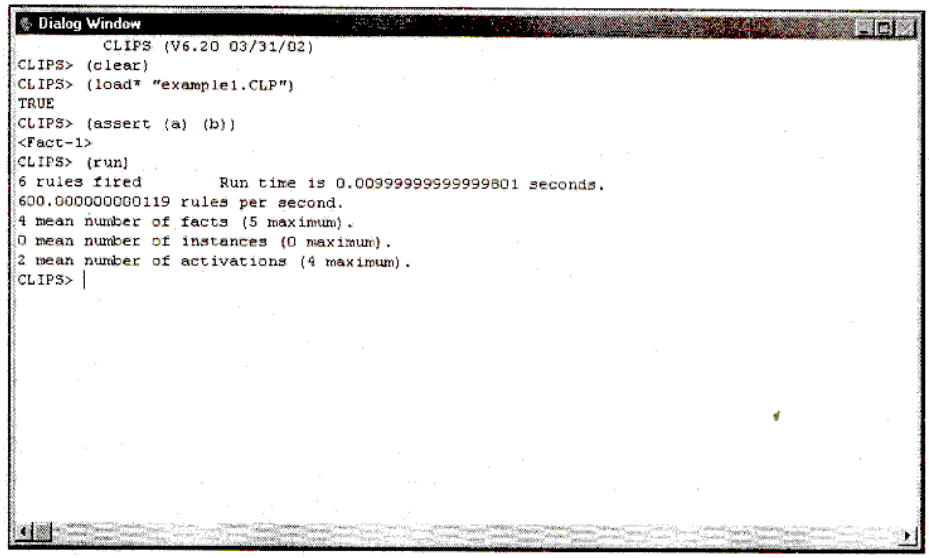

Рис. 6.11. Получение статистической информации

Если вы установите флажок **Statistics,** загрузите файл example 1.CLP, добавите факты (а) и (Ь) и запустите программу, то увидите результаты, представленные на рис. 6.11.

6.6.4. Просмотр плана решения задачи

План решения задачи (agenda) можно просматривать различными способами. Самый простой из них — команда agenda, набранная в главном окне CLIPS. Очистите CLIPS, загрузите файл example1.CLP, добавьте факты a, b, c и d и вызовите команду agenda. Полученный результат должен соответствовать приведенному на рис. 6.12.

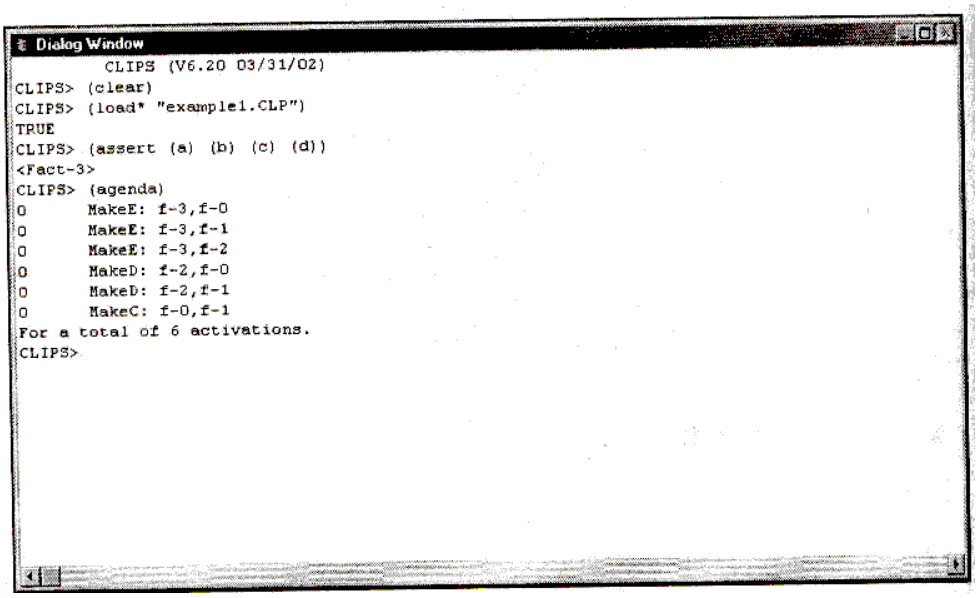

Рис. 6.12. Просмотр плана решения задачи

План решения задачи содержит 6 активаций правил. По команде agenda все эти активации будут выведены на экран вместе с приоритетом правил (слева от имени правила) и списком данных, активировавших правило (справа от имени правила). Порядок правил в плане решения задачи сильно зависит от выбранной стратегии разрешения конфликтов и приоритета правил.

Кроме этого, Windows-версия CLIPS позволяет выводить план решения задачи в отдельном окне — **Agenda.** Для того чтобы сделать окно видимым, воспользуйтесь пунктом **Agenda Window** меню **Window.** Внешний вид этого окна показан на рис. 6.13, его содержимое полностью соответствует информации, получаемой с помощью команды agenda. Данный инструмент чрезвычайно полезен при отладке программ или для наблюдения за изменением плана решения задачи в процессе выполнения программы.

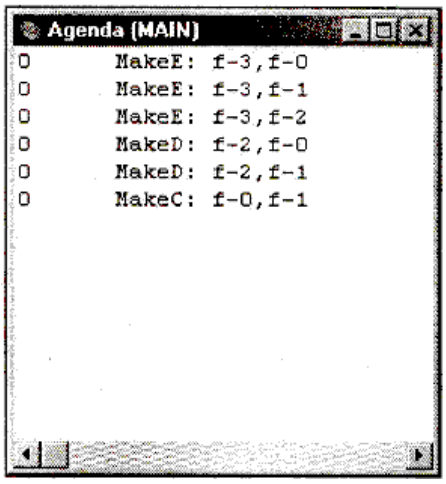

Рис. 6.13. Окно **Agenda**

Помимо окна **Agenda,** которое позволяет только просмотр, CLIPS предоставляет еще один удобный визуальный инструмент — **Agenda Manager** (Менеджер плана решения задачи), который позволяет в случае необходимости корректировать план решения задачи. Для вызова менеджера плана решения задачи выберите пункт **Agenda Manager** меню **Browse.** Внешний вид этого инструмента приведен на рис. 6.14. С его помощью можно удалять из плана решения задачи отдельные активации правил или запускать правила в некотором произвольном порядке.

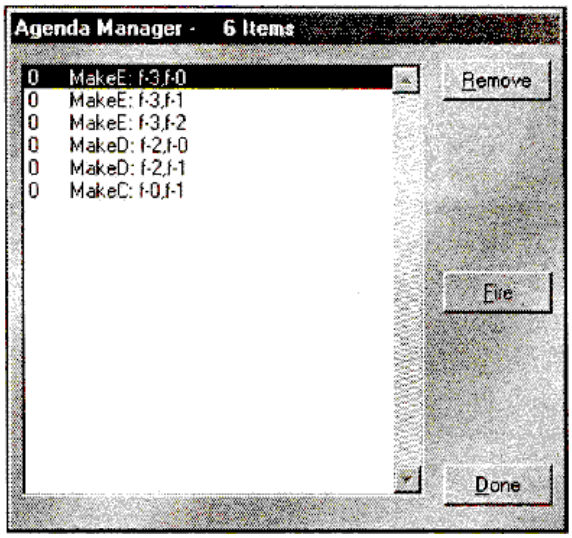

Рис. 6.14. Окно менеджера плана решения задачи

С помощью диалогового окна **Watch Options** (см. рис. 6.10) или менеджера правил можно задавать режим отображения активаций и/или запуска правил. В этом случае пользователь будет получать соответствующее информационное сообщение при добавлении правила в план решения задачи или при удалении правила из него, а также при каждом запуске правила.

6.6.5. Просмотр данных, способных активировать правило

CLIPS предоставляет возможность просматривать списки наборов данных (фактов или объектов), способных активировать заданное правило.

Определение 6.35. Синтаксис команды matches

(matches <имя-правила>)

Команда matches выводит информацию обо всех возможных наборах данных, способных активировать это правило. Посмотрите на результаты выполнения данной команды для правил MakeC и MakeD (наличие фактов a, b и c обязательно), приведенные на рис. 6.15 и 6.16 соответственно.

На этом мы закончим изучение синтаксиса правил CLIPS и основных команд и функций для работы с ними. Более полное описание команд и функций для работы с правилами приведено в *гл. 15* и *16*.

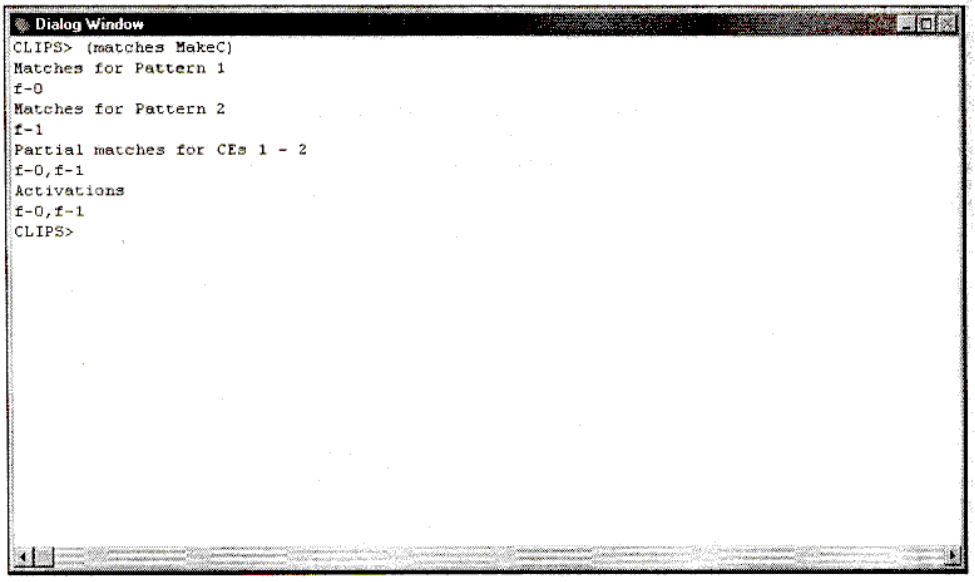

Рис. 6.15. Данные, активирующие правило MakeC

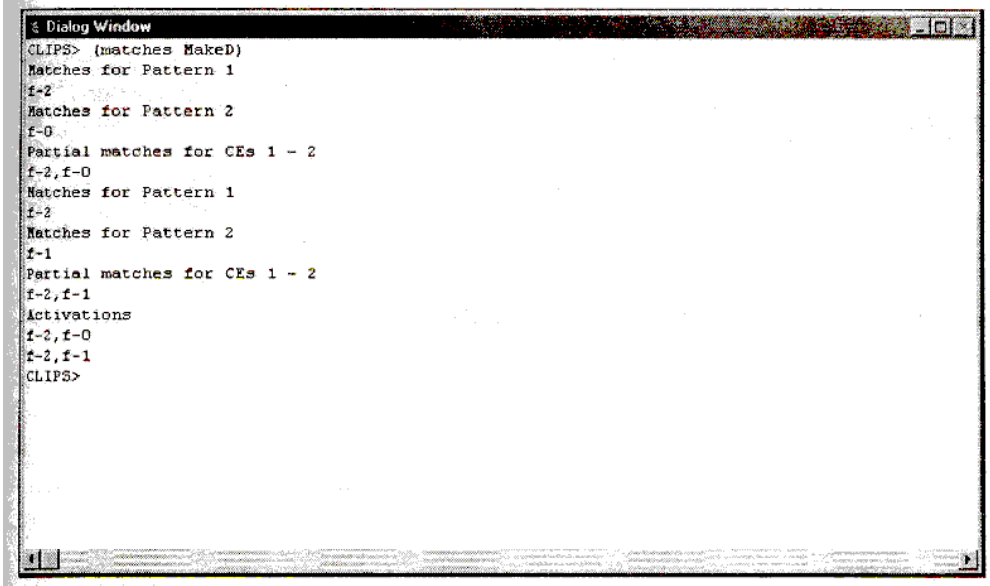

Рис. 6.16. Данные, активирующие правило `MakeD`

ГЛАВА 7. Глобальные переменные

Помимо фактов, CLIPS предоставляет еще один способ представления данных — *глобальные переменные* (globals). В отличие от переменных, связанных со своим значением в левой части правила, глобальная переменная доступна везде после своего создания (а не только в правиле, в котором она получила свое значение). Глобальные переменные CLIPS подобны глобальным переменным в процедурных языках программирования, таких как C или ADA. Однако, в отличие от переменных большинства процедурных языков программирования, глобальные переменные в CLIPS слабо типизированы. Фактически переменная может принимать значение любого примитивного типа CLIPS при каждом новом присваивании значения. Данная глава полностью посвящена способам создания глобальных переменных и приемам работы с ними.

7.1. Конструктор *defglobal и* функции для работы с глобальными переменными

С помощью конструктора defglobal в среде CLIPS могут быть объявлены глобальные переменные и присвоены их начальные значения. В дальнейшем значения созданных таким образом переменных будут доступны в любых конструкциях CLIPS. Глобальные переменные могут использоваться в процессе сопоставления образцов, но их изменения не запускает этот процесс.

Определение 7.1. Синтаксис конструктора defglobal

(defglobal [<имя-модуля>] <определение-переменной>*)

<определение-переменной> ::= <имя-переменной> = <выражение>
<имя-переменной>::= ?* <значение-типа-symbol>*

В одном конструкторе может быть объявлено произвольное количество переменных. CLIPS позволяет использовать произвольное количество конструкторов defglobal. Необязательный параметр <имя-модуля> определяет модуль, в котором должны быть определены конструируемые переменные. Если имя модуля не задано, то переменные будут помещены в текущий модуль. В случае если создаваемая переменная уже была определена, то старое определение будет заменено новым. Если при выполнении конструктора defglobal возникает ошибка, то CLIPS произведет добавление всех переменных, заданных до ошибочного определения.

Команды, использующие глобальные переменные, например, такие как ppdefglobal или undefglobal, применяют значение типа symbol, являющееся именем переменной без символов ? и * (например, max для переменной, определенной как ?*max*).

Глобальные переменные могут быть использованы в любом месте, где могут быть использованы переменные, созданные в левой части правил с некоторыми исключениями. Во-первых, глобальные переменные не могут использоваться как параметры в конструкторах deffunction, defmethod или обработчиках сообщений. Во-вторых, глобальные переменные не могут использоваться для получения новых значений в левой части правил. Например, правило из примера 7.1 недопустимо.

Пример 7.1. Неверное использование глобальной переменной

```
(defrule    example
           (fact  ?*x*)  =>)
```

А применение глобальной переменной так, как представлено в примере 7.2, вполне возможно.

Пример 7.2. Допустимое применение глобальной переменной

```
(defrule     example
           (fact  ?y  &  :(>  ?y  ?*x*))  =>)
```

Изменение глобальной переменной не приводит к запуску процесса сопоставления образцов. Например, если в базу знаний системы был добавлен факт (fact 3) и переменной ?*x* было присвоено значение 4, правило не будет активировано из-за того, что в системе отсутствует набор

данных, удовлетворяющих правило. Если после этого переменной ?*x* присвоить значение 2, несмотря на то, что текущий набор данных удовлетворяет всем условиям правила, оно все равно не будет активировано, т. к. изменение глобальной переменной не привело к запуску процесса сопоставления образцов.

Рассмотрим пример использования конструктора defglobal.

Пример 7.3. Использование конструктора defglobal

```
(defglobal
        ?*x*=3
        ?*y*=?*x*
        ?*z*=(+?*x* ?*y*)
        ?*q*=(create$ a b c)
    )
```

После выполнения данного конструктора в CLIPS появятся 4 глобальные переменные: x, y, z и q. Переменной x присваивается целое значение 3. Переменной y — значение, сохраненное в глобальной переменной x (т. е. 3). Переменной *z* — сумма значений x и y (т. е. 6). Переменной q присваивается значение, равное составному полю, содержащему 3 значения типа symbol (a, b и c), созданному с помощью функции create$. В случае если в конструкторе defglobal не было допущено синтаксических ошибок, то defglobal не возвращает никаких значений. Если ошибки имели место, то пользователь получит соответствующее сообщение. Обратите внимание, что переменная y не является указателем на переменную x, просто их значения в данный момент совпадают. Если изменить значение x, значения переменных y и z, несмотря ни на что, останутся равными 3 и 6 соответственно.

Чтобы увидеть результат работы конструктора defglobal, можно воспользоваться командой list-defglobals, для вывода на экран списка всех глобальных переменных. Добавьте еще один конструктор defglobal, объявляющий переменные вещественного и текстового типа, а также переменную со значением типа symbol.

Пример 7.4. Глобальные переменные различных типов

```
(defglobal
        ?*d*=7.8
        ?*e*=”string”
        ?*f*= symbol
    )
```

Выполните после этого команду (list-defglobals), а также команду (ppdefglobal q), которая выведет на экран определение конкретной переменной. Результат описанных действий должен соответствовать рис. 7.1.

Помимо приведенных выше команд, Windows-версия содержит два визуальных инструмента для контроля количества и состояния созданных глобальных переменных. Первый из этих инструментов — **Globals Window** (окно глобальных переменных), изображен на рис. 7.2. Для того чтобы сделать окно глобальных переменных видимым, используйте пункт **Globals Window** меню **Window.** Этот инструмент позволяет следить за изменением списка глобальных переменных, определенных в системе, например при трассировке или отладки программы.

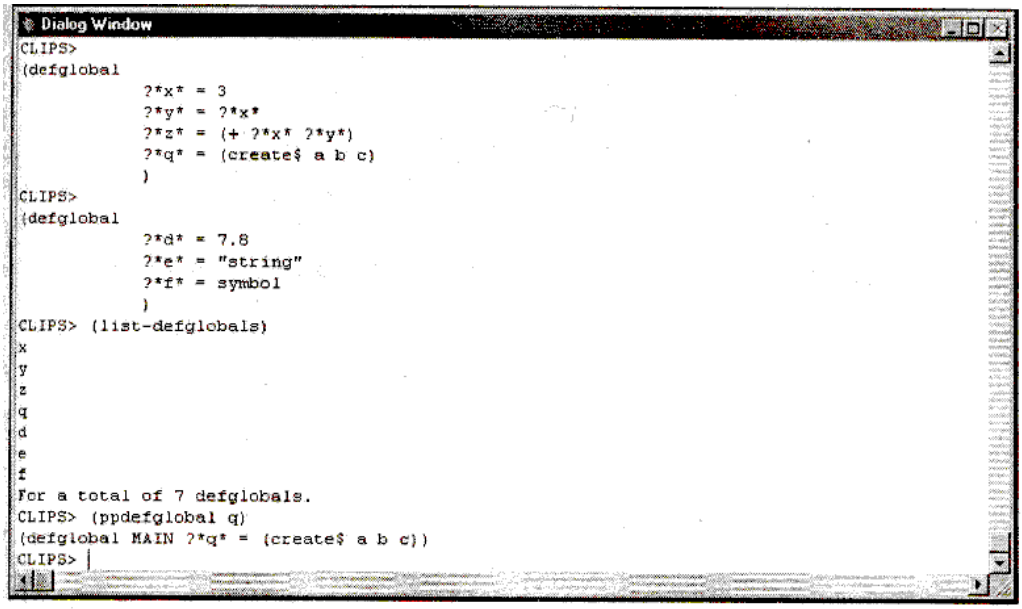

Рис. 7.1. Результат выполнения команд list-defglobals и ppdefglobal

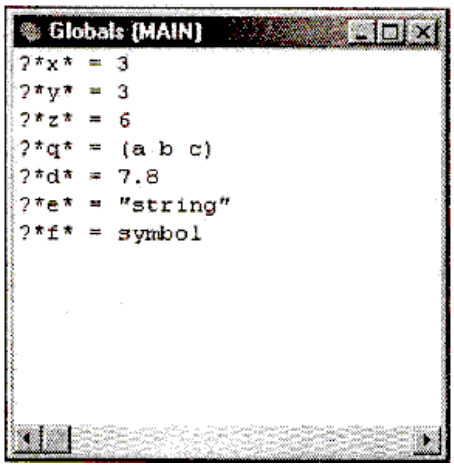

Рис. 7.2. Окно глобальных переменных

Другой инструмент, предназначенный для работы с глобальными переменными, называется **Defglobal Manager** (Менеджер глобальных переменных).Этот инструмент доступен в меню **Browse,** пункт **Defglobal Manager.** Его внешний вид представлен на рис. 7.3. Обратите внимание, что он выводит список глобальных переменных в алфавитном порядке. Общее количество переменных отображается в заголовке окна **Defglobal Manager — 7 Items.** С помощью этого инструмента можно просматривать определение глобальной переменной или удалять ее из системы. Если вы не хотите использовать менеджер глобальных переменных, то удалить созданные ранее глобальные переменные можно с помощью команды undefglobal.

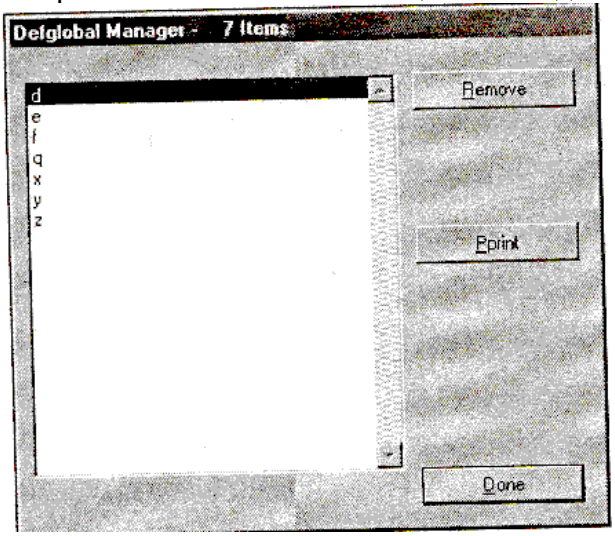

Рис. 7.3. Окно менеджера глобальных переменных

При выполнении команды reset все глобальные переменные получают начальные значения, определенные в конструкторе. Такое поведение системы можно изменить, для этого в диалоговом окне **Execution Options** сбросьте флажок **Reset Global Variables.**

Вы можете установить режим просмотра изменений значений глобальных переменных. Для этого установите флажок **Globals** в диалоговом окне **Watch Options,** как показано на рис. 7.4.

В этом случае, например при выполнении команды reset, вы увидите результат, приведенный на рис. 7.5.

Для полноценной работы с глобальными переменными необходимо рассмотреть еще одну важную функцию — bind. Эта функция позволяет устанавливать переменным новые значения:

Определение 7.2. Синтаксис функции bind

(bind <имя-переменной> <выражение>*)

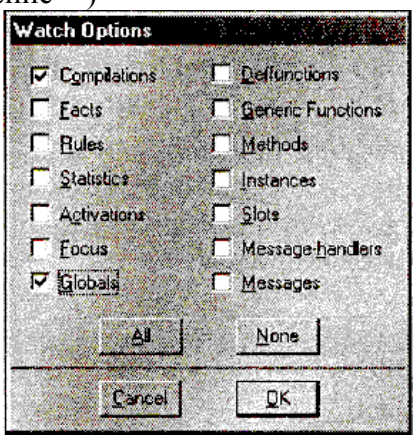

Рис. 7.4. Установка режима просмотра изменения глобальных переменных

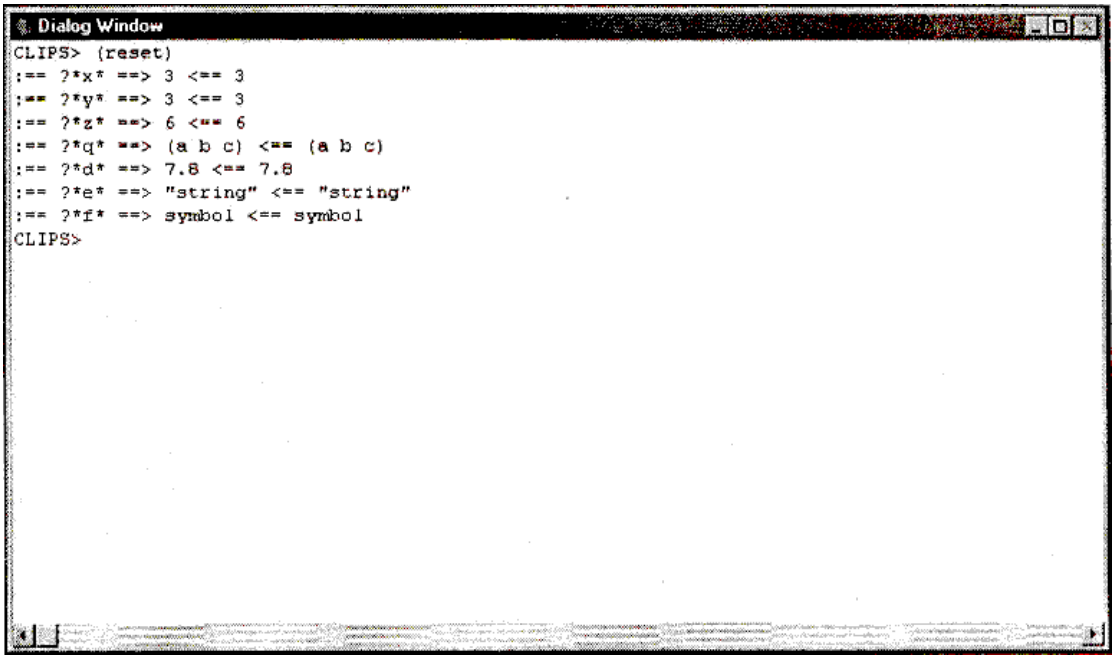

Рис. 7.5. Режим просмотра изменения глобальных переменных

Параметр выражения является необязательным. Если он не задан, то переменной будет установлено начальное значение, заданное в конструкторе defglobal. В случае если выражение было задано, то его значение будет вычислено и результат присвоен переменной. Если было задано несколько выражений, все они будут вычислены, из их результатов будет составлено составное поле, которое будет присвоено глобальной переменной.

Функция bind возвращает значение FALSE в случае, если переменной по какой-то причине не было присвоено никакого значения. В противном случае функция возвращает значение, присвоенное переменной.

Поскольку переменные в CLIPS слабо типизированы, типы значений, присваиваемые одной и той же переменной, в разные моменты времени могут не совпадать.

В качестве примера попробуйте присвоить переменной x следующие значения: (+ 5 10), (create$ a b c d) , три отдельных выражения (c), (b) и (a), а так же не присваивать переменной вообще никакого выражения. Результаты описанных действий приведены на рис. 7.6.

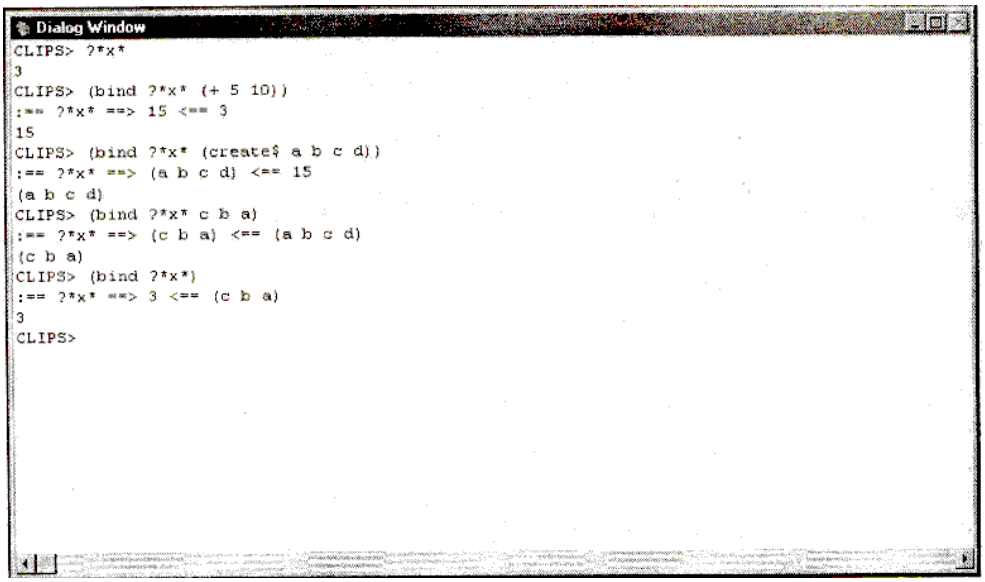

Рис. 7.6. Изменение типа глобальной переменной

Обратите внимание на то, что глобальная переменная x в нашем примере постоянно меняла тип своего значения.

ГЛАВА 8. Функции

Как уже отмечалось, CLIPS поддерживает не только эвристическую парадигму представления знаний (в виде правил), но и процедурную парадигму, используемую в большинстве языков программирования, таких, например, как Pascal или C. Функции в CLIPS являются последовательностью действий с заданным именем, возвращающей некоторое значение или выполняющей различные полезные действия (например, вывод информации на экран). Как уже упоминалось в *гл. 4,* в CLIPS существуют внутренние и внешние функции. Внутренние функции реализованы средой CLIPS, поэтому их можно использовать в любой момент. Описание внутренних функций приведено в *гл. 15.* Внешние функции — это функции, написанные пользователем. Внешние функции можно создавать как с помощью среды CLIPS, так и на любых других языках программирования, а затем подключать готовые, откомпилированные исполнимые модули к CLIPS. Однако эта тема выходит за рамки данной книги. Подробную информацию о создании внешних функций можно найти в книге *"CLIPS Reference Manual, Volume II, Advanced Programming Guide".* Для создания новых функций в CLIPS используется конструктор deffunction, описанный далее в этой главе.

8.1. Конструктор *deffunction* и способы работы с внешними функциями

Конструктор deffunction позволяет пользователю создавать новые функции непосредственно в среде CLIPS. Способ вызова функций, определенных пользователем, эквивалентен способу вызова внутренних функций CLIPS. Вызов функции осуществляется по имени, заданному пользователю. За именем функции следует список необходимых аргументов, отделенный одним или большим числом пробелов. Вызов функции вместе со списком аргументов должен заключаться в скобки. Последовательность действий определенной с помощью конструктора deffunction функции исполняется интерпретатором CLIPS (в отличие от функций, созданных на других языках программирования, которые должны иметь уже готовый исполнимый код).

Синтаксис конструктора deffunction включает в себя 5 элементов:

- имя функции;
- необязательные комментарии;
- список из нуля или более параметров;
- необязательный символ групповых параметров для указания того, что функция может иметь переменное число аргументов;
- последовательность действий или выражений, которые будут выполнены (вычислены) по порядку в момент вызова функции.

Определение 8.1. Синтаксис конструктора deffunction

```
(deffunction   <имя-функции>
               [<комментарии>]
               <обязательные-параметры>
               [<групповой-параметр>]
               <действия>)

<обязательные-параметры>   ::= <выражение-простое-поле>
<групповой-параметр>       ::= <выражение-составное-поле>
```

Функция, создаваемая с помощью конструктора deffunction, должна иметь уникальное имя, не совпадающее с именами других внешних и внутренних функций. Функция, созданная с помощью deffunction, не может быть перегружена *(см. гл. 10).* Конструктор deffunction должен быть объявлен до первого использования создаваемой им функции. Исключения составляют только рекурсивные функции.

В зависимости от того, задан ли групповой параметр, функция, созданная конструктором, может принимать точное число параметров или число параметров не меньшее, чем некоторое заданное. Обязательные параметры определяют минимальное число аргументов, которое должно быть передано функции при ее вызове. В действиях функции можно ссылаться на каждый из этих параметров как на обычные переменные, содержащие простые значения. Если был задан групповой параметр, то функция может принимать любое количество аргументов большее или равное минимальному числу. Если групповой параметр не задан, то функция может принимать

число аргументов точно равное числу обязательных параметров. Все аргументы функции, которые не соответствуют обязательным параметрам, группируются в одно значение составного поля. Ссылаться на это значение можно, используя символ группового параметра. Для работы с групповым параметром могут использоваться стандартные функции CLIPS, предназначенные для работы с составными полями *(см. гл. 15),* такие как length и nth. Определение функции может содержать только один групповой параметр.

Приведенный пример 8.1 демонстрирует описанные выше возможности работы с групповыми параметрами.

Пример 8.1. Использование группового параметра

```
(deffunction   print-args (?a ?b $?c)
               (printout t ?a " " ?b " and " (length ?c) " extras: " ?c
               crlf))
(print-args 1 2)
(print-args a b c d)
(print-args a)
```

В данном примере с помощью конструктора deffunction определяется функция print-args, которая принимает два обязательных параметра: ?a и ?b, и имеет групповой параметр $?c. Функция выводит на экран свои обязательные параметры, а также число полей в составном параметре и его содержимое. Результат выполнения данного примера приведен на рис. 8.1.

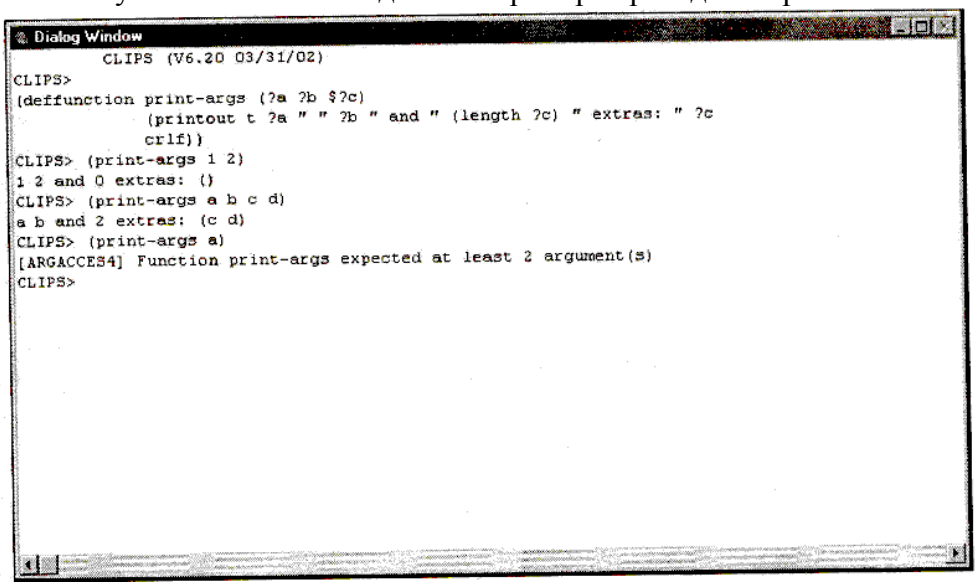

Рис. 8.1. Результат работы функции print-args

Обратите внимание, что вызов функции с числом параметров, меньшим минимального, приводит к сообщению об ошибке.

При вызове функции интерпретатор CLIPS последовательно выполняет действия в порядке, заданном конструктором. Функция возвращает значение, равное значению, которое вернуло последнее действие или вычисленное выражение. Если последнее действие не вернуло никакого результата, то выполняемая функция также не вернет результата (как в приведенном выше примере). Если функция не выполняет никаких действий, то возвращенное значение равно FALSE. В случае возникновения ошибки при выполнении очередного действия выполнение функции будет прервано и возвращенным значением также будет FALSE.

Функции могут быть само- и взаимно рекурсивными. Саморекурсивная функция просто вызывает сама себя из списка своих собственных действий. В качестве примера можно привести функцию, вычисляющую факториал.

Пример 8.2. Использование рекурсии для вычисления факториала

```
(deffunction factorial (?a)
                        (if (or (not (integerp ?a)) (< ?a 0)) then
                            (printout t "Factorial Error!" crlf)
                         else
                            (if (= ?a 0) then
                                1
                            else
                                    (  * ?a   (factorial   (-   ?a  1))))))
```

Взаимная рекурсия между двумя функциями требует предварительного объявления одной из этих функций. Для предварительного объявления функции в CLIPS используется конструктор deffunction с пустым списком действий. В следующем примере функция foo предварительно объявлена и таким образом может быть вызвана из функции bar. Окончательная реализация функции foo выполнена конструктором после объявления функции bar.

Пример 8.3. Создание взаимно рекурсивных функций

```
(deffunction foo ())
(deffunction bar ()
                (foo))
 (deffunction foo ()
                (bar) )
```

Внимательно следите за рекурсивными вызовами функций, слишком большой уровень рекурсии может привести к переполнению стека памяти. Например, приведенный выше пример с функциями bar и foo приводит к результату, представленному на рис. 8.2, и аварийному завершению CLIPS.

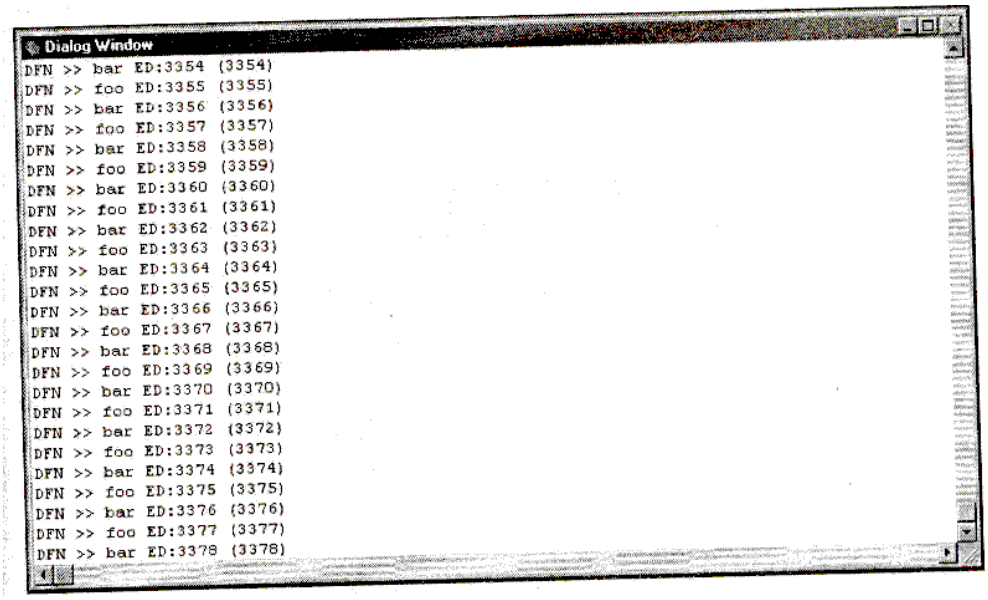

Рис. 8.2. Переполнение стека

Обратите внимание, что на рис. 8.2 в главном окне CLIPS выводится информация о запуске каждой функции. Для установки этого режима воспользуйтесь диалоговым окном **Watch Options.** Для этого откройте диалоговое окно, выбрав пункт **Watch** из меню **Execution,** и установите флажок **Deffunctions,** как показано на рис. 8.3. Этот режим также позволяет просматривать аргументы, которые использовались при каждом конкретном вызове функции.

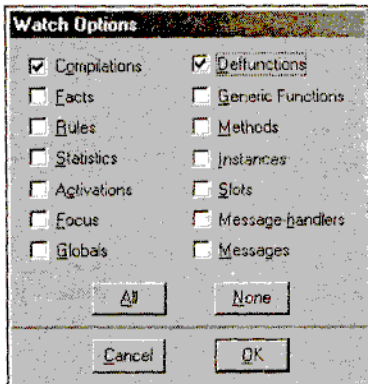

Рис. 8.3. Установка режима просмотра вызова функций

Так же как и для правил, предопределенных фактов, глобальных переменных Windows-версия CLIPS предоставляет специальный инструмент для работы с функциями — **Deffunction Manager** (Менеджер функций). Для запуска этого инструмента воспользуйтесь пунктом **Deffunction Manager** из меню **Browse. В** случае если в CLIPS не определена ни одна функция, данный пункт меню недоступен. Менеджер функций, отображающий функции, созданные нами в этой главе, изображен на рис. 8.4.

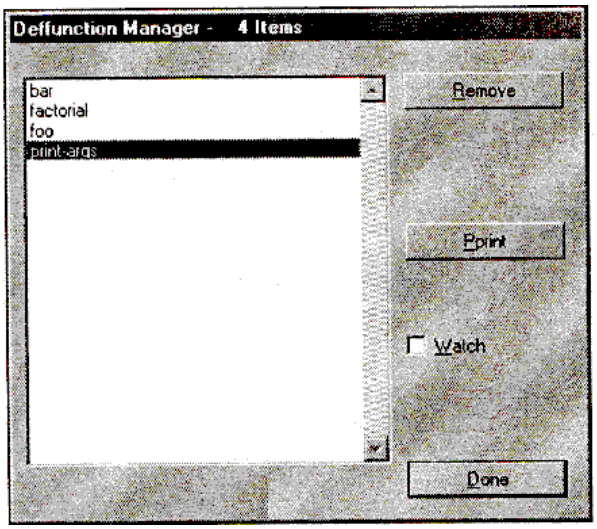

Рис. 8.4. Окно менеджера функций

Общее количество внешних функций отображается в заголовке окна менеджера — **Deffunction Manager — 4 Items.** С его помощью можно распечатать определение функции, удалить ее, а также установить режим просмотра вызова для отдельно выбранной функции.

ГЛАВА 9.Разработка экспертной системы AutoExpert

Мы рассмотрели довольно много возможностей среды CLIPS, а также способов их использования. Однако, как неоднократно говорилось ранее, среда CLIPS предназначается для создания экспертных систем, а по приведенным примерам из предыдущих глав не вполне очевидно, как именно с помощью CLIPS создавать экспертные системы. Данная глава восполнит этот пробел. В ней мы создадим полноценную, хотя и не очень сложную экспертную систему, проводящую диагностику неисправности мотора автомобиля по внешним признакам. Помимо этого наша диагностическая экспертная система должна также предоставлять пользователю соответствующие рекомендации по устранению неисправности.

Следует упомянуть, что реализация данной экспертной системы будет активно использовать управляющие команды CLIPS, такие как if-then-else и while. Однако, если вы знакомы с каким-нибудь процедурным языком программирования, вы без труда поймете приведенный ниже код. Подробно эти команды будут рассмотрены в *гл. 16.*

9.1. Исходные данные

Разработку любой экспертной системы следует начинать с выделения основных сущностей, имеющих значение при решении конкретной задачи и законов, скорее всего эмпирических, действующих над этими сущностями. В подавляющем большинстве случаев эту информацию получают при помощи эксперта, человека хорошо знающего и давно работающего в этой области. Методы получения информации от эксперта и ее обработка выходят за рамки настоящей книги, но эта тема не плохо освещена в других работах. Для решения нашей конкретной задачи предположим, что в результате бесед с экспертом в области установления неисправностей и ремонта автомобилей были установлены следующие эмпирические правила:

1.Двигатель обычно находится в одном из 3-х состояний: он может работать нормально, работать неудовлетворительно или не заводиться.

2.Если двигатель работает нормально, то это означает, что он нормально вращается, система зажигания и аккумулятор находятся в норме и никакого ремонта не требуется.

3.Если двигатель запускается, но работает ненормально, то это говорит, по крайней мере, о том, что аккумулятор в порядке.

4.Если двигатель не запускается, то нужно узнать, пытается ли он вращаться. Если двигатель вращается, но при этом не заводится, то это может говорить о наличии плохой искры в системе зажигания. Если двигатель даже не пытается заводиться, то это говорит о том, что искры нет в принципе.

5.Если двигатель не заводится, но вращается, нужно проверить наличие топлива. Если топлива нет — то, скорей всего, для ремонта машины нужно просто заправиться.

6.Если двигатель не заводится, нужно также проверить, заряжен ли аккумулятор, если нет, то его следует зарядить.

7.Если двигатель не заводится, и существует вероятность плохой искры в системе зажигания, то необходимо проверить контакты. Контакты могут быть в одном из трех состояний — чистые, опаленные и грязные, в случае опаленных контактов их необходимо заменить, в случае если контакты грязные, их достаточно просто почистить.

8.Если двигатель не заводится, искры нет и аккумулятор заряжен, то нужно проверить катушку зажигания на электрическую проводимость. В случае если ток не проходит через катушку, то ее необходимо заменить. Если катушка зажигания в порядке, значит необходимо заменить распределительные провода.

9.Если двигатель запускается, но при этом ведет себя инертно, не сразу реагирует на подачу топлива, то необходимо прочистить топливную систему.

10.Если двигатель запускается, но происходят перебои с зажиганием, то это говорит о наличии плохой искры в системе зажигания, для устранения данной неисправности необходимо отрегулировать зазоры между контактами.

11.Если двигатель запускается и стучит, то необходимо отрегулировать зажигание.

12.Если двигатель запускается, но не развивает нормальной мощности, то это может говорить об опаленных или загрязненных контактах (см. правило 7).

13.Возможны ситуации, когда состояние двигателя нельзя описать приведенными выше факторами и машине может потребоваться более детальный анализ состояния.

Имея эти данные, приступим к решению поставленной задачи.

9.2. Сущности

Из приведенных выше правил можно выделить следующие сущности, имеющие значение при решении задачи.

- Во-первых, для решения задачи экспертной системе необходимо знать, в каком состоянии находится машина, диагностика которой производится. Эксперт выделил три возможных состояния: нормальная работа двигателя, двигатель работает неудовлетворительно, не заводится (см. правило 1).
- Во-вторых, большинство приведенных правил помимо состояния двигателя в целом используют понятие состояния вращения двигателя. Согласно этим правилам двигатель может находиться в одном из двух состояний, которые определяются в зависимости от того, способен он вращаться (работать) или нет.
- В-третьих, в некоторых правилах (см. правила 4, 7, 8, 10) используется понятие состояния системы зажигания. Система зажигания может быть в одном из трех состояний: нормальное состояние, не регулярная работа и нерабочее состояние.
- В-четвертых, в правилах 6 и 8 используется понятие — состояние аккумулятора. Аккумулятор может быть в одном из двух состояний: заряженным и разряженным.

Для того чтобы решения данной задачи было более наглядным, мы не будем использовать шаблоны. Для представления в CLIPS всех перечисленных выше данных воспользуемся упорядоченными фактами CLIPS. Исходя из приведенного выше списка, нам могут понадобиться факты, приведенные в примере 9.1.

Пример 9.1. Факты, описывающие состояние автомобиля и его узлов

```
;      Группа фактов, описывающая состояние машины
working-state engine normal                ;нормальная работа
working-state engine unsatisfactory        ;неудовлетворительная работа
working-state engine does-not-start        ;не заводится
;      Группа фактов, описывающая состояние двигателя
rotation-state engine rotates              ;двигатель вращается
rotation-state engine does-not-rotate      ;двигатель не вращается
;      Группа фактов, описывающая состояние системы зажигания
spark-state engine normal                  ;зажигание в порядке
spark-state engine irregular-spark         ;искра нерегулярна
spark-state engine does-not-spark          ;искры нет
;      Группа фактов, описывающая состояние системы питания
charge-state battery charged               ;аккумулятор заряжен
charge-state battery dead                  ;аккумулятор разряжен
```

Обратите внимание, что факты, входящие в одну группу (содержат одинаковое первое поле), являются взаимоисключающими, т.е. наличие в системе сразу двух фактов из одной группы лишено смысла.

Их постановки задачи следует, что наша экспертная система должна предоставлять пользователю рекомендации, позволяющие устранить найденную неисправность. Из приведенных выше правил можно выделить следующие рекомендации: добавить топливо (правило 5); зарядить аккумулятор (правило 6); заменить или почистить контакты (правило 7 или правило 12); заменить катушку зажигания или распределительные провода (правило 8); прочистить топливную систему (правило 9); отрегулировать зазоры между контактами (правило 10); отрегулировать зажигание (правило 11). Необходимо помнить также о двух крайних случаях: ремонт не требуется в принципе; экспертная система не смогла поставить диагноз. Для представления всех этих рекомендаций будем использовать факты, представленные в примере 9.2.

Пример 9.2. Факты, описывающие рекомендации по ремонту автомобиля

```
repair   “Add gas.”
repair   “Charge the battery.”
repair   “Replace the points.”
repair   “Clean the points.”
```

repair "Replace the ignition coil."
repair "Repair the distributor lead wire."
repair "Clean the fuel line."
repair "Point gap adjustment."
repair "No repair needed."
repair "Take your car to a mechanic."

Все приведенные факты, использующиеся для предоставления пользователю рекомендаций по ремонту, во втором поле содержат текстовое значение с рекомендацией по ремонту.
Обратите внимание, что одни и те же рекомендации могут выводиться как правилом 7, так и правилом 12. Однако состояние машины при этой поломке отличается. Для того чтобы иметь возможность обрабатывать эту ситуацию с помощью одного правила CLIPS, введем еще два дополнительных факта.

Пример 9.3. Факты, описывающие мощность работы двигателя

```
symptom engine low-output          ;низкая мощность
symptom engine not-low-output      ;нормальная мощность
```

Кроме описанных выше фактов системе могут понадобиться факты, описывающие проявления неисправности. Однако в нашей версии экспертной системы таких фактов не будет. О том, как обойтись без них, вы узнаете в следующем разделе. Приведенный выше список фактов вполне достаточен для решения поставленной задачи. Приступим к следующему этапу — сбору исходной информации для диагностики.

9.3. Сбор информации

Как упоминалось выше, для работы нашей системы можно заставить пользователя вручную вводить факты, описывающие проявление возникшей неисправности. Однако такой метод имеет ряд серьезных недостатков: пользователь может забыть о каких-нибудь существенных деталях или, наоборот, указать слишком много информации, что может помешать нормальной работе системы. Кроме того, факты, описывающие проявление неисправности, должны были бы иметь строго определенный формат, и система не смогла бы их обработать в случае ошибки со стороны пользователя.

В нашей экспертной системе мы реализуем правила диагностики, которые в зависимости от той или иной ситуации будут задавать пользователю необходимые вопросы и получать ответ в строго заданной форме. Дальнейшая диагностика будет производиться с учетом предыдущих ответов на вопросы, заданные пользователю. Эти ответы будут формировать описание текущей ситуации с помощью фактов, приведенных выше.

Для реализации подобной архитектуры будет необходимо реализовать функцию, задающую пользователю произвольный вопрос и получающую ответ из заданного набора корректных ответов. В примере 9.4 приведена одна из возможных реализаций такой функции.

Пример 9.4. Функция ask-question

```
(deffunction ask-question (?question $?allowed-values)
             (printout t ?question)
             (bind ?answer (read))
             (if (lexemep ?answer)
                 then
                     (bind ?answer (lowcase ?answer)))
                      (while (not (member ?answer ?allowed-values)} do
                                  (printout t ?question)
                                  (bind ?answer (read))
                                  (if (lexemep ?answer)
                                              then
                                                      (bind ?answer (lowcase ?answer))))
              ?answer
              )
```

Функция принимает два аргумента: простую переменную question, которая содержит текст вопроса, и составную переменную allowed-values с набором допустимых ответов. Сразу после своего вызова функция выводит на экран соответствующий вопрос и читает ответ пользователя в переменную answer. Если переменная answer содержит текст, то она будет принудительно приведена к прописному алфавиту. После этого функция проверяет, является ли полученный ответ одним из заданных корректных ответов. Если нет, то процесс повторится до получения корректного ответа, иначе функция вернет ответ, введенный пользователем.

Будет также очень полезно определить функцию, задающую пользователю вопрос и допускающий ответ в виде да/нет, т. к. это один из самых распространенных типов вопросов. С учетом реализации функции ask-question эта функция примет вид, представленный в примере 9.5.

Пример 9.5. Функция yes-or-no-p

```
(deffunction yes-or-no-p   (?question)
            (bind  ?response   (ask-question  ?guestion yes  no y n))
            (if   (or   (eq ?response yes)    (eq ?response  y))
                  then
                     TRUE
                  else
                     FALSE)
        )
```

Функция yes-or-no-p вызывает функцию ask-question с постоянным набором допустимых ответов: yes, no, y и n. В случае если пользователь ввел ответ yes или y, функция возвращает значение TRUE, иначе — FALSE. Обратите внимание, что поскольку функция yes-or-no-p использует функцию ask-question, то она должна быть определена после нее.

9.4. Диагностические правила

Для упрощения реализации нашей экспертной системы введем следующее ограничение: за один запуск система может предоставить пользователю только одну рекомендацию по исправлению неисправности. В случае если в машине несколько неисправностей, то систему нужно будет последовательно вызывать несколько раз, удаляя обнаруженную на каждом новом шаге неисправность.
Таким образом, одним из образцов всех диагностических правил будет (not (repair ?)), гарантирующий, что диагноз еще не поставлен.
Первым реализуем правило, определяющее общее состояние двигателя (см. правило 1).

Пример 9.6. Правило determine-engine-state

```
(defrule determine-engine-state ""
        (not (working-state engine ?) )
        (not (repair ?) )
         =>
        (if (yes-or-no-p "Does the engine start (yes/no)? ")
                then
                        (if (yes-or-no-p "Does the engine run normally (yes/no)? ")
                            then
                                (assert (working-state engine normal) )
                            else
                                (assert (working-state engine unsatisfactory) ) )
                else
                        (assert (working-state engine does-not-start) ) )
        )
```

Условный элемент (not (working-state engine ?)) гарантирует, что общее состояние двигателя еще не определено. Если это так, то пользователю задаются соответствующие вопросы и в систему добавляется факт, описывающий текущее общее состояние двигателя.

Теперь реализуем правило, определяющее, пытается ли двигатель вращаться, в случае если он не заводится.

Пример 9.7. Правило determine-rotation-state

```
(defrule determine-rotation-state ""
        (working-state engine does-not-start)
      (not (rotation-state engine ?) )
      (not (repair ?) )
      =>
      (if (yes-or-no-p "Does the engine rotate (yes/no) ? ")
            then
                  (assert (rotation-state engine rotates) )
                  (assert (spark-state engine irregular-spark) )
            else
                  (assert (rotation-state engine does-not-rotate) )
                  (assert (spark-state engine does-not-spark) ) )
```

Это правило выполняется, в случае если общее состояние двигателя определено и известно, что он не заводится. Кроме того, условный элемент (not (rotation-state engine ?)) гарантирует, что это правило еще не вызывалось. В зависимости от того или иного ответа пользователя правило добавляет соответствующий набор фактов (см. правило 4).

Далее реализуем довольно простые правила 5 и 6. Выполняемые ими действия вы поймете без дополнительных комментариев.

Пример 9.8. Правила determine-gas-level и determine-battery-state

```
(defrule determine-gas-level ""
          (working-state engine does-not-start)
          (rotation-state engine rotates)
          (not (repair ?))
          =>
          (if (not (yes-or-no-p "Does the tank have any gas in it (yes/no)? "))
                then
                     (assert (repair "Add gas.")))
        )
(defrule determine-battery-state ""
          (rotation-state engine does-not-rotate)
          (not (charge-state battery ?))
          (not (repair ?))
          =>
          (if (yes-or-no-p "Is the battery charged (yes/no)? ")
                then
                      (assert (charge-state battery charged))
                else
                      (assert, (repair "Charge the battery."))
                      (assert (charge-state battery dead)))
      )
```

Обратите внимание, что правило determine-battery-state, помимо определения возможной неисправности, также применяется для добавления в систему факта, описывающего текущее состояние аккумулятора, который может быть использован другими правилами.

При реализации правила 7 необходимо обратить внимание на то, что рекомендации, предоставляемые этим правилом, подходят для двух в корне отличающихся ситуаций. Во-первых, в случае если двигатель не заводится, и существует вероятность плохой искры в системе зажигания (правило 7). Во-вторых, в случае если двигатель запускается, но не развивает нормальной мощности

(правило 12). Поэтому выполним реализацию этих правил так, как представлено в примере 9.9.

Пример 9.9. Правила determine-low-output и determine-point-surface-state

```
(defrule  determine-low-output   ""
          (working-state engine unsatisfactory)
          (not   (symptom engine low-output   |   not-low-output))
          (not   (repair ?))
          =>
          (if (yes-or-no-p "Is the output of the engine low (yes/no)? ")
              then
                    (assert (symptom engine low-output))
              else
                    (assert (symptom engine not-low-output)))
         )
(defrule determine-point-surface-state ""
           (or   (and  (working-state engine does-not-start)
                       (spark-state engine irregular-spark))
                 (symptom engine low-output))
           (not (repair ?))
           =>
           (bind ?response (ask-question "What is the surface state of the
                                  points (normal /burned /contaminated)?"
                                  normal burned contaminated))
           (if (eq ?response burned)
               then
                    (assert (repair "Replace the points."))
               else
                    (if (eq ?response contaminated)
                        then
                             (assert (repair "Clean the points."))))
      )
```

Правило determine-low-output определяет, имеет ли место низкая мощность двигателя или нет. Правило determine-point-surface-state адекватно реагирует на условия, заданные в правилах *7* и 12. Обратите внимание на использование условных элементов or и and, которые обеспечивают одинаковое поведение правила в двух абсолютно разных ситуациях. Кроме того, правило determine-point-surface-state отличается от приведенных ранее тем, что непосредственно использует функцию ask-question, вместо yes-or-no-p, т. к. в данный момент пользователю задается вопрос, подразумевающий три варианта ответа.

Реализация оставшихся диагностических правил (8—11) также не должна вызвать у вас затруднений.

Пример 9.10. Оставшиеся диагностические правила

```
(defrule determine-conductivity-test ""
          (working-state engine does-not-start)
          (spark-state engine does-not-spark)
          (charge-state battery charged)
          (not (repair ?))
          =>
          (if (yes-or-no-p "Is the conductivity test for the ignition coil positive(yes/no)? ")
              then
                    (assert (repair "Repair the distributor lead wire."))
              else
                    (assert (repair "Replace the ignition coil.")))
        )
(defrule determine-sluggishness ""
          (working-state engine unsatisfactory)
          (not (repair ?))
          =>
```

```
        (if (yes-or-no-p "Is the engine sluggish (yes/no)? ") then
           (assert (repair "Clean the fuel line."))) ) (defrule determine-misfiring ""
        (working-state engine unsatisfactory) (not (repair ?))

        (if (yes-or-no-p "Does the engine misfire (yes/no)? ")

            then
                (assert (repair "Point gap adjustment."))
                (assert (spark-state engine irregular-spark)))
    )

(defrule determine-knocking  ""
            (working-state engine unsatisfactory)
            (not   (repair ?))
            =>
            (if (yes-or-no-p "Does the engine knock (yes/no)? ")
                    then
                            (assert (repair "Timing adjustment.")))
        )
```

9.5. Последние штрихи

Внимательно взглянув на список правил, мы увидим, что некоторые правила (2, 3 и 13) остались до сих пор не реализованными.

В качестве реализации правила 13 мы будем использовать правило no-repairs, приведенное в примере 9.11.

Пример 9.11. Правило no-repairs

```
    (defrule no-repairs  ""
            (declare   (salience -10) )
            (not   (repair ?) )
            = >
            (assert   (repair   "Take  your  car  to a mechanic."))
        )
```

Обратите внимание на использование приоритета при определении этого правила. Все правила, приведенные в предыдущем разделе, определялись с приоритетом, по умолчанию равным нулю. Использование для правила no-repairs приоритета, равного —10, гарантирует, что правило не будет выполнено, пока в плане решения задачи находится, по крайней мере, одно из диагностических правил. Если все активированные диагностические правила отработали и ни одно из них не смогло подобрать подходящую рекомендацию по устранению неисправности, то CLIPS запустит правило no-repairs, которое просто порекомендует пользователю обратиться к более опытному механику.

Реализация правил 2 и 3 приведена ниже.

Пример 9.12. Правила normal-engine-state-conclusions и unsatisfactory-engine-state-conclusions

```
(defrule normal-engine-state-conclusions  ""
        (declare   (salience 10) )
        (working-state engine normal)
        =>
        (assert (repair "No repair needed."))
        (assert (spark-state engine normal))
        (assert (charge-state battery charged))
        (assert (rotation-state engine rotates))
    )
```

```
(defrule unsatisfactory-engine-state-conclusions ""
          (declare (salience 10))
          (working-state engine unsatisfactory)
          =>
          (assert (charge-state battery charged))
          (assert (rotation-state engine rotates))
       )
```

В этих правилах, наоборот, используется более высокий приоритет, что гарантирует их выполнение до выполнения любого диагностического правила (естественно, только в случае удовлетворения условий, заданных в левой части правил). Это избавит нашу систему от лишних проверок, а пользователя от лишних вопросов.

Наша экспертная система фактически готова к работе. Единственное, чего ей не хватает, — это метода вывода итоговой информации и правила, сообщающего пользователю о начале работы. Ниже приведена реализация этих правил.

Пример 9.13. Правила system-banner и print-repair

```
(defrule system-banner ""
          (declare (salience 10))
          =>
          (printout t crlf crlf)
          (printout t "*********************************" crlf)
          (printout t "* The Engine Diagnosis Expert System   *" crlf)
          (printout t "*********************************" crlf)
          (printout t crlf crlf)
      )
(defrule print-repair ""
          (declare (salience 10))
          (repair ?item)
          =>
          (printout t crlf crlf)
          (printout t "Suggested Repair:")
          (printout  t  crlf  crlf)
          (format  t   "   %s%n%n%n"   ?item)
      )
```

9.6. Листинг программы

В данном разделе приведен полный листинг программы с подробными комментариями. Если у вас еще не сложилась целостная картина о том, как работает наша экспертная система, из каких частей она состоит, внимательно изучите приведенный ниже код.

Пример 9.14. Полный листинг программы

```
;=============================================================
;        Пример экспертной системы на языке CLIPS
;
;        Приведенная ниже экспертная система способна
;        диагностировать некоторые неисправности автомобиля и
;        предоставлять пользователю рекомендации по устранению
;        неисправности.
;
;=============================================================
```

```
;=========================================================
;       Вспомогательные функции
;=========================================================
;---------------------------------------------------------
;       Функция ask-question задает пользователю вопрос, полученный
;        в переменной ?question, и получает от пользователя ответ,
;       принадлежащий списку допустимых ответов, заданному в $?allowed-values
(deffunction ask-question (?question $?allowed-values)
                (printout t ?question)
                (bind ?answer (read))
                (if (lexemep ?answer)
                        then
                                (bind ?answer (lowcase ?answer)))
                                (while (not (member ?answer ?allowed-values)) do
                                        (printout t ?question)
                                        (bind ?answer (read))
                                        (if (lexemep ?answer)
                                            then
                                                (bind ?answer (lowcase ?answer))))
                 ?answer
               )
;---------------------------------------------------------------------
;  Функция yes-or-no-p задает пользователю вопрос, полученный
;  в переменной ?question, и получает от пользователя ответ yes(y) или
;  по(n). В случае положительного ответа функция возвращает значение TRUE,
;   иначе — FALSE
 (deffunction yes-or-no-p (?question)
                (bind ?response (ask-question ?question yes no y n))
                (if (or (eq ?response yes) (eq ?response y))
                      then
                         TRUE
                      else
                         FALSE)
            )
;=========================================================
;       Диагностические правила
;=========================================================
;---------------------------------------------------------
;      Правило determine-engine-state определяет текущее состояние двигателя
;      машины по ответам, получаемым от пользователя. Двигатель может
;      находиться в одном из трех состояний: работать нормально
;      (working-state engine normal) , работать неудовлетворительно
;       (working-state engine unsatisfactory) и не заводиться
;     (working-state engine ;does-not-start) (см. правило 1).
(defrule determine-engine-state ""
            (not (working-state engine ?) )
            (not (repair ?) )
            =>
            (if (yes-or-no-p "Does the engine start (yes/no) ? ")
                then
                    (if (yes-or-no-p "Does the engine run normally (yes/no)? ")
                        then
                            (assert (working-state engine normal) )
                        else
```

```
                    (assert (working-state engine unsatisfactory) ) )
          else
               (assert (working-state engine does-not-start) ) )
   )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;     Правило determine-rotation-state определяет состояние вращения двигателя по ответу,
;     получаемому от пользователя. Двигатель может вращаться (rotation-state engine rotates)
;      или не вращаться (spark-state engine does-not-spark) (см. правило 4 ) .
;      Кроме того, правило делает предположение о наличии плохой искры
;      или ее отсутствии в системе зажигания
(defrule determine-rotation-state ""
        (working-state engine does-not-start)
        (not (rotation-state engine ?))
        (not (repair ?))
        =>
        (if (yes-or-no-p "Does the engine rotate (yes/no)? ")
             then
                ; Двигатель вращается
                  (assert (rotation-state engine rotates))
                ; Плохая искра
                  (assert (spark-state engine irregular-spark))
             else
                ; Двигатель не вращается
                (assert (rotation-state engine does-not-rotate))
                ; Нет искры
                (assert (spark-state engine does-not-spark)))
        )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;      Правило determine-gas-level по ответу пользователя определяет
;      наличие топлива в баке. В случае если топлива нет, пользователю
;      выдается рекомендация по ремонту — машину необходимо заправить
;      (repair "Add gas.") (см. правило 5) . При появлении соответствующей
;      рекомендации выполнение диагностических правил прекращается.
(defrule determine-gas-level ""
        (working-state engine does-not-start)
        (rotation-state engine rotates)
        (not (repair ?) )
         =>
         (if (not (yes-or-no-p "Does the tank have any gas in it (yes/no) ? "))
             then
                  ; Машину необходимо заправить
                  (assert (repair "Add g a s . " ) ) )
       )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;      Правило determine-battery-state по ответу пользователя определяет,
;      заряжен ли аккумулятор. В случае если это не так, пользователю
;      выдается рекомендация по ремонту — Зарядите аккумулятор (repair
;      "Charge the battery.") (см. правило 6) .
;     Кроме того, правило добавляет факт, описывающий состояние аккумулятора.
;     Выполнение диагностических правил прекращается
(defrule determine-battery-state ""
       (rotation-state engine does-not-rotate}
       ; Состояние аккумулятора еще не определено
       (not (charge-state battery ? ) )
       (not (repair ?))
```

```
        =>
        (if (yes-or-no-p "Is the battery charged (yes/no)? ")
            then
                ; Аккумулятор заряжен
                (assert (charge-state battery charged))
            else
                ; Зарядите аккумулятор
                (assert (repair "Charge the battery."))
                ; Аккумулятор разряжен
                (assert (charge-state battery dead)))
    )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;     Правило determine-low-output определяет, развивает ли двигатель
;     нормальную выходную мощность или нет и добавляет в систему факт,
;     описывающий эту характеристику (см. правило 12).
(defrule determine-low-output ""
        (working-state engine unsatisfactory)
        ; Мощность работы двигателя еще не определена
        (not (symptom engine low-output | not-low-output))
        (not (repair ?))
        =>
        (if (yes-or-no-p "Is the output of the engine low (yes/no)? ")
            then
              ; Низкая выходная мощность двигателя
                (assert (symptom engine low-output))
            else
              ; Нормальная выходная мощность двигателя
                (assert (symptom engine not-low-output)))
    )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;     Правило determine-point-surface-state определяет по ответу
;     пользователя состояние контактов (см. правила 7, 12). Контакты могут
;     находиться в одном из трех состояний: чистые, опаленные и
;     загрязненные. В двух последних случаях пользователю выдаются
;     соответствующие рекомендации.
;     Выполнение диагностических правил прекращается.
(defrule determine-point-surface-state ""
        (or (and (working-state engine does-not-start);не заводится
                 (spark-state engine irregular-spark));и плохая искра
            (symptom engine low-output));или низкая мощность
        (not (repair ?))
        =>
        (bind ?response (ask-question "What is the surface state of the
                                 points (normal /burned /contaminated)?"
                                 normal burned contaminated))
        (if (eq ?response burned)
            then
              ; Контакты опалены — замените контакты
                (assert (repair "Replace the points."))
            else
                (if (eq ?response contaminated)
                    then
                        ; Контакты загрязнены - почистите их
                        (assert (repair "Clean the points."))))
    )
```

```
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;	Правило determine-conductivity-test по ответу пользователя определяет,
;	пропускает ли ток катушка зажигания. Если нет, то ее следует заменить.
;	Если пропускает, то причина неисправности — распределительные провода.
;	Для нормальной работы правила необходимо убедиться, что аккумулятор
;	заряжен и искры нет (см. правило 8)
;	Выполнение диагностических правил прекращается.
(defrule determine-conductivity-test ""
	(working-state engine does-not-start)
	(spark-state engine does-not-spark)  ;нет искры
	(charge-state battery charged)          ;аккумулятор заряжен
	(not (repair ?) )
	=>
	(if (yes-or-no-p "Is the conductivity test for the ignition coil positive (yes/no) ? ")
	    then
	          ; Замените распределительные провода
	            (assert (repair "Repair the distributor lead wire."))
	    else
	          ; Замените катушку зажигания
	            (assert (repair "Replace the ignition coil.")))
	)
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;	Правило determine-sluggishness спрашивает пользователя, не ведет ли
;	себя машина инертно (не сразу реагирует на подачу топлива) .
;	Если такой факт обнаружен, то необходимо прочистить
;	топливную систему (см. правило 9) и выполнение диагностических правил
;	прекращается.
(defrule determine-sluggishness ""
	(working-state engine unsatisfactory)
	(not (repair ?))
	=>
	(if (yes-or-no-p "Is the engine sluggish (yes/no)? ")
	        then
	            ; Прочистите систему подачи топлива
	              (assert (repair "Clean the fuel line.")))
	)
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;	Правило determine-misfiring узнает — нет ли перебоев с зажиганием.
;	Если это так, то необходимо отрегулировать зазоры между контактами
;	(см. правило 10).
;	Выполнение диагностических правил прекращается.
(defrule determine-misfiring ""
	(working-state engine unsatisfactory)
	(not (repair ?))
	=>
	(if (yes-or-no-p "Does the engine misfire (yes/no)? ")
	      then
	          ; Отрегулируйте зазоры между контактами
	          (assert (repair "Point gap adjustment."))
	          ; Плохая искра
	          (assert (spark-state engine irregular-spark)))
	)
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
;      Правило determine-knocking узнает — не стучит ли двигатель.
;      Если это так, то необходимо отрегулировать зажигание (см. правило 11).
;      Выполнение диагностических правил прекращается.
 (defrule determine-knocking ""
            (working-state engine unsatisfactory)
            (not (repair ?))
            =>
             (if (yes-or-no-p "Does the engine knock (yes/no)? ")
                  then
                     ; Отрегулируйте положение зажигания
                     (assert (repair "Timing adjustment.")))
        )
;=============================================================
;      Правила, определяющие состояние некоторых подсистем автомобиля
;      по характерным состояниям двигателя
;=============================================================
;-------------------------------------------------------------
;      Правило normal-engine-state-conclusions реализует правило 2
(defrule normal-engine-state-conclusions ""
           (declare (salience 10))
          ; Если двигатель работает неудовлетворительно
           (working-state engine normal)
            =>
          ; то
           (assert (repair "No repair needed."))    ; ремонт не нужен
           (assert (spark-state engine normal))     ; зажигание в норме
           (assert (charge-state battery charged))  ; аккумулятор заряжен
           (assert (rotation-state engine rotates))   ; двигатель вращается
        )
;      Правило unsatisfactory-engine-state-conclusions реализует правило 3
(defrule unsatisfactory-engine-state-conclusions ""
            (declare (salience 10))
            ; Если двигатель работает нормально
            (working-state engine unsatisfactory)
            =>
          ; то
            (assert (charge-state battery charged))  ; аккумулятор заряжен
            (assert (rotation-state engine rotates))   ; двигатель вращается
        )
;=============================================================
=    ;      Запуск и завершение
;=============================================================
;-------------------------------------------------------------
;      Правило no-repairs запускается в случае, если ни одно
;      из диагностических правил не способно определить неисправность.
;      Правило корректно прерывает выполнение экспертной системы и предлагает
;      пройти более тщательную проверку (см. правило 13).
(defrule no-repairs ""
            (declare (salience -10))
            (not (repair ?))
            =>
             (assert (repair "Take your car to a mechanic."))
        )
;-------------------------------------------------------------
;      Правило print-repair выводит на экран диагностическое сообщение
```

```
;       по устранению найденной неисправности,
(defrule print-repair ""
            (declare (salience 10))
            (repair ?item)
           =>
            (printout   t   crlf crlf)
            (printout   t   "Suggested Repair:")
            (printout   t   crlf crlf)
            (format t   "   %s%n%n%n" ?item)
           )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;       Правило system-banner выводит на экран название экспертной системы
;       при каждом новом запуске.
 (defrule system-banner " каждом новом запуске."
          (declare (salience 10) )
           =>; каждом новом запуске.
          (printout t crlf crlf)
          (printout t "*****************************************" crlf)
          (printout t "* The Engine Diagnosis Expert System  *" crlf)
          (printout t "*********************************" crlf)
          (printout t crlf crlf)
       )
```

Помните, что среда CLIPS (по крайней мере, последняя версия 6.20) воспринимает только символы английского алфавита. Все комментарии в приведенном листинге даны на русском языке для наглядности, однако при вводе программы в таком виде CLIPS выведет сообщение об ошибке.

9.7. Запуск программы

Для запуска программы наберите приведенный в примере 9.14 листинг в каком-нибудь текстовом редакторе (лучше использовать встроенный редактор CLIPS по причинам, упоминавшимся в *гл. 6).* Сохраните набранный файл, например, с именем auto.CLP.

После этого запустите CLIPS или, если он уже был у вас запущен, очистите его командой (clear). Загрузите созданный вами файл с помощью команды (load "auto.CLP"). Если файл был набран без ошибок, то вы должны увидеть сообщения, представленные на рис. 9.1.

Рис. 9.1 демонстрирует успешную попытку загрузки файла конструкторов. Обратите внимание, что функция load вернула значение TRUE. Если это нетак, значит, в синтаксисе определений функций или правил была допущена ошибка. Для загрузки вы также могли бы воспользоваться функцией load*. В этом случае на экран не выводилась бы информация, отражающая процесс загрузки.

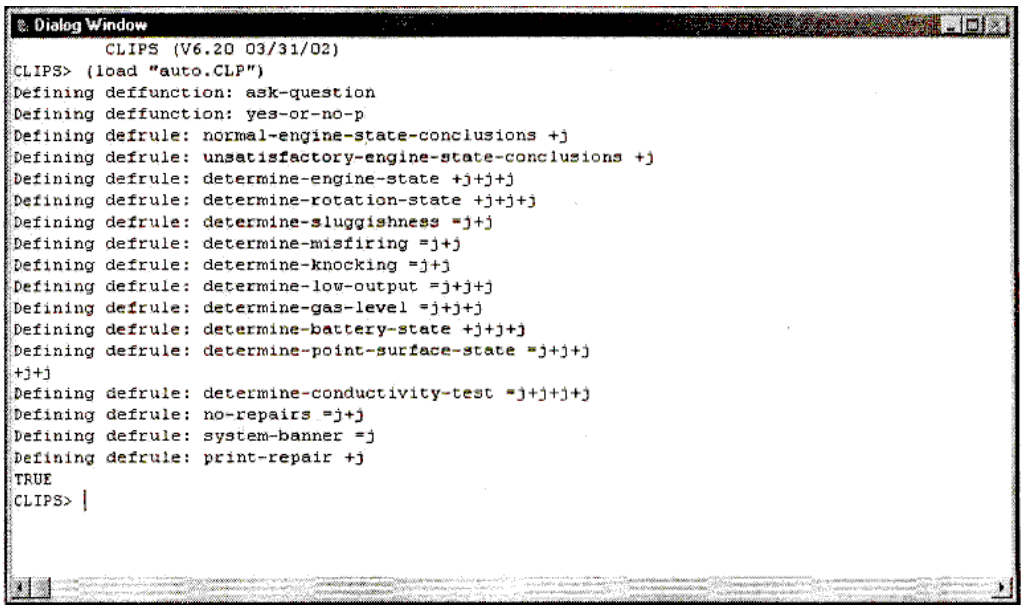

Рис. 9.1. Загрузка экспертной системы

После удачной загрузки файла убедитесь, что все правила присутствуют в списке правил CLIPS, а функции — в списке функций. Легче всего это выполнить с помощью менеджеров правил и функций соответственно. Внешний вид этих менеджеров показан на рис. 9.2 и 9.3.

Для того чтобы запустить нашу экспертную систему, достаточно выполнить команду reset, которая добавит факт initial-fact, необходимый для правила system-banner, и команду run. После этого вы сразу увидите сообщение "The Engine Diagnosis Expert system", которое означает, что система начала работать, и получите серию вопросов, ответы на которые помогут экспертной системе оценить текущее состояние вашей машины и подобрать соответствующую рекомендацию по ремонту. Пример работы системы показан на рис. 9.4.

Обратите внимание, что если после завершения работы нашей экспертной системы в списке фактов CLIPS остаются факты, описывающие состояние автомобиля, их легко просмотреть с помощью команды **Fact Window** из меню **Window.** Факты для нашего примера изображены на рис. 9.5.

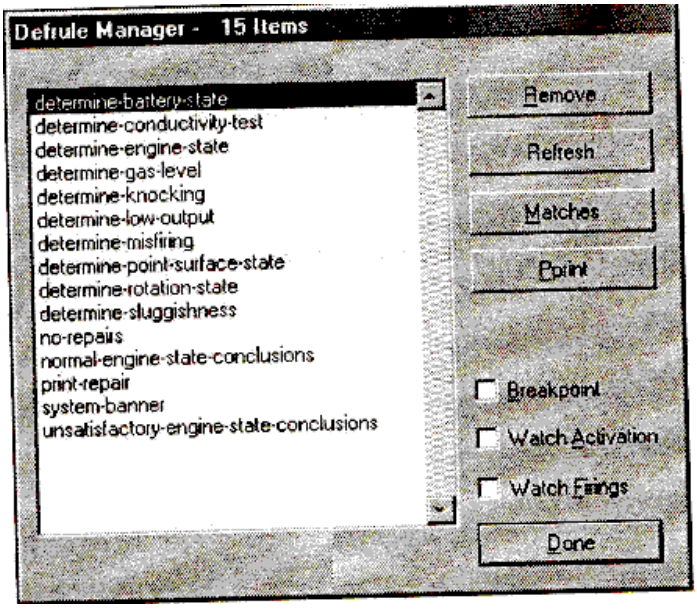

Рис. 9.2. Правила экспертной системы

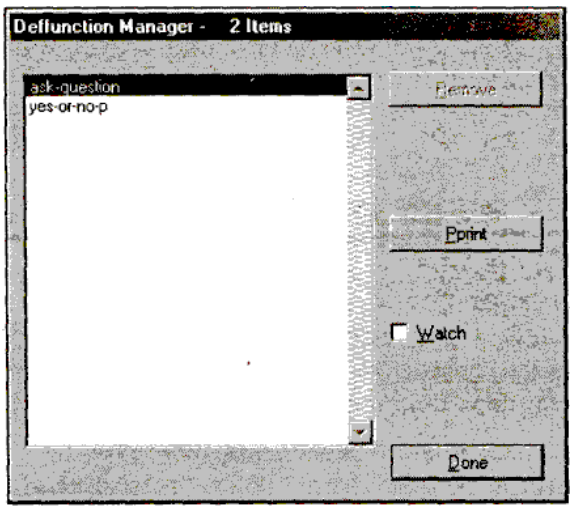

Рис. 9.3. Функции экспертной системы

Для повторного запуска экспертной системы необходимо еще раз выполнить команды reset и run. Протестируйте экспертную систему, по-разному отвечая на ее вопросы. Чтобы лучше понять механизмы ее работы и логический механизм вывода CLIPS, перед запуском системы сделайте видимым окно фактов **(Fact Window)** и окно плана решения задачи **(Agenda Window).** Приведенный в этой главе пример доступен в Интернете по адресу: **www.ghg.net/clips/download/executables/examples/auto.clp.**

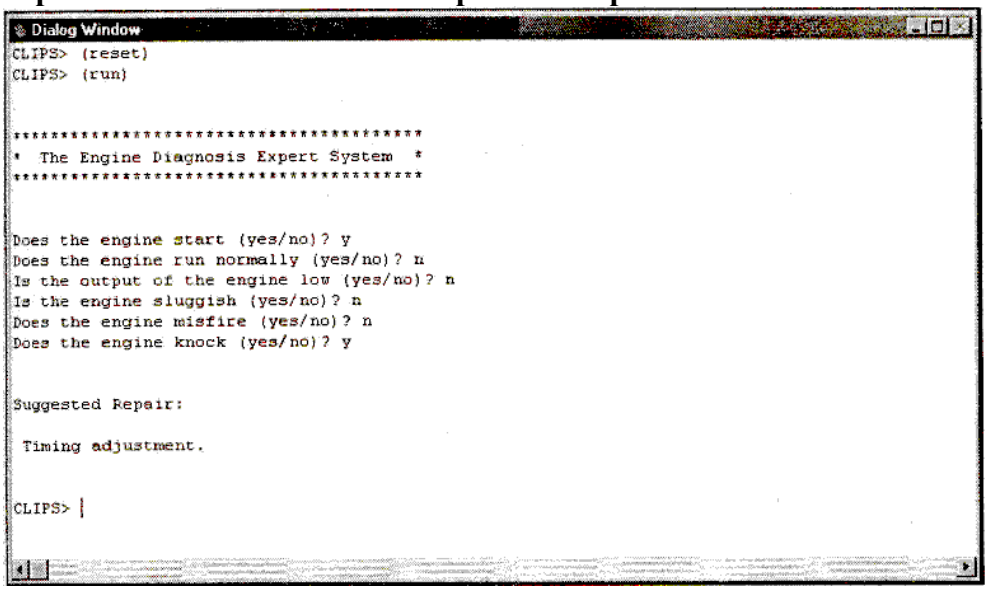

Рис. 9.4. Диалог с экспертной системой

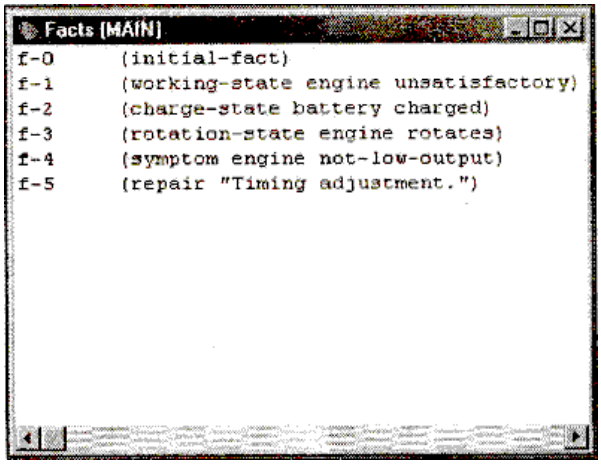

Рис. 9.5. Результаты работы экспертной системы

В данной главе был рассмотрен достаточно реальный пример построения экспертной системы в CLIPS. Он является наглядным подтверждением того, что с помощью фактов, функций и правил CLIPS можно построить вполне работоспособную систему. Однако CLIPS содержит и более сложные конструкции, такие как родовые функции, объекты и модули. С их помощью вы сможете строить еще более гибкие и мощные экспертные системы. Изучением этих конструкций мы и займемся в следующей главе.

ЧАСТЬ IV. Дополнительные возможности CLIPS.

ГЛАВА 10. Родовые функции

Помимо функций, описанных в *гл. 8,* CLIPS предоставляет еще один механизм, поддерживающий процедурную парадигму представления знаний, — *родовые функции.* Родовые функции подобны функциям, созданным с помощью конструктора deffunction. Они также могут использоваться для определения нового процедурного кода в CLIPS и могут быть вызваны как любые другие функции. Однако, в отличие от простых, родовые функции являются более мощным средством обработки данных, т. к. они способны выполнять различные наборы действий в зависимости от числа и типа полученных в данный момент аргументов. Родовые функции подобны перегруженным операторам или функциям языка C++. Например, функция + может выполнять как операцию конкатенации строк, так и простое арифметическое сложение чисел. Родовые функции определяются с помощью конструкторов defgeneric и defmethod, которые подробно будут описаны В данной главе. Родовые функции обычно состоят из нескольких компонентов, называемых *методами.* Каждый метод определяет последовательность действий, выполняющих обработку различных наборов аргументов. Родовая функция, которая имеет более одного метода, называется *перегруженной.*

Родовые функции могут содержать как системные методы, так и методы, определенные пользователем. Например, перегруженная функция + состоит из двух методов:

- *неявный* метод, являющийся системной функцией, которая обрабатывает арифметическое сложение;
- *явный* (определенный пользователем) обработчик сложения строк.

CLIPS не позволяет использовать функции, созданные с помощью конструктора deffunction, в качестве методов родовых функций. Конструкторы deffunction предоставляют возможность добавления в CLIPS новых функций без использования концепции перегрузки. Родовая функция, имеющая только один метод, по своему поведению идентична функции, созданной с помощью конструктора deffunction.

В большинстве случаев методы родовых функций не вызываются напрямую, несмотря на то, что CLIPS предоставляет такую возможность с помощью функции call-specific-method *(см. гл. 15).* CLIPS самостоятельно распознает вызов родовой функции и использует аргументы для поиска и запуска соответствующего метода. Этот процесс называется *родовым связыванием.*

10.1. Замечание относительно термина "метод"

Большинство объектно-ориентированных систем поддерживают процедурное поведение объектов через обработку сообщений (например, Smalltalk) или с помощью родовых функций (например, CLOS). Можно утверждать, что CLIPS поддерживает оба этих механизма, несмотря на то, что родовые функции и не являются составной часть языка COOL (объектно-ориентированный язык, поддерживаемый CLIPS) и могут использоваться без него. Родовые функции могут использовать классы в качестве аргументов своих методов, но они должны обеспечивать выдачу сообщений для манипуляции с объектами таких классов. (Изучением языка COOL мы займемся в следующей главе.)

Поддержка системой CLIPS обоих механизмов приводит к путанице в терминологии. В объектно-ориентированных системах, которые поддерживают только обработку сообщений, термин *"метод"* применяется для определения сообщения, реализованного в другом классе, по отношению к рассматриваемому. В системах, где поддерживаются только родовые функции, термин *"метод"* служит для определения различных реализаций поведения родовой функции.

Во избежание подобной путаницы в дальнейшем для указания на реализацию обработки сообщения в некотором классе будем использовать термин *"обработчик сообщений".* Термин *"метод"* станет употребляться только в контексте родовых функций.

10.2. Рекомендации по использованию родовых функций

Запуск родовых функций требует от системы больших манипуляций, чем вызов системных функций или функций, определенных с помощью конструктора deffunction. Это происходит потому, что CLIPS должен сначала исследовать аргументы родовой функции и определить, какой из ее методов применим в данном случае. Вызов родовой функции может быть на 15—20% медленнее вызова обычной функции. Поэтому не используйте родовые функции в операциях, для которых время критично. Например, не вызывайте родовую функцию в цикле, если это возможно.

Кроме того, родовые функции всегда должны иметь, по крайней мере, два метода. В случае если перегрузка функции не требуется, используйте обычные функции.

Если некоторый конструктор создается до определения родовой функции и использует системную или определенную пользователем функцию, которая впоследствии становится одним из методов родовой функции, родовое связывание не применяется. Например, если родовая функция, перегружающая функцию +, определена после правила с функцией +, то правило всегда станет вызывать системную функцию +. Однако если подобное правило будет определено после родовой функции, то при вызове функции + будет использоваться родовое связывание.

10.3. Создание родовой функции

Родовая функция состоит из *заголовка* (подобного предварительному объявлению функции) и *нескольких методов* (число которых теоретически может быть равным нулю). Заголовок родовой функции может быть либо явно определен пользователем, либо не явно объявлен определением метода. Объявление метода состоит из 6 элементов:

- имя (которое отображает, к какой основной функции относится метод);
- необязательный индекс;
- необязательные комментарии;
- набор ограничений для параметров;
- необязательный групповой параметр для обработки переменного числа аргументов;
- последовательность действий или выражений, которые будут выполнены в заданном порядке в момент вызова метода.

Ограничения параметров используются в процессе родового связывания для определения применимости метода к некоторому набору аргументов. Для создания заголовка родовой функции служит конструктор defgeneric, а для создания каждого нового метода родовой функции — конструктор defmethod.

Определение 10.1. Синтаксис конструктора defgeneric

```
(defgeneric <имя-функции>
            [комментарии])
```

Определение 10.2. Синтаксис конструктора defmethod

```
(defmethod <имя-функции>

           [<индекс>]

           [<комментарии>]

           (<ограничения-параметра>*

           [<групповой-параметр>])

           <действие>*)

<ограничения-параметров> ::= <простая-переменная> |
                             (<простая-переменная>
                              <ограничение-по-типу> *
                              [<ограничение-по-запросу>])

<групповой-параметр>   ::= <составная-переменная> |
                           (<составная-переменная>
                           <ограничение-по-типу>*
                           [<ограничение-по-запросу >])
 <ограничение-по-типу>   ::= <имя-класса>
 <ограничение-по-запросу> ::= <глобальная-переменная> |
                              <вызов-функции>
```

Родовая функция должна быть либо явно объявлена конструктором defgeneric, либо одним из своих методов, до того как она будет вызвана из другой функции, правила или обработчика сообщения. Исключение составляют рекурсивные родовые функции.

10.3.1. Заголовок родовой функции

Родовая функция однозначно определяется по своему имени. В случае использования родовой функции в правиле, другой функции, обработчике сообщения или до объявления первого метода родовой функции необходимо явное создание заголовка родовой функции (с помощью конструктора defgeneric). В других случаях объявление первого метода родовой функции само не явно создает заголовок. Например, в случае если две родовые функции имеют методы, которые взаимно вызывают друг друга (взаимная рекурсия родовых функций), то необходимо явное объявление заголовков.

10.3.2. Индексы методов

Метод родовой функции однозначно определяется либо по имени и индексу, либо по имени и ограничениям параметров. Каждому методу родовой функции назначается целый индекс, уникальный в группе всех методов этой функции. В случае если определен новый метод, который точно совпадает по ограничениям параметров и имени с другим методом этой функции, CLIPS автоматически заменит им существующий метод. Однако малейшее несовпадение в ограничениях параметров приведет к созданию нового метода. Если нужно заменить некоторый уже существующий метод новым методом с другими ограничениями параметров, то в определении нового метода необходимо явно указать индекс существующего метода. При этом ограничения параметров нового метода должны не совпадать с ограничениями других методов той же функции. Если индекс не задан, CLIPS автоматически назначает индекс, который еще не был использован другими методами родовой функции. Индекс, соответствующий методам родовой функции, можно определить, например, с помощью команды list-defmethods *(см. гл. 16).*

10.3.3. Ограничения параметров метода

Каждый параметр метода может быть определен с некоторыми произвольными комплексными *ограничениями* или без них. Ограничения параметров применяются к аргументам родовой функции во время работы программы для определения того, какой именно метод должен принимать эти аргументы. Параметр может иметь два типа ограничений: *ограничение типа* и *ограничение запросом*. Ограничение типа содержит классы аргументов, которые может принимать параметр. Ограничение запросом является определенным пользователем условным выражением, которое должно удовлетвориться для аргументов в момент вызова функции. Совмещение ограничений и их сложность прямо влияет на скорость родового связывания.

Если параметр не имеет ограничений, это означает, что метод может принимать любые значения в качестве данного аргумента. Однако каждый метод родовой функции должен иметь определенные ограничения параметров, которые будут отличать его от других методов той же родовой функции. В противном случае, процесс родового связывания не сможет определить, какой именно метод необходимо вызывать. В случае если процесс родового связывания не смог подобрать соответствующий метод для некоторого набора аргументов, CLIPS сгенерирует ошибку.

Ограничение типа позволяет пользователю определить список типов (классов), один из которых должен соответствовать (или являться суперклассом) аргументу родовой функции. Если в используемой вами конфигурации CLIPS не установлен COOL, то в качестве ограничения типа будут доступны только следующие типы (классы): object, primitive, lexeme, symbol,
STRING, NUMBER, INTEGER, FLOAT, MULTIFIELD, FACT-ADDRESS И EXTERNAL-ADDRESS.

В *гл. 11* все эти системные классы будут описаны подробно. Если COOL установлен, то, помимо перечисленных выше, будут доступны классы
INSTANCE, INSTANCE-ADDRESS, INSTANCE-NAME, USER, INITIAL-OBJECT, а также любой определенный пользователем класс. Родовая функция, которая использует только первую группу типов в своих методах, будет работать как с установленным COOL, так и без него. Классы, заданные в ограничении типа, должны быть определены до определения приоритета метода *(см. разд. 10.4.2).* CLIPS не поддерживает избыточность в списке ограничений типов аргументов

методов. Например, для представленного ниже метода ограничения типов аргументов избыточны, т. к. класс INTEGER — подкласс NUMBER.

Пример 10.1. Избыточные ограничения типов

```
(defmethod foo ((?a INTEGER NUMBER)))
```

Если ограничение типа удовлетворяется для некоторого аргумента, то к нему будет применено ограничение запросом (если оно задано). Ограничение запросом должно быть либо глобальной переменной, либо вызовом функции. CLIPS вычисляет заданное выражение, и если полученный результат не равен FALSE — ограничение полагается удовлетворенным.

Так как ограничения запросом вычисляются каждый раз при поиске соответствующего метода, они не могут использоваться для произведения какого-нибудь побочного действия, потому что вычисляемое ограничение может принадлежать методу, неподходящему к данной конкретной ситуации.

Поскольку все ограничения просматриваются слева направо, запрос с несколькими параметрами должен быть записан после ограничений типов всех используемых параметров. Этим правилом обеспечивается условие удовлетворения ограничений типов всех необходимых параметров. Например, метод из примера 10.2 не вычисляет ограничение запросом до тех пор, пока не удовлетворятся два соответствующих ограничения типа.

Пример 10.2. Использование ограничения запросом с двумя параметрами

```
(defmethod foo ((?a INTEGER ) (?b INTEGER(> ?a ?b))))
```

Если аргумент удовлетворяет всем своим ограничениям, то считается, что он применим для данного метода. Если все аргументы родовой функции применимы к ограничениям метода, метод полагается применимым для данного набора аргументов. В случае если существует более одного метода, применимого для некоторого набора аргументов, процесс родового связывания определяет некоторый упорядоченный список этих методов и использует первый метод из этого списка. Для создания списка служит приоритет методов, описанный в *разд. 10.4.2.*

В примере 10.3 первое обращение к родовой функции + вызовет выполнение системной функции + — неявный метод, выполняющий арифметическое сложение. Второй вызов приведет к выполнению явного метода родовой функции, осуществляющего конкатенацию строк, т. к. оба аргумента являются строками. Третий вызов сгенерирует ошибку, поскольку явный метод для конкатенации строк принимает только два аргумента, а неявный метод для арифметического сложения не принимает строковые аргументы вообще.

Пример 10.3. Перегрузка системной функции +

```
(defmethod + ( (?a STRING) (?b STRING))
             (str-cat ?a ?b) )
(+ 1 2)
(+ "foo" "bar")
(+ "foo" "bar" "woz")
```

10.3.4. Групповой параметр

В зависимости от того, задан ли групповой параметр, метод может принимать точное число параметров или число параметров не меньшее, чем некоторое заданное. Обязательные параметры определяют минимальное число аргументов, которое должно быть передано методу при его вызове. В действиях, выполняемых методом, можно ссылаться на каждый из этих параметров так же, как на обычные переменные, содержащие простые значения. Если был задан групповой параметр, то метод может принимать любое количество аргументов — большее или равное минимальному числу аргументов. Если групповой параметр не задан, то метод может принимать число аргументов, равное числу обязательных параметров. Все аргументы метода, которые не соответствуют обязательным параметрам, группируются в одно значение составного поля. Ссылаться на это значение можно, указывая символ группового параметра. Для работы с групповым параметром могут использоваться стандартные функции CLIPS, предназначенные

для работы с составными полями *(см.гл. 15),* такие как length и nth. Определение метода может содержать только один групповой параметр.

Ограничения типом и запросом могут применяться к аргументам, сгруппированным в групповом параметре, аналогично тому, как они употребляются с основными параметрами метода. Такие ограничения задаются для каждого отдельного поля результирующего составного значения (а не для всего значения). Выражение, содержащее групповой символ, может быть применено в запросе.

Дополнительно в запросе может быть использована специальная переменная ?current-argument для ссылки на отдельные аргументы, объединенные групповым символом. Это переменная существует только в ограничении запросом и не имеет значения в теле метода. Метод из примера 10.4 иллюстрирует версию функции +, которая находит полусумму любого количества четных целых чисел.

Пример 10.4. Еще один вариант функции +

```
(defmethod +
        (($?any INTEGER (evenp ?current-argument)))
        (div (call-next-method} 2))
```

Ограничения по типу и запросу для группового параметра применяются к каждому аргументу, сгруппированному групповым символом, даже если для проверки используются функции для работы с составным значением. Таким образом, в примере 10.5 функции > и length$ вызываются 3 раза для каждого из трех аргументов.

Пример 10.5. Ограничение запросом для группового поля

```
(defmethod foo
        (($?any (> (length$ ?any) 2)))
        TRUE)
(foo 1 red 3)
```

Кроме того, если у метода нет обязательных параметров (как в предыдущем примере) и функция вызвана без аргументов, заданные ограничения группового параметра не рассматриваются. Например, метод из примера 10.5 можно применить к следующему вызову родовой функции (метод успешно вызовется и вернет значение TRUE): (foo).

Как правило, ограничения запросом применяют ко всему групповому параметру для проверки *мощности* (числа аргументов, переданных методу). В таких случаях первое поле группового аргумента выносится в обязательный параметр (если это возможно). Приведенный выше пример можно усовершенствовать.

Пример 10.6. Улучшенная версия функции foo

```
(defmethod foo
        ((?arg (> (length$ ?any) 1)) $?any)
        TRUE)
```

Теперь попытка вызова функции (foo) без параметров закончится ошибкой.

10.4. Родовое связывание

В момент вызова родовой функции CLIPS выбирает метод с наивысшим приоритетом, для которого удовлетворяются все ограничения параметров. Этот метод выполняется, и его значение возвращается как значение родовой функции. Такой процесс называется *родовым связыванием.*

10.4.1. Применимость методов

Явный (определенный пользователем) метод применим к вызову родовой функции при следующих трех условиях:

- имя совпадает с именем родовой функции;
- метод принимает не меньше аргументов, чем родовая функция;

- каждый аргумент родовой функции удовлетворяет соответствующим ограничениям параметров метода.

Ограничения метода рассматриваются слева направо. Как только будет найдено одно ограничение, не удовлетворяющее некоторому параметру, метод забраковывается, и оставшиеся ограничения не рассматриваются.

При перегрузке стандартной системной функции CLIPS создает неявный метод с определением соответствующей системной функции. Этот неявный метод получает ограничения аргументов благодаря вызову внутренней системной функции DefineFunction2 (более детальную информацию о данной функции можно найти в книге *"CLIPS Reference Manual, Volume II, Advanced Programming Guide'}*. Строка с соответствующими ограничениями также может быть получена с помощью функции get-function-restriction. Определение неявного метода можно просмотреть функциями list-defmethods или get-method-restrictions.

Перечисленные ниже системные функции нельзя перегрузить. CLIPS сгенерирует сообщение об ошибке при попытке их перегрузки.

Определение 10.3. Список не перегружаемых внутренних функций CLIPS

active-duplicate-instance
active-initialize-instance
active-make-instance
active-message-duplicate-instance
active-message-modify-instance
active-modify-instance
any-instancep
assert
bind
break
call-next-handler
call-next-method
call-specific-method
delayed-do-for-all-instances
do-for-all-instances
do-for-instance
duplicate
duplicate-instance
expand$
find-all-instances
find-instance
if
make-instance
initialize-instance
loop-for-count
message-duplicate-instance
message-modify-instance
modify
modify-instance
next-handlerp
next-methodp
object-pattern-match-delay
override-next-handier
override-next-method
progn
progn$
return
switch
while

10.4.2. Приоритет методов

Когда два или более метода применимы к некоторому вызову родовой функции, CLIPS выполняет метод с наивысшим *приоритетом.* Приоритет метода определяется в момент его создания. Для того чтобы просмотреть приоритеты существующих методов, можно воспользоваться функцией list-defmethods *(см. гл. 15).*

Приоритет определяется сравнением ограничений параметров для пар методов. Метод с большим числом заданных ограничений параметров имеет больший приоритет. Кроме того, CLIPS учитывает диапазон значений, задаваемых приоритетом. Например, метод, который требует наличие типа integer, для некоторого аргумента имеет больший приоритет, чем метод, который требует для этого аргумента тип number. Ниже приведены правила, используемые CLIPS для определения приоритета между двумя методами.

1. Последовательно, слева направо сравниваются ограничения параметров обоих методов. Другими словами, первое ограничение параметра первого метода сравнивается с первым ограничением параметра второго метода и т. д. Сравнение между этими парами ограничений параметров двух методов определяет приоритет между двумя методами. Сравнение прекращается, как только будет найдена первая пара ограничений, однозначно определяющая метод с более высоким приоритетом. Для сравнения пар ограничений параметров применяются следующие правила в указанном порядке:
   - обязательные параметры имеют более высокий приоритет, чем групповой параметр;
   - более строгие ограничения типа имеют более высокий приоритет. Например, класс имеет

больший приоритет, чем его суперкласс;

- параметр с ограничением запроса имеет приоритет выше, чем параметр, который его не имеет.

2.Метод с большим числом постоянных параметров имеет больший приоритет.

3.Метод без групповых параметров имеет более высокий приоритет, чем метод с групповыми параметрами.

4.Если метод определен раньше другого, то первый метод имеет более высокий приоритет.

Если в одном ограничении задано несколько классов, определение приоритета усложняется. Поскольку определение приоритетов выполняется в момент создания нового метода, а конкретный класс аргумента станет известен только в момент вызова родовой функции, то для определения приоритета методов со списком классов в ограничении типа необходим специальный алгоритм для определения приоритета между двумя списками классов. В этом случае списки классов рассматриваются парами слева направо. Приоритет определяет первая пара, содержащая класс и его суперкласс. Список классов, содержащий класс, имеет более высокий приоритет, чем список, содержащий суперкласс данного класса. Если таких пар нет, то более приоритетным считается самый короткий список. В случае если приоритет списка классов установить не удалось, то ограничение параметра, использующее этот список классов, не рассматривается при определении приоритета метода.

Рассмотрим процесс определения приоритета методов на нескольких примерах.

Пример 10.7. Перегрузка функции

```
; Системная функция  '+'  является неявным методом                    ;#1

; Данный метод имеет следующее определение:
;  (defmethod +  {(?a NUMBER)  (?b NUMBER)  ($?rest NUMBER)))
(defmethod +  ((?a NUMBER)  (?b  INTEGER)))                           ; #2
(defmethod  +  ((?a  INTEGER)  (?b  INTEGER)))                        ; #3
(defmethod +  ( (?a  INTEGER)  (?b NUMBER) ))                         ; #4
(defmethod  +  ((?a NUMBER)  (?b NUMBER)
              ($?rest  PRIMITIVE)))                                   ; #5
(defmethod +  ((?a NUMBER)  (?b  INTEGER  (>  ?b  2 ) ) ) )           ; #6
(defmethod +( ( ?a  INTEGER  (>  ?a  2 ) )
             (?b  INTEGER  (>  ?b  3 ) ) ) )                          ;#7
(defmethod  +( ( ?a  INTEGER  (>  ?a  2 ) )  (?b NUMBER)))            ; #8
```

Приоритет методов, в приведенном выше примере, будет следующим: #7, #8, #3, #4, #6, #2, #1, #5. В данной ситуации для определения приоритета методы могут быть сразу разбиты на три группы по уменьшению приоритета с помощью ограничения, заданного для их первого параметра:

- методы, которые имеют ограничение и типом и запросом для класса integer (#7', #8);
- методы, которые имеют ограничение типа для класса integer (#3, #4);
- методы, которые имеют ограничение типа для класса number (#1, #2, #5, #6).

Методы первой группы имеют более высокий приоритет, чем методы второй группы, т. к. ограничение параметров типом и запросом имеет более высокий приоритет, чем ограничение только типом. Вторая группа имеет более высокий приоритет, чем третья группа, поскольку integer является подклассом number. Поэтому порядок методов можно представить так:
(#7, #8) (#3, #4) (#1, *#2, #5,* #6).

Следующим шагом будет определение приоритета между методами в выделенных нами группах, учитывая ограничение по второму параметру. Метод #1 имеет больший приоритет, чем #8, т. к. integer является подклассом number. Метод #3 имеет больший приоритет по сравнению с #4 по той же причине. Методы *#6* и #2 имеют приоритет больший, чем методы #1 и #5, т. к. integer является подклассом number. Метод #6 имеет больший приоритет по сравнению с #2, т. к. #6 имеет ограничение запросом, а #2 нет. Таким образом, порядок методов примет следующий вид: #7, #8, #3, #4, #6, #2, (#1, #5)

Ограничение по групповому аргументу дает методу #1 (системной функции + — неявному методу родовой функции +) больший приоритет по сравнению с методом #5, т.к. number является

подклассом primitive. Итак, окончательным порядком расстановки методов по приоритету будет такой: #7, #8, #3, #4, #6, #2, #1, #5.

Пример 10.8. Определение приоритета

```
(defmethod foo   ((?a NUMBER STRING)))     ;  #1
(defmethod foo   (?a  INTEGER LEXEME)))    ;  #2
```

Несмотря на то, что класс string является подклассом класса lexeme, порядок методов будет *#2,* #1, т. к. integer является подклассом number, а пара number/integer находится левее пары string/lexeme в списке классов.

Пример 10.9. Определение приоритета

```
(defmethod foo ((?a MULTIFIELD STRING)))  ; #1
(defmethod foo ((?a LEXEME)))             ; #2
```

Порядок методов из примера 10.9 будет следующим: #2, *#1*. Такой порядок определяется тем фактом, что классы первой пары ограничений типов — multifield/iexeme — не связаны, а метод *#2* имеет более короткий список классов.

Пример 10.10. Определение приоритета

```
(defmethod foo ((la INTEGER LEXEME)))     ; #1
(defmethod foo ((la STRING NUMBER)))      ; #2
```

В данном примере обе пары классов (integer/string и lexeme /number) не связаны. Кроме того, списки классов ограничений имеют одинаковую длину. Таким образом, приоритет будет установлен по порядку создания следующих методов: #1, #2.

10.4.3. Скрытые методы

Если один из методов родовой функции вызывается другим, то такой метод называется *скрытым*. Обычно, только один метод должен быть применим к конкретному вызову родовой функции. Если для данного вызова существует больше одного применимого метода, родовое связывание выполнит метод с наивысшим приоритетом. Такой подход называется *декларативным методом* родового связывания.

Однако с помощью функций call-next-method и override-next-method метод родовой функции может вызвать некоторый другой метод данной родовой функции (скрыть вызов). Такой подход называется *императивным методом* (после вызова некоторого метода он играет роль родового связывания).

Не рекомендуется использовать данный подход без крайней необходимости. В большинстве случаев обработку вызова с заданным набором аргументов должен осуществлять только один метод.

Помимо функций call-next-method и override-next-method для реализации императивного подхода можно использовать функцию call-specific-method для перегрузки установленного приоритета метода *(см. гл. 15)*.

10.4.4. Ошибки выполнения метода

Если в момент выполнения происходят ошибки, то любое еще невыполненное действие в текущем методе будет прервано, а любой еще невызванный метод не будет вызван. Родовая функция в этом случае вернет значение FALSE.

Если не был найден метод, применимый к конкретному вызову родовой функции, то такая ситуация расценивается как ошибка выполнения метода.

10.4.5. Значение, возвращаемое родовой функцией

Значение, возвращаемое родовой функцией, является значением, возвращаемым применимым методом с наивысшим приоритетом. Каждый применимый метод может игнорировать или использовать значение, возвращаемое всеми скрытыми методами. Значение, возвращаемое методом, является последним действием, вычисленным в действиях данного метода.

10.5. Визуальные инструменты для работы с родовыми функциями

В заключение данной главы рассмотрим визуальные инструменты, которые предоставляет CLIPS для работы с родовыми функциями (подробное описание функций и команд для работы с родовыми функциями и их методами *см. в гл. /*5 и *16).*

Для работы с родовыми функциями Windows-версия среды CLIPS предоставляет инструмент — **Defgeneric Manager** (Менеджер родовых функций). Для его запуска выберите пункт **Defgeneric Manager** в меню **Browse.** Соответствующий пункт в меню недоступен, если в данный момент в среде не определена ни одна родовая функция. Общий вид менеджера представлен на рис. 10.1.

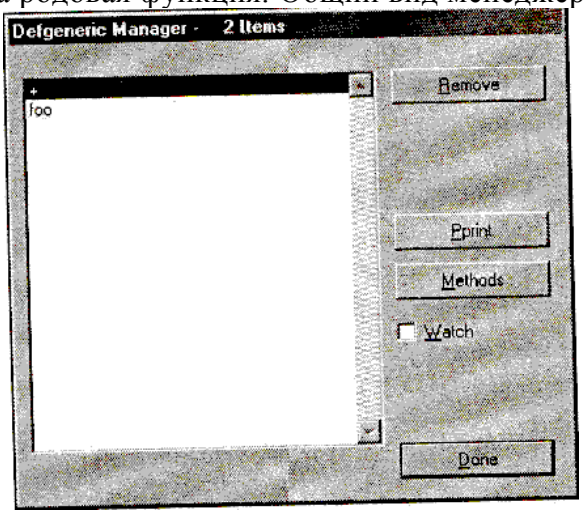

Рис. 10.1. Окно менеджера родовых функций

Общее количество родовых функций отображается в заголовке окна менеджера — **Defgeneric Manager — 2 Items.** С помощью этого инструмента вы можете удалить родовую функцию из системы (кнопка **Remove),** вывести на экран ее определение (кнопка **Pprint),** установить режим просмотра вызова отдельной функции и вызвать менеджер методов для заданной функции (кнопка **Methods).**

Учтите, что удаление родовой функции приводит к удалению всех ее методов.

Для тренировки использования родовых функций и менеджера родовых функций очистите CLIPS и добавьте в него методы, приведенные в примере 10.11.

Пример 10.11. Перегрузка функции +

```
(defmethod + ((?a INTEGER (> ?a 0)) (?b INTEGER (> ?b 0) ) )
             (call-next-method))
(defmethod + ((?a INTEGER) (?b FLOAT))
             (call-next-method))
(defmethod + ((?a FLOAT) (?b FLOAT))
             (call-next-method))
(defmethod + ((?a STRING) (?b STRING))
             (str-cat ?a ?b))
```

Обратите внимание на реализацию методов для сложения чисел. После проверки своих аргументов они просто вызывают системную функцию +. Если бы мы вместо вызова (call-next-method) использовали системную функция + напрямую (+ ?a ?b), то получили бы бесконечную рекурсию, которая привела бы к переполнению стека и аварийному завершению программы.

Попробуйте несколько раз вызвать функцию + с различными аргументами:

Пример 10.12. Тестирование родовой функции +

```
(+ "Hello "   "World")
(+ 1 3)
(+ 1  3.5)
(+  1.5   3)
(+  1  -3)
(+ 1.5  3.0)
(+ 1.5  3.0  5.0)
(+"Hello "   "World"   "!!!")
(+  1  3.5   4)
```

Полученный результат должен соответствовать приведенному на рис. 10.2.

Обратите внимание, что для вызовов (+ 1.5 3), (+ 1 -3), (+ 1.5 3.0 5.0), (+ 1 3.5 4) применяется вызов системной функции +, т. к. мы не определили методов, способных принять такие аргументы, но, тем не менее, мы получили корректные ответы. Родовое связывание не смогло подобрать метод, применимый к вызову (+"Hello " "World" "!!!") (наша функция для конкатенации строк принимает строго два аргумента), поэтому мы получили соответствующее сообщение об ошибке.

Установите режим отображения вызова родовой функции с помощью менеджера и попробуйте еще раз повторить вызовы, приведенные выше. Обратите внимание на сообщения о вызовах родовой функции.

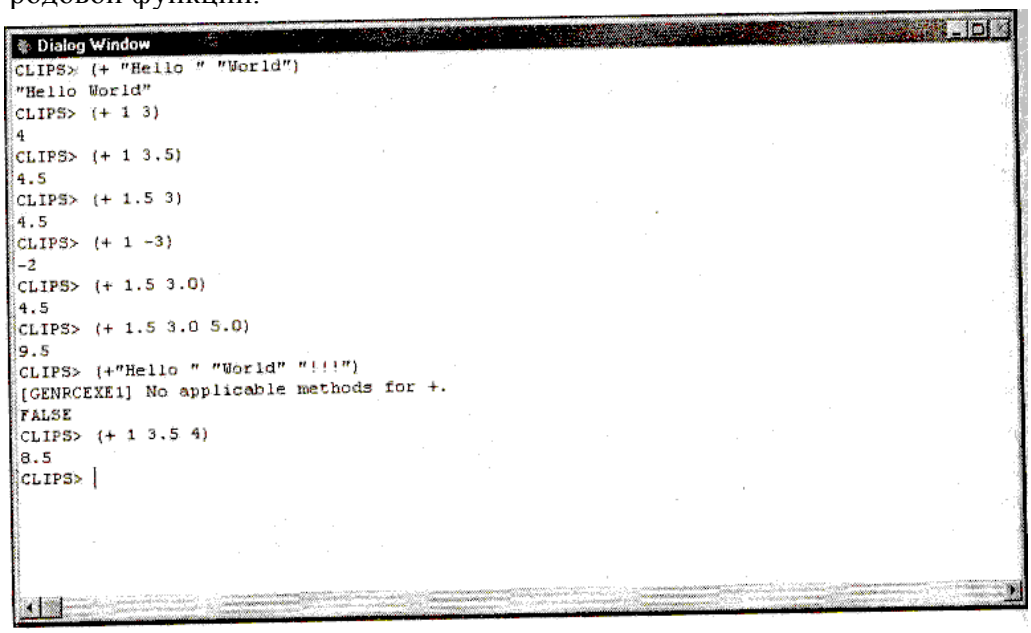

Рис. 10.2. Результаты тестирования родовой функции +

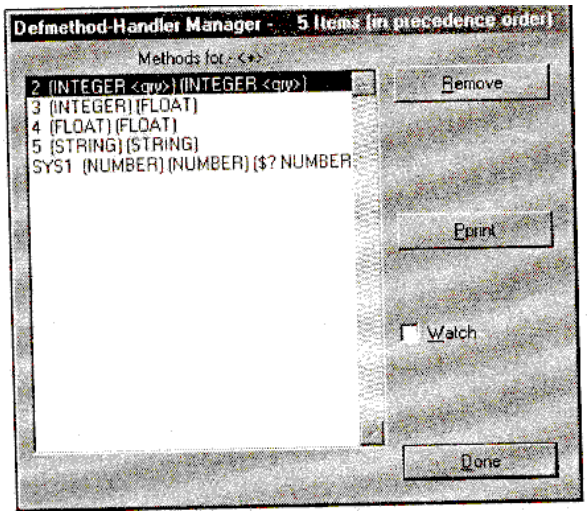

Рис. 10.3. Окно менеджера методов родовой функции

Defmethod-Handler Manager (Менеджер методов родовой функции) — еще один инструмент, предоставляемый CLIPS. Внешний вид этого инструмента представлен на рис. 10.3. Этот инструмент выводит на экран список методов родовой функции, указанной менеджером родовых функций. Список методов сортируется по приоритету, установленному для этих методов. Общее количество методов заданной родовой функции отображается в заголовке окна менеджера — **Defmethod-Handler Manager — 5 Items (in precedence order).**

С помощью менеджера методов вы можете удалить некоторый метод (кнопка **Remove**), вывести на экран его определение (кнопка **Pprint**) или установить режим просмотра вызовов отдельного метода. Обратите внимание, что метод, неявно определенный системой, например метод, представляющий системную функцию +, не может быть удален.

Снимите установку вывода сообщений о вызове родовой функции + и установите вывод сообщений о вызове методов с помощью менеджера методов. Выполните следующие вызовы:

Пример 10.13. Тестирование родовой функции +

```
(+ "Hello " "World")
(+ 1 3)
(+ 1 3.5)
(+ 1 -3)
```

Результат этих действий представлен на рис. 10.4.

Обратите внимание, что, в случае получения сообщений о вызове конкретного метода родовой функции, мы можем получить информацию о том, какой именно метод обработал полученный вызов.

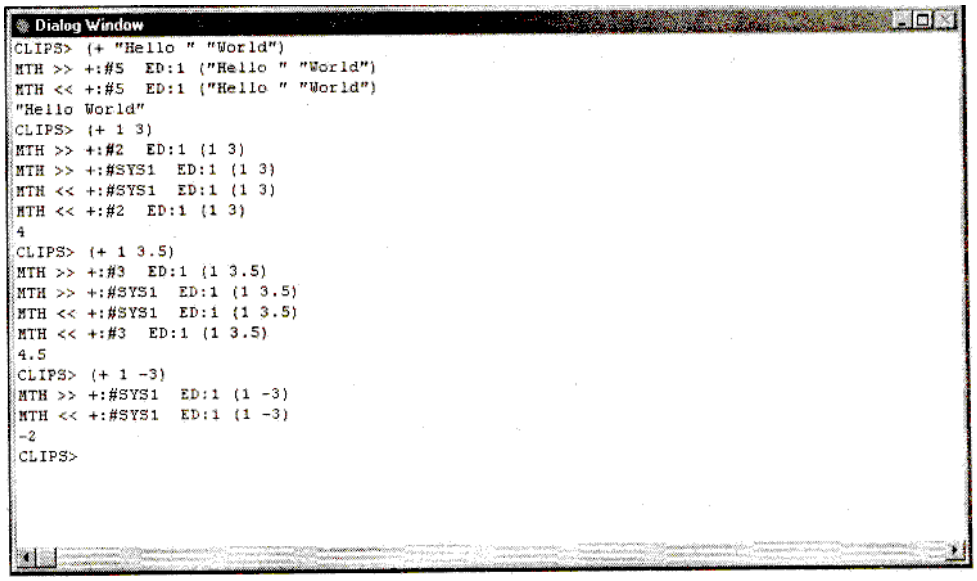

Рис. 10.4. Результаты тестирования родовой функции +

В случае если вы хотите установить режим просмотра вызовов всех методов или всех родовых функций, воспользуйтесь диалоговым окном **Watch Options** из меню **Execution.** Установите флажки в полях **Generic Functions** или/и **Methods,** как показано на рис. 10.5.

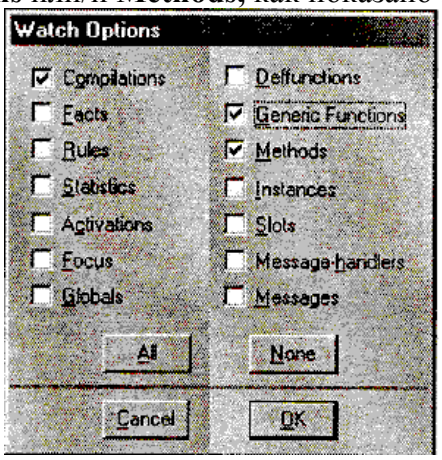

Рис. 10.5. Установка режима отображения вызовов родовых функций и методов

В данной главе была описана такая конструкция языка CLIPS, как родовые функции, методы их создания, приемы и способы использования, а также алгоритм родового связывания, делающий возможным функционирование родовых функций. Как вы успели заметить, в качестве ограничения типов для аргументов методов родовых функций используются классы. Внутренние классы CLIPS, а также объектно-ориентированный язык COOL (расширение CLIPS) будут рассмотрены в следующей главе.

ГЛАВА 11. Объектно-ориентированный язык CLIPS

В данной главе приводится подробное описание деталей языка COOL (CLIPS Object-Oriented Language). Благодаря наличию языка COOL, пользователи CLIPS могут манипулировать не только фактами и правилами, переменными и функциями, но и объектами. Объекты позволяют объединять данные со способами их обработки. Объекты CLIPS можно использовать в правилах и функциях в качестве данных почти так же, как факты или переменные.

COOL вобрал в себя идеи различных объектно-ориентированных систем, а также привнес несколько новых идей. Например, концепции инкапсуляции объектов подобны их представлению в Smalltalk, а Common Lisp Object System (CLOS) предоставила основы правил множественного наследования. Основные идеи из Smalltalk, CLOS и других систем заложили базис для сообщений CLIPS. В *разд. 10.1* объясняется важная разница между терминами "метод" и "обработчик сообщений" в CLIPS.

11.1. Предопределенные системные классы

COOL предоставляет 17 системных классов: OBJECT, USER, INITIAL-OBJECT, PRIMITIVE, NUMBER, INTEGER, FLOAT, INSTANCE, INSTANCE-NAME, INSTANCE-ADDRESS, ADDRESS, FACT-ADDRESS, EXTERNAL-ADDRESS, MULTIFIELD, LEXEME, SYMBOL И STRING.
Пользователь не может удалять или изменять эти классы. На рис. 11.1 представлена диаграмма наследования этих классов.

Все перечисленные системные классы, за исключением INITIAL-OBJECT, являются абстрактными. Это означает, что они могут быть использованы только для наследования и не могут применяться для создания экземпляра объекта данного класса. Ни один из этих классов не имеет слотов и, за исключением класса USER, ни один из них не имеет обработчиков сообщений. Однако пользователь может явно присоединить обработчики сообщений ко всем системным классам за исключением INSTANCE, INSTANCE-ADDRESS и INSTANCE-NAME. Класс OBJECT является суперклассом для всех остальных классов, включая классы, определенные пользователем. Все классы, определенные пользователем, должны (хотя, в крайнем случае, это условие можно нарушить) наследоваться прямо или косвенно от класса USER, поскольку этот класс имеет все стандартные системные обработчики сообщений, например, обработчики сообщения для инициализации или удаления объекта к нему уже присоединены (эти системные обработчики сообщений описываются в *разд.11. 3.3).*

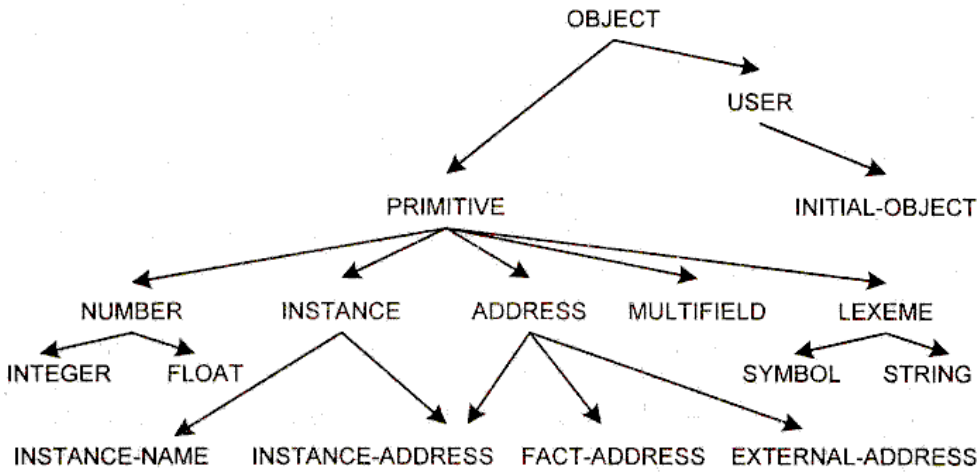

Рис. 11.1. Диаграмма наследования системных классов CLIPS

Системный класс PRIMITIVE и все его подклассы предоставляют классы, чаще всего используемые в родовых функциях для ограничения типов. Но при необходимости к ним можно добавить новые обработчики сообщений или унаследовать от них новые классы. Однако три системных наследника класса: PRIMITIVE — INSTANCE, INSTANCE-ADDRESS и NSTANCE-NAME — предназначены только для использования в методах родовых функций (в частности они применяются при формировании неявно объявленных методов, перегружающих системные функции — *см. разд. 10.5.1).* Эти классы не могут иметь подклассов или обработчиков сообщений.

Класс INITIAL-OBJECT используется в конструкторе definstances по умолчанию для создания объекта [initial-object] во время выполнения команды reset. Это *конкретный системный класс* (термин "конкретный класс" будет введен чуть позже) отвечает за сопоставления образцов в левой части правил, хотя он во всех отношениях идентичен системному классу USER.

11.2. Конструктор *defclass*

Конструктор defclass создает новый пользовательский класс в среде CLIPS. Он определяет свойства (слоты) и поведение (обработчики сообщений) класса объектов. Конструктор defclass состоит из пяти элементов:

- имя;
- список суперклассов, от которых новый класс наследует слоты и обработчики сообщений;
- спецификатор, объявляющий, позволять или нет создание прямых объектов нового класса;
- спецификатор, объявляющий, могут или нет экземпляры этого класса использоваться при сопоставлении образцов объектов в левой части правил;
- список слотов, определенных в новом классе.

Все классы, определенные пользователем, должны наследоваться, по крайней мере, от одного класса. С этой целью COOL предоставляет предопределенные системные классы для использования в качестве базовых в новых производных классах.

Любой слот, явно заданный в defclass, заменяет (перегружает) слоты, полученные с помощью наследования от базовых классов. Для всех новых классов COOL к списку суперклассов применяет специальный алгоритм получения *списка предшествования классов (см. разд. 11.2.1).* Слоты класса описываются с помощью *граней* (facets). Примеры некоторых граней, включая грань значения по умолчанию, грань мощности (количества элементов) и грань типа доступа, представлены ниже.

Определение 11.1. Синтаксис конструктора defclass

```
(defclass      <имя-класса>  [<комментарии>]
                             (is-a <список-суперклассов>+)
                             [<роль-класса>]
                             [<активность-класса >]
                              <слот>*
                             <объявление-обработчика-сообщений>*)
<роль-класса>                ::= (role concrete | abstract)
<активность-класса>          ::= (pattern-match reactive | non-reactive)
<слот>                       ::= (slot <имя> <грани>*) |
                                 (single-slot <имя> <грани>*)|
                                 (multislot <имя> <грани>*)
<грань>                      ::= < значение-по-умолчанию > |
                                 <грань-хранения> |<грань-доступа> |
                                 <грань-распространения> |
                                 <грань-источника> |
                                 <грань-сопоставления-образцов> |
                                 <грань-видимости> |
                                 <грань-создания-акцессоров> |
                                 <грань-переопределения-сообщений> |
                                 <ограничения-атрибутов>
<значение-по-умолчанию>              ::= (default ?DERIVE | ?NONE | <выражение>*) |
                                         (default-dynamic <выражение>*)
<грань-хранения>                     ::= (storage local | shared)
<грань-доступа>                      ::= (access read-write | read-only | initialize-only)
<грань-распространения >             ::= (propagation inherit | no-inherit)
<грань-источника >                   ::= (source exclusive | composite)
<грань-сопоставления-образцов >      ::= (pattern-match reactive | non-reactive)
<грань-видимости >                   ::= (visibility private | public)
<грань-создания-акцессоров >         ::= (create-accessor ?NONE | read | write | read-write )
```

<грань-переопределения-сообщений > ::= (override-message ?DEFAULT │<имя-сообщения>)
<объявление-обработчика-сообщений >::= (message-handler <имя-обработчика>
[<тип-обработчика>])
<тип-обработчика > ::= primary │ around │ before │ after

Замечание

В представленном выше определении значения по умолчанию подчеркнуты.

Попытка переопределения существующего класса приведет к удалению этого класса и всех ассоциированных с ним обработчиков сообщений. Если определен хотя бы один объект удаляемого класса, произойдет ошибка, и новый класс не будет создан.

11.2.1. Множественное наследование

Если один класс наследуется от другого класса, первый класс называется *подклассом* класса, а второй класс — *суперклассом* первого. Каждый определенный пользователем класс должен иметь хотя бы один прямой суперкласс, т. е. по крайней мере один класс должен появиться в части is-a конструктора defclass.

Множественное наследование происходит, когда класс имеет более одного суперкласса. В этом случае COOL проверяет список прямых суперклассов для нового класса и устанавливает некоторый линейный порядок, называемый *списком предшествования классов.* Новый класс наследует слоты и обработчики сообщений каждого класса в списке предшествования классов.

Слоты и обработчики сообщений класса, стоящего в начале списка предшествования, переопределяют конфликтующие определения из других классов, ранее найденных в списке предшествования. Класс, стоящий перед другим классом в списке предшествования, называется *более определенным.* Все списки предшествования классов оканчиваются системным классом OBJECT, и большинство (если не все) списков предшествования классов для определенных пользователем классов заканчиваются парой системных классов — USER и OBJECT. Вывести на экран список предшествования можно с помощью функции describe-class *(см. гл. 15).*

Для определения списка предшествования классов нового класса COOL использует иерархию наследования от непосредственного суперкласса. COOL рекурсивно применяет следующие два правила к непосредственным суперклассам класса, список предшествования которого необходимо определить:

1.Класс имеет более высокий приоритет, чем любой из его суперклассов.

2.Определение класса задает приоритет между его непосредственными суперклассами.

В случае если, воспользовавшись приведенными выше правилами, можно построить более одного списка предшествования, то COOL построит наиболее простой список, получаемый с помощью обхода дерева наследования класса в глубину. Эта эвристика максимально долго сохраняет родственное дерево класса. Например, если ребенок наследует основные черты от матери и отца, а мать и отец в свою очередь наследуют черты от своих родителей, список предшествования классов для ребенка будет следующим: ребенок, мать, мать матери, отец матери, отец, мать отца, отец отца. Возможны и другие варианты создания списка предшествования, удовлетворяющие правилам, например: ребенок, мать, отец, отец отца, мать матери, мать отца, отец матери. Однако COOL воспользуется первым вариантом, т. к. он сохраняет фамильное дерево класса настолько долго, насколько это возможно.

Приведем несколько примеров, иллюстрирующих построение списка предшествования класса.

Пример 11.1. Класс A

```
(defclass A   (is-a USER))
```

Класс A наследует свои свойства непосредственно от класса USER. Список предшествования классов для A: A USER OBJECT.

Пример 11.2. Класс B

```
(defclass  B   (is-a  USER))
```

Класс в также наследует свои свойства непосредственно от класса USER. Список предшествования классов для B: B USER OBJECT.

Пример 11.3. Класс C

```
(defclass  C   (is-a A B))
```

Класс C наследует свои свойства от классов A и B. Список предшествования классов для C: C A B USER OBJECT.

Пример 11.4. Класс D

```
(defclass  D   (is-a B A))
```

Класс D наследует свои свойства от классов B и A. Список предшествования классов для D: D B A USER OBJECT.

Пример 11.5. Класс E

```
(defclass  E   (is-a A C))
```

По правилу 2 класс A должен предшествовать C. Однако C является подклассом A и не может следовать за A в списке предшествования классов без нарушения правила 1. Таким образом, подобное определение класса E является ошибкой.

Пример 11.6. Допустимое определение класса E

```
(defclass  E   (is-a C A))
```

Указание класса A в списке суперклассов класса E в данном случае является излишним, т. к. класс C является наследником A. Однако такое определение класса E не нарушает правил, используемых для определения списка предшествования классов, и допустимо. Список предшествования классов для E: E C A B USER OBJECT.

Пример 11.7. Класс F

```
(defclass F   (is-a C B))
```

Указание класса B в списке суперклассов класса F излишне, т. к. C является наследником B. Список предшествования классов для F: F C A B USER OBJECT. По правилу 2, в списке предшествования классов для класса F, в должно следовать за C, но это не означает, что класс в должен непосредственно следовать за классом C.

Пример 11.8. Класс G

```
(defclass  G   (is-a  C  D) )
```

В данном конструкторе defclass допущена ошибка, нарушающая правило 2. Определение класса C утверждает, что класс A должен предшествовать классу B, но список предшествования классов D говорит обратное.

Пример 11.9. Классы H, I и J

```
(defclass H  (is-a A) )
(defclass  I   (is-a B) )
(defclass  J   (is-a H I A B) )
```

Списки предшествования классов H и I соответственно: H A USER OBJECT и I B USER OBJECT. Если J не имело бы A и B в качестве прямых суперклассов, то J мог бы иметь один из трех возможных списков предшествования классов: J H A I B USER OBJECT, J H I A B USER OBJECT или J H I B A USER OBJECT. Из этих трех вариантов COOL выбрал бы первый список, т. к. он отражает наследование (H A и I B) наиболее реально. Однако поскольку J прямо наследует от A и B, то правило 2 изменяет список предшествования следующим образом: J H I A B USER OBJECT.

В качестве последнего воспользуемся приведенным выше примером с ребенком, наследующим свойства своих родителей. Введите в CLIPS следующие конструкторы новых классов.

Пример 11.10. Класс _child

```
(defclass  _fgf (is-a USER))  ;отец отца
(defclass  _fgm (is-a USER))  ;мать отца
(defclass  _father (is-a _fgm  _fgf))
(defclass  _mgm (is-a USER) )  ;мать матери
(defclass  _mgf (is-a USER))  ;отец матери
(defclass  _mother (is-a _mgm  _mgf))
(defclass  _child (is-a _mother  _father))
```

Выполните команду (describe-class _child). Полученный результат должен соответствовать рис. 11.2.

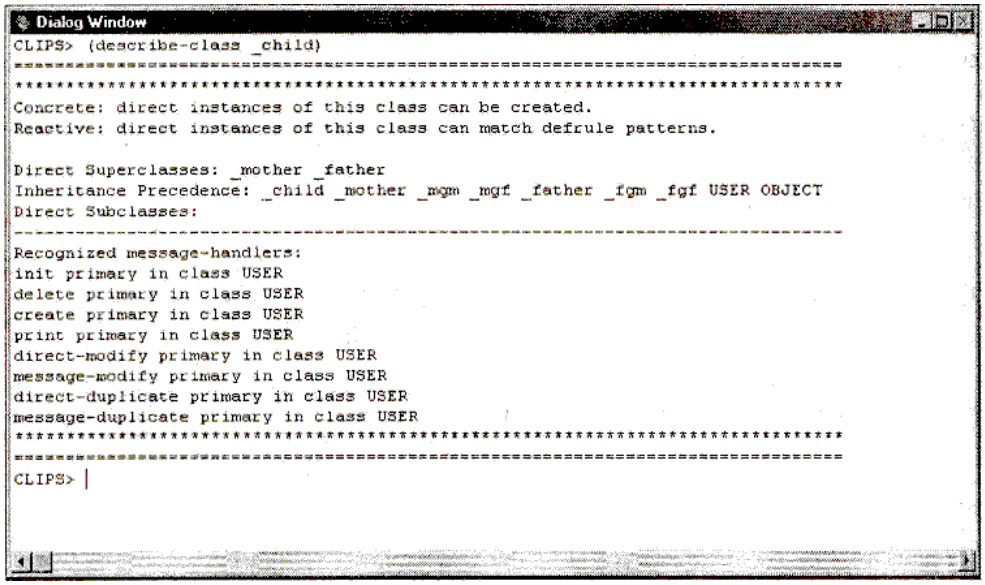

Рис. 11.2. Списков предшествования классов для класса _child

Команда describe-class выводит на экран довольно много информации о заданном классе. Значение этой информации станет более понятно по мере изучения различных свойств конструктора defclass и обработчиков сообщений. Сейчас обратите внимание только на список непосредственных суперклассов (Direct Superclasses) и на список предшествования классов (Inheritance Precedence) для _child.

11.2.2. Абстрактные и конкретные классы

С помощью спецификатора роли класса можно задать роль для класса, создаваемого конструктором defclass. Класс может быть либо *абстрактным,* либо *конкретным.*

Определение 11.2. Синтаксис спецификатора роли класса

<роль-класса> ::= (role concrete | abstract)

Абстрактные классы предназначаются только для наследования. Нельзя создать экземпляры объектов абстрактного класса. Конкретные классы могут быть использованы как для наследования, так и для создания экземпляров объектов этих классов.

Если с помощью спецификатора роли был создан абстрактный класс, а потом данный класс был использован в функции make-instance для создания экземпляра объекта этого класса, COOL сгенерирует соответствующую ошибку.

Если спецификатор роли класса не был задан, роль класса определяется наследованием. В этом случае роль нового класса принимает значение роли первого непосредственного суперкласса из списка наследования (пример 11.11).

Пример 11.11. Наследование спецификатора роли класса

```
(defclass A  (is-a      USER)
             (role  concrete))
(defclass  B  (is-a     USER))
(defclass  C  (is-a     A B) )
(defclass  D  (.is-a    B A) )
```

Класс A является конкретным, поскольку это явно задано спецификатором роли. Класс B — абстрактным, т. к. его первый суперкласс USER абстрактный. Класс C будет конкретным классом, потому что первый его суперкласс (A) является конкретным классом, а класс D, напротив, станет абстрактным классом, поскольку абстрактным классом является класс B , его первый суперкласс в списке наследования.

11.2.3. Активные и неактивные классы

С помощью спецификатора активности класса можно задать поведение объекта данного класса при проведении сопоставления образцов. Класс может быть либо *активным,* либо *неактивным.*

Определение 11.3. Синтаксис спецификатора активности класса

<активность-класса> ::= (pattern-match reactive | non-reactive)

Объекты активного класса могут применяться при сопоставлении образцов во время выполнения правил. Объекты неактивного класса не могут использоваться при сопоставлении образцов и не принимают участие в определении списков классов, применимых для сопоставления образцов. Абстрактный класс не может быть активным.

Если спецификатор роли класса не был задан явно, активность класса определяется наследованием. В этом случае активность нового класса принимает значение активности первого непосредственного суперкласса из списка наследования (пример 11.12).

Пример 11.12. Наследование спецификатора активности класса

```
(defclass A  (is-a USER)
             (role concrete)
             (pattern-match reactive))
(defclass B  (is-a USER)
             (role concrete))
(defclass C  (is-a A B))
(defclass D  (is-a B A))
```

Класс A является активным, т. к. это явно задано спецификатором активности класса. Класс B — неактивным, поскольку он наследуется от абстрактного, а значит, и неактивного класса USER. Класс C будет активным классом, потому что первый его суперкласс (A) активен. Класс D, напротив, будет неактивным, т. к. первый суперкласс в его списке наследования (B) неактивен.

11.2.4. Слоты класса

Слот — это место для хранения значений, ассоциированных с объектом определенного пользователем класса. Каждый экземпляр объекта имеет свою копию набора слотов, определенных в его классе непосредственно, а также копию любых слотов, полученных наследованием. Количество слотов ограничивается только доступной памятью вашего компьютера. Именем слота может быть любое значение типа symbol за исключением ключевых слов is-a и name, которые зарезервированы для использования в образцах объекта.

Для определения набора слотов объекта список предшествования классов рассматривается в порядке от более определенных к более общим классам (слева направо). Класс является более определенным, чем любой его суперкласс. Слоты, определенные в любом классе в списке предшествования классов, помещаются в экземпляр объекта за исключением не наследуемых слотов *(см. подразд. "Грань распространения при наследовании" данного раздела).* Если слот наследуется более чем от одного класса, то используется определение, данное ему в наиболее определенном классе, за исключением композитных слотов *(см. подразд. "Грань источника" этого раздела).*

Рассмотрим пример 11.13.

Пример 11.13. Наследование слотов

```
 (defclass A (is-a USER)
             (slot fooA)
             (slot barA))
(defclass B  (is-a A)
             (slot fooB)
             (slot barB))
```

Список предшествования для класса A: A USER OBJECT. Экземпляр класса A будет иметь два слота: fooA и barA. Список предшествования для класса B: B A USER OBJECT. Экземпляр класса в будет иметь четыре слота: fooB, barB, fooA и barA.

Для определения свойств слотов используются грани (facets). Они описывают различные особенности слота, которые присущи всем объектам, содержащим этот слот. С помощью граней можно устанавливать следующие свойства слотов: значение по умолчанию, место хранения, доступ, распространение при наследовании, источники граней, активность при сопоставлении образцов, видимость для обработчиков сообщений подкласса, автоматическое создание обработчиков сообщений для доступа к слотам, имя посылаемого сообщения для установки слота или ограничения на значения слота.

Каждый объект может хранить свое собственное значение в некотором слоте за исключением общих слотов *(см. подразд. «Грань хранения» этого раздела).*

Тип слота

Слот может содержать значение либо *простого*, либо *составного поля*. Ключевое слово myltislot определяет, что слот может содержать значение составного поля, состоящее, возможно, из пустой последовательности полей. Ключевое слово slot или single-slot определяет, что слот может содержать только одно значение. Значениями составного слота разрешено манипулировать с помощью стандартных функций для составных полей, таких как nth$ и length$, начиная с момента присвоения им некоторого значения. COOL также предоставляет функции для установки составных слотов, например, slot-insert$. Простой слот может содержать любое значение примитивного типа CLIPS, такое как целое или строковое.

Грани значений по умолчанию

Грани default и default-dynamic применяются для задания начальных значений присваиваемых слотам при создании экземпляра класса или его инициализации. Если эта грань не использована при определении слота, то в качестве значения по умолчанию берется значение, заданное в грани ограничений слота (*см. подразд. "Грань ограничений" данного раздела*). Значения по умолчанию назначаются без использования сообщений, в отличие от переопределения слота при вызове make-instance *(см. разд. 11.5.1).*

Грань default задает *статическое значение по умолчанию* — некоторое выражение, вычисленное однажды, в момент создания класса. Полученный результат вычислений сохраняется в определении класса. Этот результат присваивается соответствующему слоту, когда создается новый экземпляр объекта. Если в качестве выражения указано ключевое слово ?DERIVE, то

значением по умолчанию будет значение, заданное в грани ограничений слота *(см. подразд. "Грань ограничений" этого раздела*). По умолчанию значение грани default равно (default ? DERIVE). Если в качестве выражения используется ключевое слово ?NONE, то слоту не будет присваиваться значение по умолчанию. В случае применения этого ключевого слова, при создании объекта в функции make-instance необходимо переопределить значения слотов. Заметьте, что в CLIPS 6.0 и более старших версиях слот имеет значение по умолчанию, даже если оно явно не задано (в отличие от CLIPS 5.1). Это может быть причиной различного поведения программ, написанных на CLIPS 5.1 и использующих функцию initialize-instance. Ключевое слово ?NONE можно указывать для обеспечения работы программ, написанных на CLIPS, начиная с версии 6.0, в среде CLIPS версии 5.1.

Грань default-dynamic задает *динамическое значение по умолчанию* — некоторое выражение, вычисляемое каждый раз при создании экземпляра объекта данного класса. Результат вычислений назначается соответствующему слоту. Работу грани default-dynamic можно продемонстрировать на примере 11.14.

Пример 11.14. Использование грани default-dynamic

```
(defclass A   (is-a USER)
              (role concrete)
              (slot foo (default-dynamic (gensym))
                        (create-accessor read)))
```

Выражение, заданное в грани default-dynamic — вызов системной функции gensym. Эта функция возвращает постоянно увеличивающийся системный идентификатор genX, где X — положительное число, увеличивающееся на единицу с каждым вызовом. Грань create-accessor read обеспечивает возможность чтения значения слота объекта. (Эта грань будет подробно рассмотрена в *подразд. "Грань акцессоров" данного раздела*.) Очистите CLIPS и создайте несколько объектов класса A так, как представлено в примере 11.15.

Пример 11.15. Создание нескольких объектов класса A

```
(make-instance al of A)
(make-instance a2 of A)
(make-instance a3 of A)
```

После этого просмотрите значения слотов foo всех экземпляров объектов класса A.

Пример 11.16. Просмотр содержимого слотов объектов класса A

```
(send [al] get-foo)
(send [a2] get-foo)
(send [a3] get-foo}
```

Полученный результат должен соответствовать результату, приведенному на рис. 11.3.

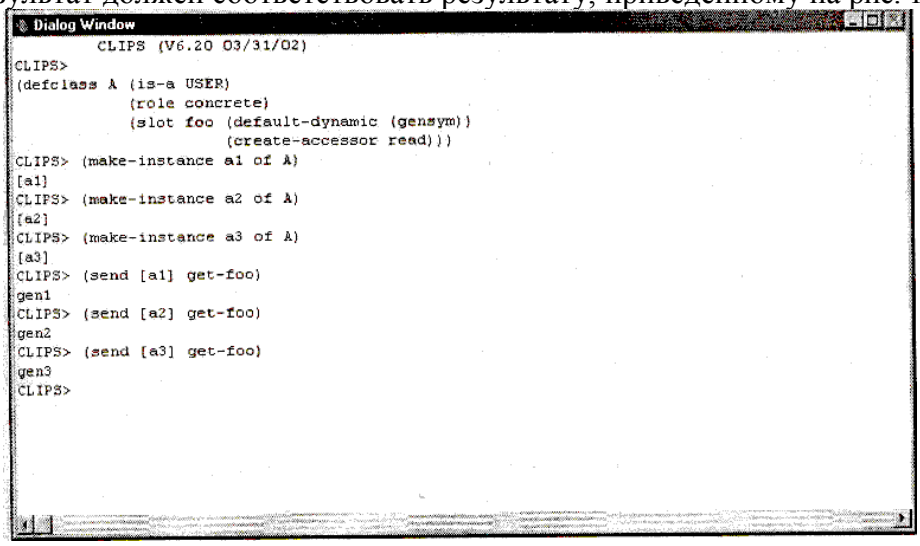

Рис. 11.3. Результат работы грани default-dynamic

Обратите внимание, что значения слотов всех объектов класса A, получивших значения по умолчанию при создании, отличаются друг от друга. Можете повторить выполнение этого примера с использованием грани default вместо default-dynamic и убедиться, что значения всех слотов будут одинаковыми.

Грань хранения

Реальное значение слота копии экземпляра может быть сохранено либо в экземпляре объекта, либо в классе. Грань local определяет, что значение слота будет сохранено с экземпляром. Это установка по умолчанию для грани хранения. Грань shared определяет, что значение сохраняется в классе. Если значение слота сохранено локально, то каждый класс может иметь свое собственное значение слота. Однако, если значение слота сохранено в классе, все экземпляры будут иметь одинаковое значение этого слота. Изменение значения общего слота изменит этот слот во всех экземплярах класса. Такое поведение делает общие слоты похожими на static-члены классов в языке программирования C++.

Если общий слот имеет динамическое значение по умолчанию, то при создании нового объекта данного класса вновь вычисленное значение слота по умолчанию присваивается слотам всех объектов данного класса. Общий слот игнорирует статическое значение по умолчанию, если уже существуют объекты данного класса, и значение данного слота отлично от значения по умолчанию. Если класс является активным, то любые изменения общего слота будут причиной проведения сопоставления образцов для обновления плана решения задачи.

Рассмотрим пример 11.17, демонстрирующий особенности поведения общих и локальных слотов объектов.

Пример 11.17. Использование граней хранения

```
(defclass A (is-a USER)
            (role concrete)
            (slot foo (create-accessor write)
                      (storage shared)
                      (default 1))
            (slot bar (create-accessor write)
                       (storage shared)
                       (default-dynamic 2) )
             (slot woz (create-accessor write)
                        (storage local)))
(make-instance a of A)
(send [a] print)
(send [a] put-foo 56)
(send [a] put-bar 104)
```

Результат этих действий приведен на рис. 11.4.

Выполните действия, представленные в примере 11.18. (Несмотря на то, что в данном примере используются еще нерассмотренные нами обработчики сообщений, он должен быть интуитивно понятен.) Результат — на рис. 11.5.

Пример 11.18. Просмотр содержимого общих и локальных слотов

```
(make-instance b of A)
(send   [b]   print)
(send   [b]   put-foo 34)
(send   [b]   put-woz  68)
(send   [a]   print)
(send   [b]   print)
```

Внимательно рассмотрите поведение общих слотов: при присвоении им значений или создании новых объектов, с динамическими и нединамическими значениями по умолчанию, для общих слотов.

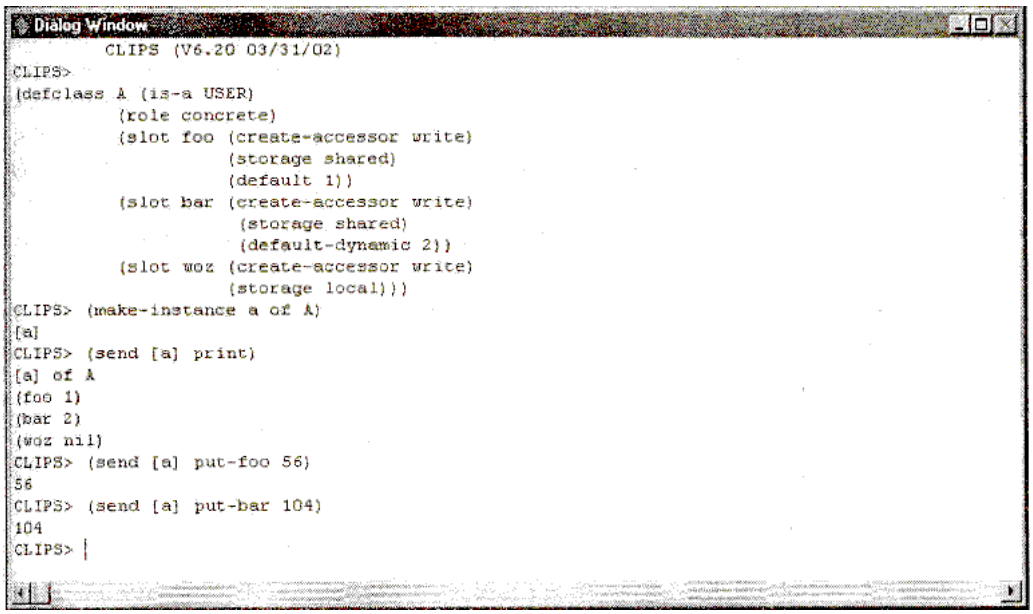

Рис. 11.4. Создание объекта с общими и локальными слотами и присвоение значений

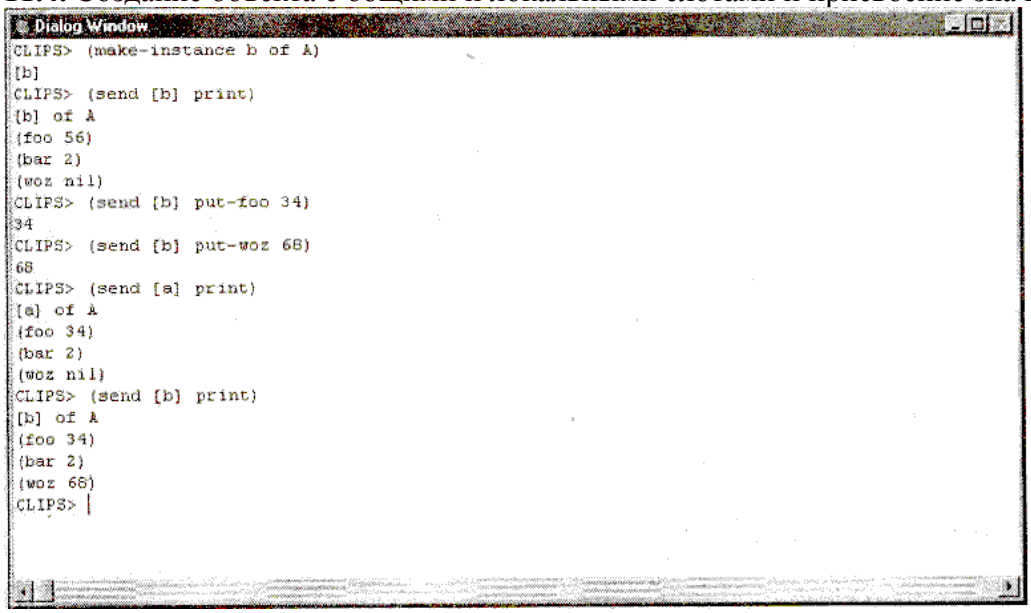

Рис. 11.5. Результат просмотра содержимого общих и локальных слотов

Грани доступа

Существуют три значения грани доступа access, которые могут быть присвоены слоту: read-write, read-only, initialize-only. Грань read-write устанавливается по умолчанию и объявляет, что значение слота можно читать и изменять. Грань read-only объявляет, что из слота можно только читать. Единственный способ установить значение этого слота — использование грань default в определении класса. Грань initialize-only аналогична грани read-only за исключением того, что слот также может быть установлен с помощью переопределения при вызове make-instance *(см разд. 11.5.1)* и в обработчике сообщений init *(см. подразд. "Инициализация объекта" разд. 11.3.3).* Заметьте, что слот read-only, который имеет статическое значение по умолчанию, будет неявно иметь общую грань хранения.

Создайте следующую версию класса A и обработчик сообщения для записи в слот bar.

Пример 11.19. Класс A с различными значениями грани доступа для слотов

```
(defclass A (is-a USER)
        (role concrete)
        (slot foo (create-accessor write)
                (access read-write))
        (slot bar (access read-only)
                (default abc))
        (slot woz (create-accessor write)
```

```
                (access initialize-only)))
(defmessage-handler A put-bar (?value)
                (dynamic-put (sym-cat bar) ?value))
```

Выполните действия, перечисленные в примере 11.20.

Пример 11.20. Создание экземпляра класса A и работа со слотами

```
(make-instance a of A (bar 34))
(make-instance a of A (foo 34) (woz 65))
(send [a] put-bar 1)
(send [a] put-woz 1)
(send [a] print)
```

Вы должны получить результат, приведенный на рис. 11.6.

Внимание!

Несмотря на то, что у нас получилось переопределить значение слота woz при создании объекта с помощью make-instance, изменить его с помощью стандартного сообщения put нельзя.

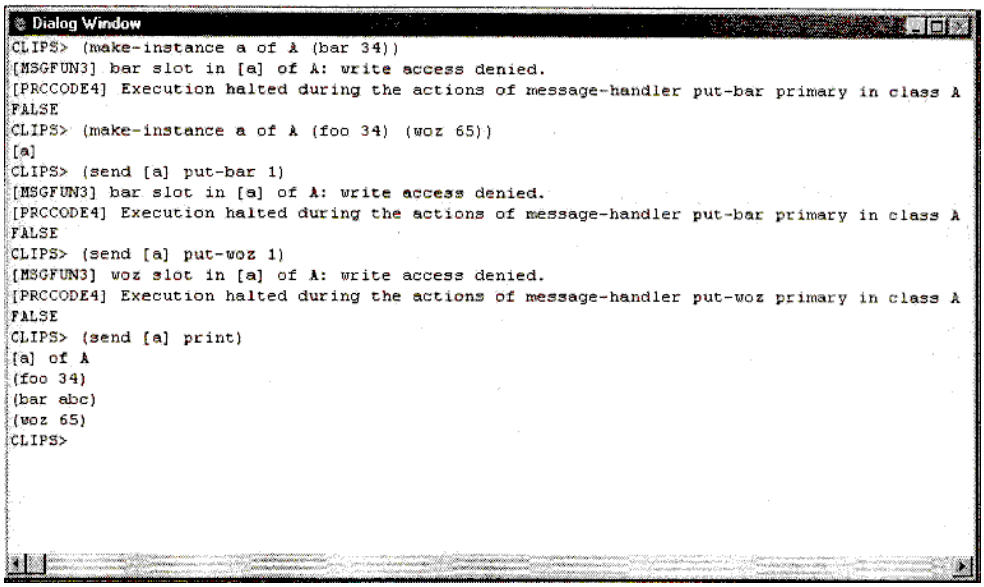

Рис. 11.6. Создание экземпляра класса A и работа со слотами

Грань распространения при наследовании

COOL позволяет регулировать процесс наследования слотов с помощью грани propagation. Она может принимать два значения: inherit и no-inherit. Значение inherit позволяет всем классам-наследникам получать и использовать данный слот. Это значение по умолчанию принимается для всех слотов. Значение no-inherit приводит к тому, что слот, обладающий такой гранью, не распространяется по наследованию и содержится только в классе, где он был определен.

Создадим два простых класса.

Пример 11.21. Слоты с различными гранями распространения при наследовании

```
(defclass A (is-a USER)
            (role concrete)
            (slot foo (propagation inherit))
            (slot bar (propagation no-inherit)))
(defclass B (is-a A))
```

Создайте экземпляры объектов обоих классов и просмотрите содержащиеся в них слоты помощью стандартного сообщения print. Результат этой операции представлен на рис. 11.7.

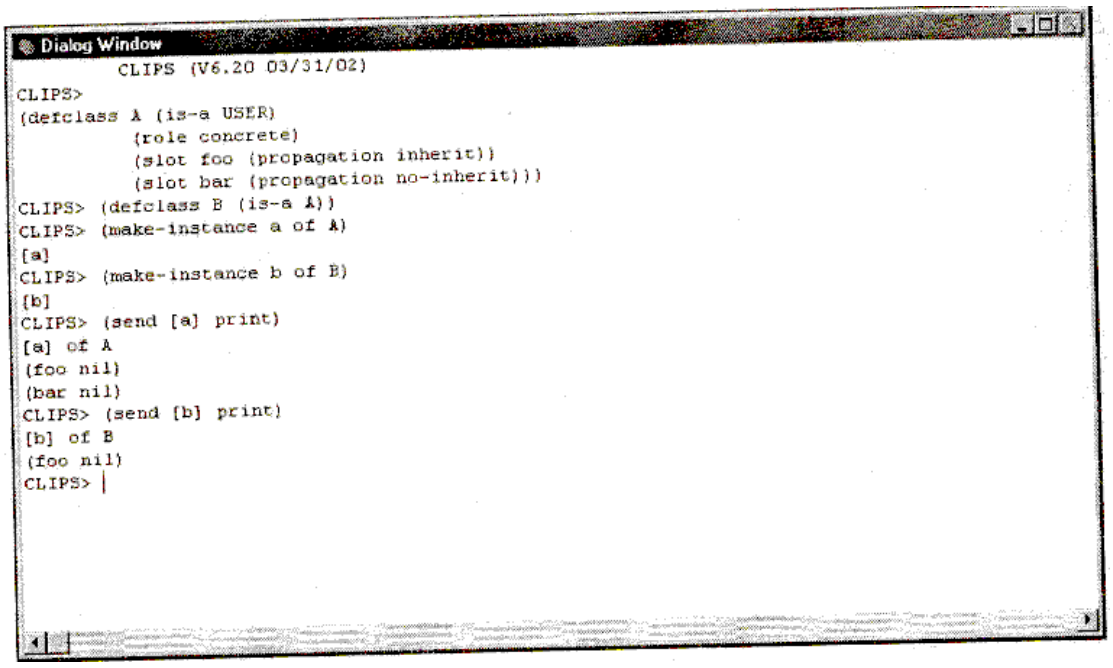

Рис. 11.7. Просмотр слотов классов A и B

Грань источника

В момент создания объект получает слоты, определенные в классах, находящихся в его списке наследования. При получении некоторого слота по умолчанию его свойства определяются гранями, заданными в наиболее определенном классе (находящемся левее в списке предшествования). Незаданные грани получают значения по умолчанию. Для изменения этого поведения служит грань source источника. Эта грань может принимать одно из двух значений: exclusive и composite. Значение exclusive реализует поведение по умолчанию. Если при создании слота указана грань source composite, то при создании слота неопределенные грани, не заданные в наиболее определенном классе, берутся из следующего, менее определенного, класса и т. д. Таким образом, в формировании свойств слотов могут участвовать несколько классов. Рассмотрим пример 11.22.

Пример 11.22. Использование грани источника

```
(defclass A (is-a USER)
            (role concrete)
            (slot foo (default A)))
(defclass B (is-a A)
            (slot foo))
(defclass C (is-a A))
            (slot foo (source composite)))
```

Создайте экземпляры объектов всех трех классов и просмотрите содержащиеся в них слоты и их значения с помощью стандартного сообщения print. Вы должны получить результат, идентичный приведенному на рис. 11.8.

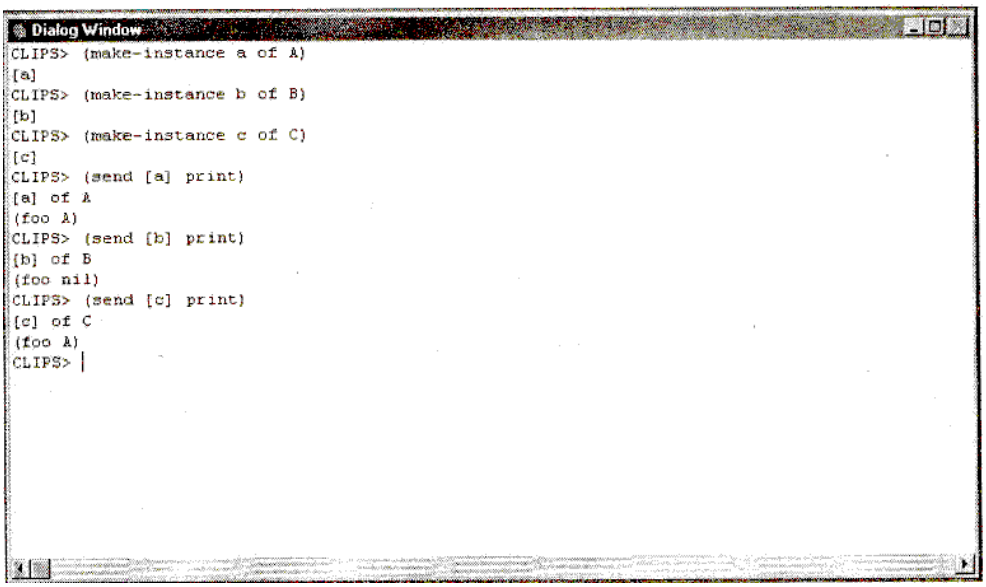

Рис. 11.8. Просмотр слотов классов A, B и C

В данном примере класс A определяет слот foo со значением по умолчанию для слота foo, равным A. Классы B и C являются наследниками класса A, и оба переопределяют слот foo. Однако, в отличие от класса B, класс C определяет для слота foo грань source со значением composite. Таким образом, слот foo класса C получает значение по умолчанию, определенное в классе-предке A.

Грань активности при сопоставлении образцов

Обычно любое изменение слота экземпляра объекта рассматривается как изменение с целью сопоставления образцов. Однако существует возможность указать, что изменения слота объекта не должно вызывать процесс сопоставления образцов. Для этой цели служит грань pattern-match. Значение reactive определяет, что изменения слота активизируют процесс сопоставления образцов. Эта установка принята по умолчанию. Значение non-reactive указывает, что изменения слота не приведут к активизации процесса сопоставления образцов. Рассмотрим пример 11.23. Создайте классы A и B.

Пример 11.23. Использование грани активности

```
(defclass A (is-a USER)
            (role concrete)
            (pattern-match reactive)
            (slot foo  (create-accessor write)
                       (pattern-match non-reactive)))
(defclass B (is-a USER)
            (role concrete)
            (pattern-match reactive)
            (slot foo  (create-accessor write)
                       (pattern-match reactive)))
```

После это добавьте следующие правила:

Пример 11.24. Правила, использующие объекты

```
(defrule Create
            ?ins<-(object (is-a A | B))
            =>
            (printout t "Create " (instance-name ?ins) crlf) )
(defrule Foo-Access
            ?ins<-(object (is-a A | B) (foo ?))
             =>
            (printout t "Foo-Access " (instance-name ?ins) crlf))
```

С помощью первого правила фактически будет получен список всех объектов классов A или B. Второе правило будет выводить имена объектов классов A или в, имеющих хоть какое-то значение слота foo (если этот слот может принимать участие в процессе сопоставления образцов). Создайте по одному экземпляру каждого класса и запустите программу. После этого измените значения слотов и попробуйте запустить программу еще раз. Вы должны получить результат, приведенный на рис. 11.9.

Обратите внимание, что только второе сообщение send вызовет изменение плана решения задачи (это хорошо видно с помощью окна **Agenda Window).** Это происходит потому, что слот foo класса A не является активным в процессе сопоставления образцов.

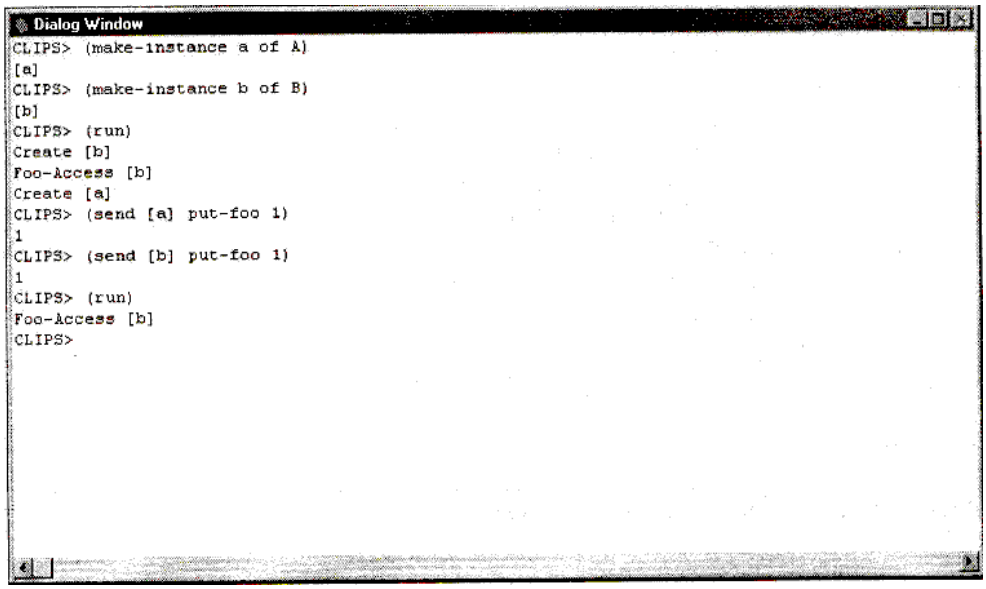

Рис. 11.9. Работа правил, использующих объекты

Грань видимости

Обработчики сообщений, присоединенные к некоторому классу, могут непосредственно оперировать со слотами, определенными в этом классе. Однако существует способ напрямую организовать манипуляцию со слотами для обработчиков, присоединенных к суперклассу или подклассу. Для этого предусмотрена грань visibility. Значение private указывает, что только обработчики сообщений определяемого класса могут иметь прямой доступ к слоту. Это значение задается по умолчанию. Объявление данной грани со значением public определяет, что обработчики сообщений подклассов, которые наследуют слот, а также суперклассы, тоже могут непосредственно обращаться к слоту.

Грань акцессоров

Акцессор слота — это особый обработчик сообщений, присоединенный к классу. Такой обработчик позволяет читать или изменять значение некоторого слота класса. Грань акцессора дает возможность управлять процессом автоматического создания подобных обработчиков. Эта грань уже использовалась в предыдущих примерах. Теперь рассмотрим ее более подробно.

В CLIPS 5.1 акцессор неявно создавался для всех слотов класса. Начиная с версии 6.0 ситуация иная. Пользователь должен самостоятельно определить необходимые ему акцессоры для чтения или записи некоторых слотов. Это удобно, потому что в большинстве случаев акцессоры не используются, т. к. все обработчики могут напрямую манипулировать со значениями слотов класса. Грань create-accessor сообщает CLIPS, что он должен автоматически создать явные обработчики сообщений для чтения и/или записи слота. По умолчанию акцессоры не создаются.

Созданные CLIPS акцессоры являются обычными обработчиками сообщений, ими можно манипулировать обычном образом. Однако, в отличие от обработчиков, созданных пользователем, они не имеют печатной формы (т. е. пользователь не может посмотреть код их реализации), и их нельзя удалить или изменить.

Если грань create-accessor получает значение read, то CLIPS автоматически создаст следующий обработчик:

Определение 11.4. Акцессор чтения

```
(defmessage-handler <class> get-<slot-name> primary   ( )
                    ?self:<slot-name>)
```

Если для этой грани определено значение write, CLIPS создаст обработчик сообщений для простого слота:

Определение 11.5. Акцессор записи простого слота

```
(defmessage-handler <class> put-<slot-name> primary (?value)
                    (bind ?self:<slot-name> ?value)
```

Или обработчик для составного слота:

Определение 11.6. Акцессор записи составного слота

```
(defmessage-handler <class> put-<slot-name> primary   ($?value)
                    (bind  ?self:<slot-name>  ?value)
```

Если грань create-accessor получит значение read-write, создаются оба обработчика сообщений: get- и один из put-.

Нежелательно создавать акцессоры для статических слотов. В этом случае лучше явно создать необходимые обработчики с помощью defmassage-handler.

Рассмотрим простой пример. Создайте следующий класс.

Пример 11.25. Использование грани акцессора

```
(defclass A (is-a USER)
            (role concrete)
            (slot foo (create-accessor write))
            (slot bar (create-accessor read)))
```

В классе A содержатся слоты foo и bar, причем для слота foo будет автоматически создан акцессор для записи нового значения в слот, а для слота bar — акцессор для чтения. Попробуйте создать объект класса A с инициализацией слота foo, а затем инициализацией слота bar. Вы должны получить результат, приведенный на рис. 11.10.

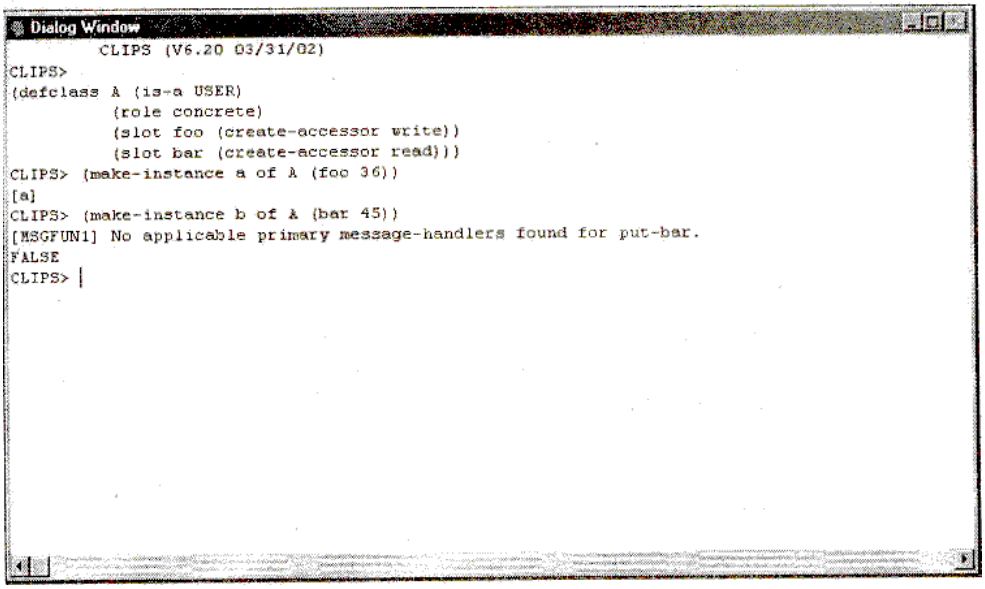

Рис. 11.10. Использование акцессоров

Как видно из рисунка, создание объекта b класса A с попыткой проинициализировать слот bar значением 45 не увенчалась успехом. CLIPS выдал сообщение об ошибке, в котором говорится,

что CLIPS не нашел подходящего обработчика сообщений для инициализации слота bar. Другие примеры использования грани create-accessor и соответствующих обработчиков сообщений вы уже встречали в этой главе.

Грань переопределения сообщений

Некоторые функции CLIPS устанавливают значения слотов объектов с помощью сообщений, например: make-instance, initialize-instance, message-modify-instance, message-duplicate-instance. По умолчанию все эти функции пытаются установить значение слота с помощью вызова сообщения put-<имя слота>. Если пользователь не задал автоматическое создание акцессоров с помощью грани create-accessor, но хочет, чтобы эти функции работали и могли переопределять значения слотов, то необходимо использовать специальную грань override-massage. Эта грань позволяет задавать имя сообщения, которое будет послано экземпляру объекта некоторого класса, при попытке изменить значение данного слота. Грань override-massage можно также использовать, если стандартные акцессоры существуют, но их применение приведенными выше функциями нежелательно.

В качестве иллюстрации использования этой грани рассмотрим класс из примера 11.26.

Пример 11.26. Использование грани переопределения сообщений

```
(defclass A (is-a USER)
        (role concrete)
        (slot special (create-accessor read-write)
                (override-message special-put)))
```

Класс A содержит слот special, который с помощью грани override-massage сообщает среде CLIPS, что для изменения слота special необходимо использовать сообщение special-put, определенное пользователем, а не стандартное сообщение, созданное с помощью грани create-accessor с именем put-special. Определим обработчик сообщения special-put следующим образом (более подробно создание обработчиков сообщений будет рассмотрено в *разд. 11.3):*

Пример 11.27. Создание обработчика special-put

```
(defmessage-handler A special-put primary (?value)
                (printout t "special-put " ?value crlf)
                (bind ?self:special ?value))
```

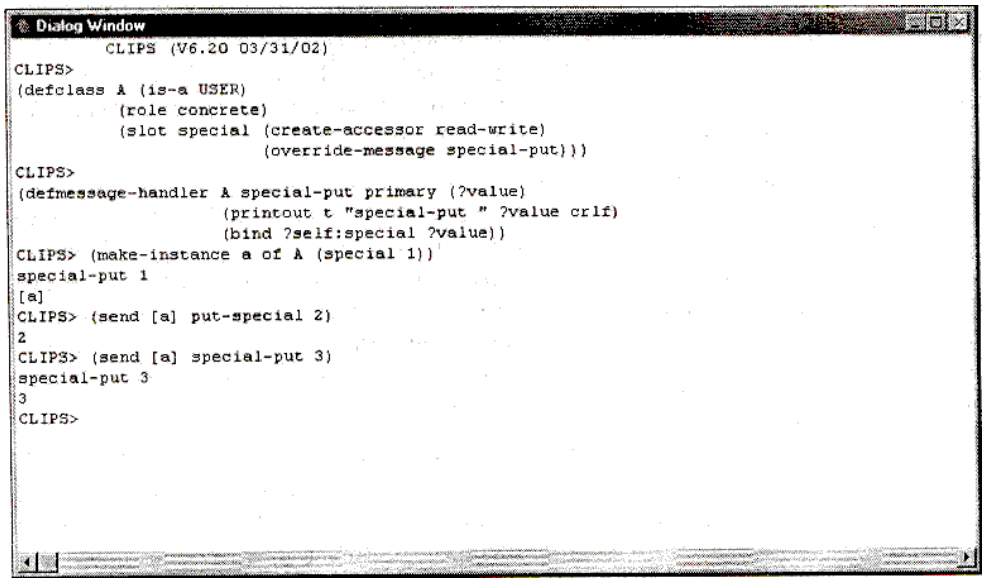

Рис. 11.11. Использование обработчиков put-special и special-put

Данный обработчик просто выводит на экран имя сообщения и новое значение слота special.

Выполним последовательность действий, представленную в примере 11.28.

Пример 11.28. Использование обработчиков put-special и special-put

```
(make-instance a of A (special 1))
(send [a] put-special 2)
(send [a] special-put 3)
```

Вы должны получить результат, идентичный приведенному на рис. 11.11. Легко заметить, что несмотря на то, что для данного класса A существует стандартный акцессор, функция make-instance для инициализации слота special использует обработчик special-put.

Грань ограничений

Синтаксис и функциональность ограничений для значений простых и составных слотов, полей и переменных детально описаны в *гл. 13.* CLIPS поддерживает проверку статических и динамических ограничений для классов и объектов классов. Проверка статических ограничений осуществляется при выполнении конструктора или команды, определяющих значение слота. Кроме того, выполняется проверка объектов-образцов, используемых в левой части правил для определения наличия конфликтов между ограничениями переменными. В случае возникновения ошибки соответствующая информация сразу предоставляется пользователю. Режим статической проверки ограничений включен по умолчанию. Эту установку можно изменить с помощью функции set-static-constraint-checking.

Кроме статической проверки CLIPS поддерживает возможность динамической проверки ограничений слотов. Если этот режим включен, то значения слотов проверяются при каждом изменении, включая изменения с помощью стандартных акцессоров или обработчиков сообщений, определенных пользователем. По умолчанию этот режим выключен. Данную установку можно изменить с помощью функции set-dynamic-constraint-checking. Если нарушение ограничения происходит в момент выполнения программы, то выполнение будет завершено.

Помимо описанных выше функций для изменения состояния статической и динамической проверки ограничений, пользователям Windows-версии среды CLIPS доступен визуальный способ установки режимов проверки ограничений. Для этого необходимо открыть новое окно **Execution Options,** выбрав пункт **Options** в меню **Execution.** Внешний вид этого диалогового окна приведен на рис. 11.12. Для включения или отключения статической и динамической проверки ограничений установите в соответствующее положение флажки Static Constraint Checking и Dynamic Constraint Checking.

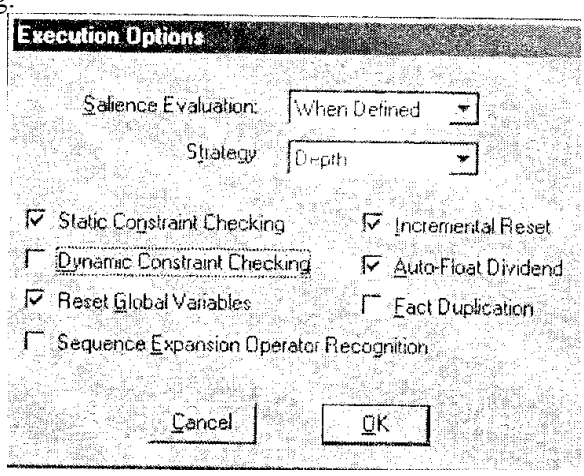

Рис. 11.12. Установка режимов проверки ограничений

Независимо от включенных режимов проверки ограничений, значения составных полей не могут быть сохранены в простые слоты. Значения простых слотов конвертируются в значения составных слотов с длиной, равной 1, при сохранении в простой слот значения составного поля. Кроме того, CLIPS не допускает использование функций, не имеющих возвращаемого значения для инициализации или изменения значения слота.

Объявление обработчиков сообщений

CLIPS позволяет задавать предварительное объявление обработчиков сообщений класса в конструкторе defclass. Эти объявления применяются только для документирования, в качестве дополнительных комментариев, и игнорируются CLIPS. Однако для обеспечения большей наглядности желательно использовать эту возможность.

Для реализации обработчиков сообщений класса необходимо применять конструктор defmessage-handler. Поскольку объявления обработчиков не являются обязательным элементом конструктора defclass, к классу могут быть добавлены обработчики, которые не были объявлены в defclass. Например, класс, представленный в примере 11.29, может содержать обработчики, приведенные в примере 11.30.

Пример 11.29. Объявление обработчика сообщений в определении класса

```
(defclass rectangle (is-a USER)
        (slot side-a (default 1))
        (slot side-b (default 1))
        (message-handler find-area))
```

Пример 11.30. Реальные обработчики сообщений класса rectangle

```
(defmessage-handler  rectangle  find-area   ( )
                    (*   ?self:side-a  ?self:side-b))
(defmessage-handler  rectangle print-area   ( )
                    (printout t   (send  ?self  find-area)   crlf))
```

11.3. Конструктор *defmessage-handler*

Манипулирование объектом происходит посредством передачи ему сообщений с помощью функции send. Результатом передачи сообщения могут быть либо вычисленное значение, либо некоторые действия обработчика. Конструктор defmessage-handler предназначен для создания обработчика сообщений, который фактически задает поведение объекта данного класса в ответ на получение определенного сообщения. Реализация сообщения представляет собой некоторую заданную последовательность действий, называемую *обработчиком сообщений* (или просто обработчиком).

Конструктор defmessage-handler состоит из семи элементов:

- имя класса, к которому добавляется обработчик (класс должен быть предварительно определен);
- имя сообщения, на которое будет откликаться обработчик;
- необязательный тип обработчика (по умолчанию primary);
- необязательные комментарии;
- список параметров, которые должны быть переданы обработчику в сообщении;
- необязательный символ групповых параметров для указания, что обработчик может иметь переменное число аргументов;
- последовательность действий, которые будут выполняться в заданном порядке в момент вызова обработчика.

Значение, возвращаемое обработчиком сообщения, является результатом вычислений последнего выражения в теле обработчика.

Определение 11.7. Синтаксис конструктора defmessage-handler

```
(defmessage-handler     <имя-класса>
                        <имя-сообщения>
                        [<тип-обработчика>]
                        [<комментарии>]
                        (<обязательные-параметры>
                        [<групповой-параметр>])
                        <действия>)
<тип-обработчика >          ::= around | before | primary | after
<обязательный-параметр>     ::= <простое-значение>
<групповой-параметр>        ::= <составное-значение>
```

Каждый класс из списка предшествования классов объекта может иметь свои обработчики для сообщений. В этом случае объект класса и все его суперклассы распределяют работу по обработке сообщений между собой. Каждый обработчик обрабатывает ту часть сообщения, которая

соответствует этому классу. Обработчики сообщений могут перекрываться обработчиками классов-потомков четырьмя способами: primary, before, after, around. Назначение этих типов обработчиков описано в табл. 11.1.

Таблица 11.1. *Типы обработчиков*

Тип обработчика	Роль обработчика
primary	Выполняет основную обработку сообщения
before	Выполняет вспомогательную обработку сообщения перед вызовом основного обработчика
after	Выполняет вспомогательную обработку сообщения после вызова основного обработчика
around	Подготавливает среду для выполнения остальных обработчиков

Обработчик primary является основным обработчиком сообщения. Он переопределяет все другие primary-обработчики того же сообщения, присоединенные к суперклассам данного класса. Обработчики before и after предназначены для выполнения некоторых побочных действий. Возвращаемое ими значение всегда игнорируется. Обработчик before выполняется перед выполнением основного обработчика; обработчик after — по окончании выполнения основного обработчика. Обработчики before и after предоставляют возможность изменять поведение классов родителей в объекте класса потомка. Как правило, полезное возвращаемое сообщением значение получают только от основного обработчика, хотя обработчик around также может вернуть полезное значение. Обработчик around предоставляет пользователю возможность как бы окружить дополнительным кодом код остальных обработчиков. Обработчики around начинают работу перед выполнением остальных обработчиков и продолжают свою работу после того, как все обработчики закончат свою работу.

Типы обработчиков сообщений, присоединенных к классу и к классам его родителей, определяют, какие обработчики будут запущены и в каком порядке. Такой подход называется *декларативным.* Однако иногда реализация сообщений может не соответствовать этой модели. Например, могут понадобиться результаты более чем одного основного обработчика. В подобных случаях обработчики сами должны решать, какие еще обработчики необходимо выполнить и в каком порядке. Такой подход называется *императивным.* Обработчики around предоставляют императивный контроль над обработчиками всех других типов за исключением обработчиков around, присоединенных к более определенному классу. Обработчики around способны менять окружение других обработчиков, запускать обработчики и модифицировать возвращаемое ими значение. Придерживайтесь декларативного подхода, насколько это возможно. Это позволит обработчикам быть более модульными и независимыми.

Обработчики сообщений однозначно идентифицируются классом, именем и типом. Обработчики сообщений никогда не вызываются непосредственно. Когда пользователь посылает сообщение объекту, CLIPS выбирает и упорядочивает применимые обработчики сообщений, присоединенные к объекту класса, и затем выполняет их. Этот процесс называется *связывание сообщений.* Для иллюстрации данного процесса рассмотрим пример 11.31.

Пример 11.31. Определение класса A и обработчиков сообщения delete

```
(defclass A (is-a USER)
            (role concrete))
(defmessage-handler A delete before ()
                    (printout t "Deleting an instance of the class A..."crlf))
(defmessage-handler USER delete after ()
                    (printout t "System completed deletion of an instance." crlf))
```

Класс A является прямым потомком класса USER и не имеет слотов. Кроме того, мы добавили два обработчика сообщения delete. Один обработчик типа before относится непосредственно к классу A, другой обработчик типа after относится к самому классу USER. Помимо этих двух обработчиков сообщения delete у каждого объекта потомка класса USER существует еще один обработчик primary, выполняющий удаление объекта (о системных обработчиках сообщений *см. в*

разд. 11.3.3). Для демонстрации процесса связывания сообщений включите режим просмотра изменения списка экземпляров объектов с помощью команды (watch instances) или диалогового окна **Watch Options,** открываемого через меню **Execution.** Создайте экземпляр объекта класса A и пошлите ему сообщение delete. Результат этой последовательности действий приведен на рис. 11.13.

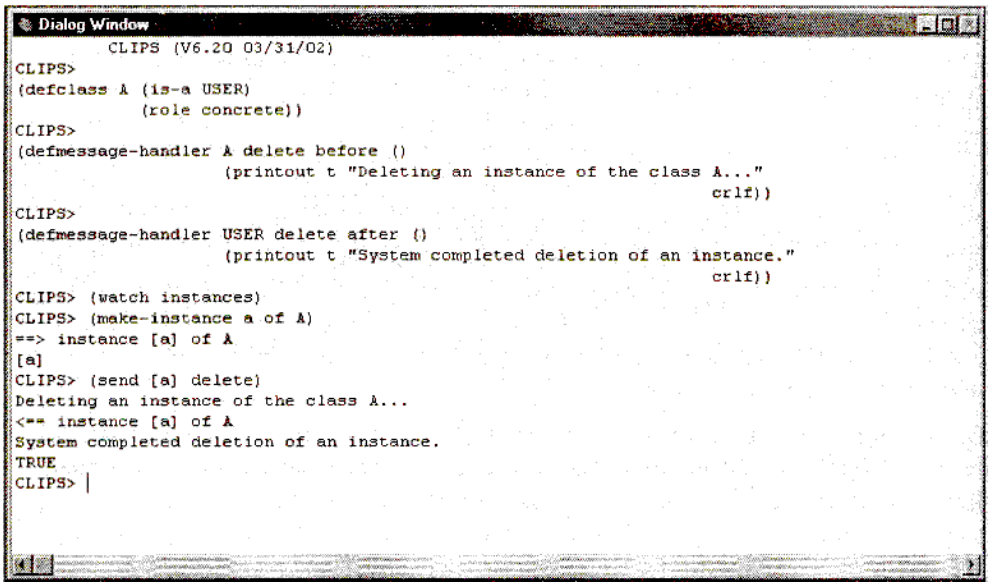

Рис. 11.13. Иллюстрация процесса связывания сообщений

Обратите внимание, что сначала был вызван обработчик before класса и, затем системный обработчик primary, который произвел удаления объекта, а после него обработчик after класса USER.

В данном разделе подробно был рассмотрен тип обработчиков сообщений. Остальные параметры конструктора defmessage-handler будут представлены ниже.

11.3.1. Параметры обработчиков сообщений

В зависимости от того, задан ли групповой параметр, созданный конструктором, обработчик может принимать точное число параметров или число параметров, не меньшее, чем некоторое заданное. Обязательные параметры определяют минимальное число аргументов, которые должны быть переданы обработчику. В действиях обработчика можно ссылаться на каждый из этих параметров так, как на обычные переменные, содержащие простые значения. Если был задан групповой параметр, то обработчик сообщения может принимать любое количество аргументов, большее или равное минимальному числу аргументов. Если групповой параметр не задан, то обработчик может принимать число аргументов, равное числу обязательных параметров. Все аргументы обработчика сообщения, которые не соответствуют обязательным параметрам, группируются в одно значение составного поля. Ссылаться на это значение можно, используя символ группового параметра. Для работы с групповым параметром могут применяться стандартные функции CLIPS, предназначенные для работы с составными полями, такие как length$ и nth$. Определение обработчика может содержать только один групповой параметр.

Параметры обработчика не имеют отношения к применимости обработчика к отдельным сообщениям. Однако, если число аргументов не соответствует необходимому, CLIPS сгенерирует сообщение об ошибке в момент вызова этого обработчика сообщения. Таким образом, принимаемое число аргументов должно быть совместимо со всеми обработчиками сообщений, применимых к некоторому сообщению.

Например, для класса CAR (пример 11.32) можно определить обработчик для инициализации слотов (пример 11.33).

Пример 11.32. Класс CAR

```
(defclass  CAR   (is-a USER)
           (role concrete)
```

```
(slot  front-seat)
(multislot  trunk)
(slot  trunk-count))
```

Пример 11.33. Инициализация объекта класса CAR

```
(defmessage-handler CAR put-items-in-car (?item $?rest)
                    (bind ?self:front-seat ?item)
                    (bind ?self:trunk ?rest)
                    (bind ?self:trunk-count (length$ ?rest)))
```

Термин *"активный экземпляр"* обозначает экземпляр объекта, который в данный момент обрабатывает сообщение. Все обработчики сообщений имеют неявный объявленный параметр ? self, ссылающийся на активный экземпляр. Это имя зарезервировано и не может явно включаться в список параметров обработчика сообщений. Кроме того, это имя не может быть переопределено в теле обработчика сообщений.

11.3.2. Действия обработчиков сообщений

Действия обработчика сообщений представляют собой последовательность выражений, которые выполняются в заданном порядке, при вызове обработчика. Значение, возвращаемое обработчиком — результат вычисления последнего выражения в теле обработчика.

Слотами объекта можно манипулировать, только используя сообщения-акцессоры объекта. Однако обработчики сообщений являются частью объекта, инкапсулирующего данные и методы их обработки. Таким образом, в действиях обработчика можно манипулировать слотами объекта непосредственно, минуя механизм сообщений. Кроме того, некоторые функции неявно оперируют активным экземпляром объекта (так же без использования сообщений). Применять эти функции можно только из обработчиков сообщений. Полный список таких функций приведен в *гл. 15*.

Для доступа к слотам активного экземпляра из действий обработчиков сообщений используется следующая конструкция:

Определение 11.8. Доступ к слоту из обработчика сообщений

?self:<имя-слота>

Приведем пример обработчика сообщений, напрямую манипулирующего со слотами объекта, и определим класс и один обработчик.

Пример 11.34. Использование слотов в обработчиках сообщений класса

```
(defclass A (is-a USER)
            (role concrete)
            (slot foo (default 1))
            (slot bar (default 2)))
(defmessage-handler A print-all-slots ()
                     (printout t ?self:foo " " ?self:bar crlf))
```

Обработчик сообщения print-all-slots выводит на экран содержимое слотов foo и bar. Причем значения, хранящиеся в этих слотах, передаются напрямую, а не с помощью сообщений.

Помимо получения текущего значения обработчики также могут устанавливать новые значения слотов. Для этого используется функция bind.

Определение 11.9. Установка значений слотов в обработчиках сообщений класса

(bind ?self:<имя-слота> <значение>*)

Например, для класса A из предыдущего примера можно определить обработчик, приведенный в примере 11.35.

Пример 11.35. Установка значений слотов в обработчиках сообщений класса

```
(defmessage-handler A set-foo (?value)
                    (bind ?self:foo ?value))
```

При определении обработчиков прямой доступ к слоту статически связывается с соответствующим слотом. Поэтому особого внимания требуют ситуации, при которых прямой доступ к слоту выполняется с помощью посылки сообщения объекту, являющемуся экземпляром подкласса класса, с которым было связано данное сообщение. Если данный подкласс переопределяет слот, к которому осуществляет прямой доступ обработчик сообщения суперкласса, CLIPS остановит выполнение и выведет сообщение об ошибке. Ошибка в данном случае происходит из-за того, что обработчик сообщения суперкласса статически связан со слотом суперкласса, который переопределил предок. Для демонстрации возникновения данной ошибки приведем пример 11.36.

Пример 11.36. Переопределение слотов

```
(defclass A   (is-a USER)
          (slot foo   (create-accessor read)))
(defclass B   (is-a A)
          (role concrete)
          (slot  foo   (create-accessor write)))
```

В данном примере класс A определяет слот foo и стандартный акцессор для чтения значения этого слота. Класс B является потомком класса A, переопределяет слот foo и не переопределяет акцессор чтения. Поэтому посылка объекту класса B сообщения get-foo приведет к возникновению ошибки. Данная ситуация продемонстрирована на рис. 11.14.

Для решения подобной проблемы необходимо предпринять следующие шаги. Во-первых, переопределенные в классе наследнике слоты нужно определить с гранью visibility public. Во-вторых, обработчики, осуществляющие доступ к слотам, которые могут быть переопределены, должны использовать функции dynamic-put и dynamic-get. Приведенный выше пример можно переписать следующим образом:

Пример 11.37. Правильное переопределение слотов

```
(defclass A   (is-a USER)
          (slot  foo   (create-accessor write))
(defmessage-handler A get-foo   ( )
                    (dynamic-get   foo))
(defclass  B   (is-a A)
          (role concrete)
          (slot  foo   (visibility public)
                    (create-accessor write))
```

Посылка сообщения get-foo экземпляру класса B, созданного таким образом, не приведет к возникновению ошибки (рис. 11.15).

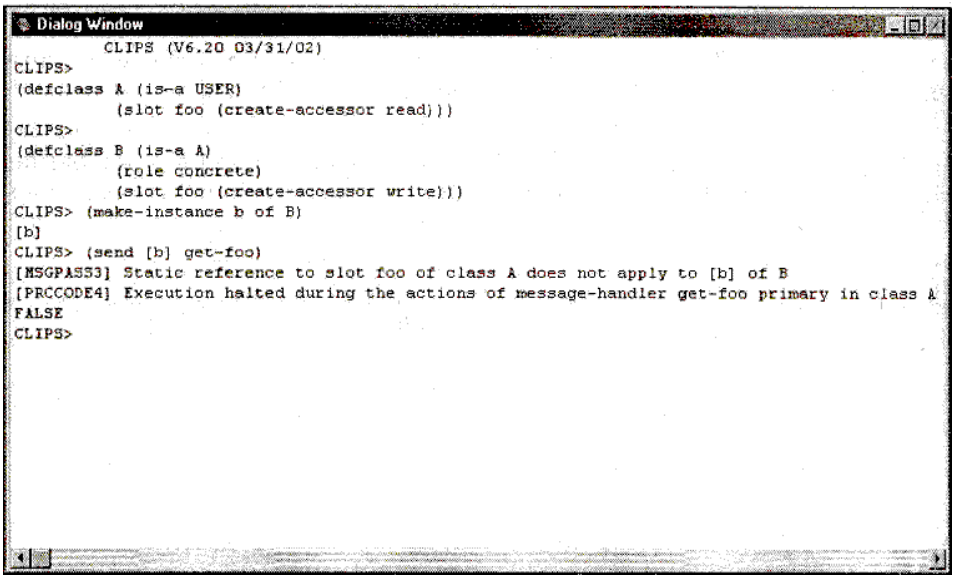

Рис. 11.14. Ошибка переопределения слотов

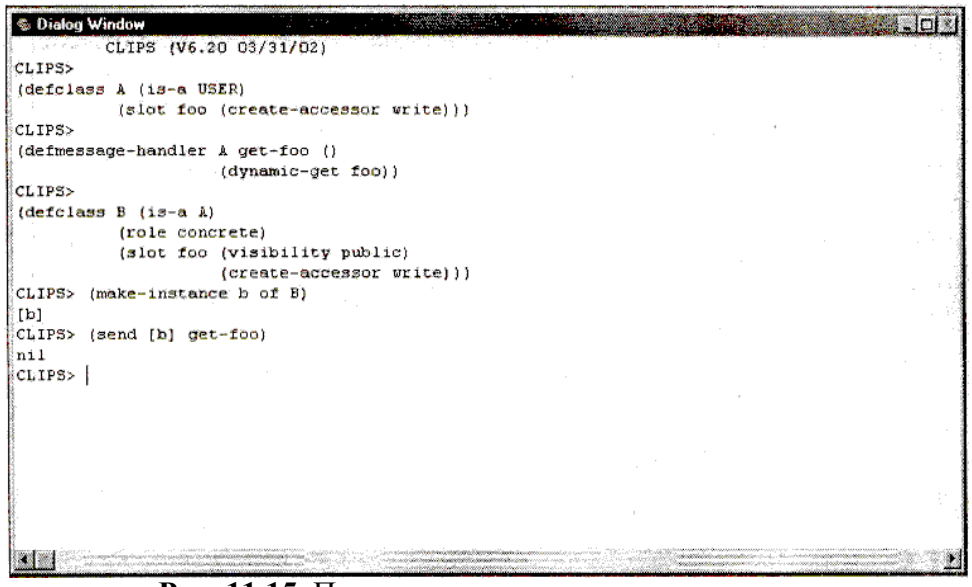

Рис. 11.15. Правильное переопределение слотов

11.3.3. Системные обработчики сообщений

Системный класс USER, предоставляемый CLIPS, содержит 7 предопределенных обработчиков сообщений: init, delete, print, direct-modify, message-modify, direct-duplicate и message-duplicate. Все классы, производные от класса USER, наследуют эти обработчики сообщений. (Функциональность указанных обработчиков будет рассмотрена далее.) Предопределенные системные обработчики сообщений не могут быть удалены или изменены.

CLIPS поддерживает понятие *"демоны"* (daemons). Демоны — это участки программного кода, неявно выполняющиеся всякий раз при возникновении некоторого предопределенного системного сообщения. На практике демоны реализуются с помощью обработчиков системных сообщений before или after, добавленных пользователем. Рассмотрим пример 11.38.

Пример 11.38. Создание демона системного обработчика init

```
(defclass  A   (is-a USER)
                (role  concrete))
(defmessage-handler A init before   ()
                        (printout  t   "Initializing a new instance  of class A..."  crlf))
```

Здесь к классу A присоединен демон-обработчик before, который выполняется всякий раз перед инициализацией объекта (т. е. получения им системного сообщения init). Результат — на рис. 11.16.

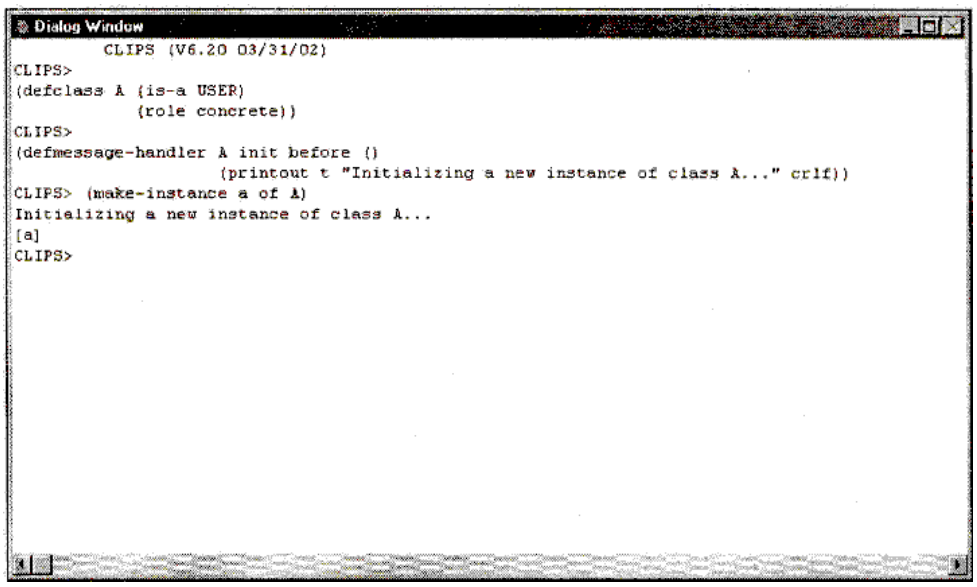

Рис. 11.16. Создание демона системного обработчика init

Инициализация объекта

При создании любой объект, который является потомком класса USER, получает сообщение init, обрабатываемое предопределенным системным обработчиком.

Определение 11.10. Синтаксис системного обработчика init

(defmessage-handler USER init primary ( ))

Данный обработчик отвечает за инициализацию объекта значением по умолчанию сразу после его создания. Сообщение init объекту посылают функции make-instance и initialize-instance. Пользователь не должен самостоятельно посылать это сообщение. Обработчик данного сообщения использует функцию init-slot. Пользователь может определить свой обработчик сообщения init, но определенный пользователем обработчик должен вызывать системный обработчик init. Рассмотрим пример 11.39.

Пример 11.39. Класс CAR

```
(defclass CAR (is-a USER)
        (role concrete)
        (slot price (default 75000))
        (slot model (default Corniche)))
```

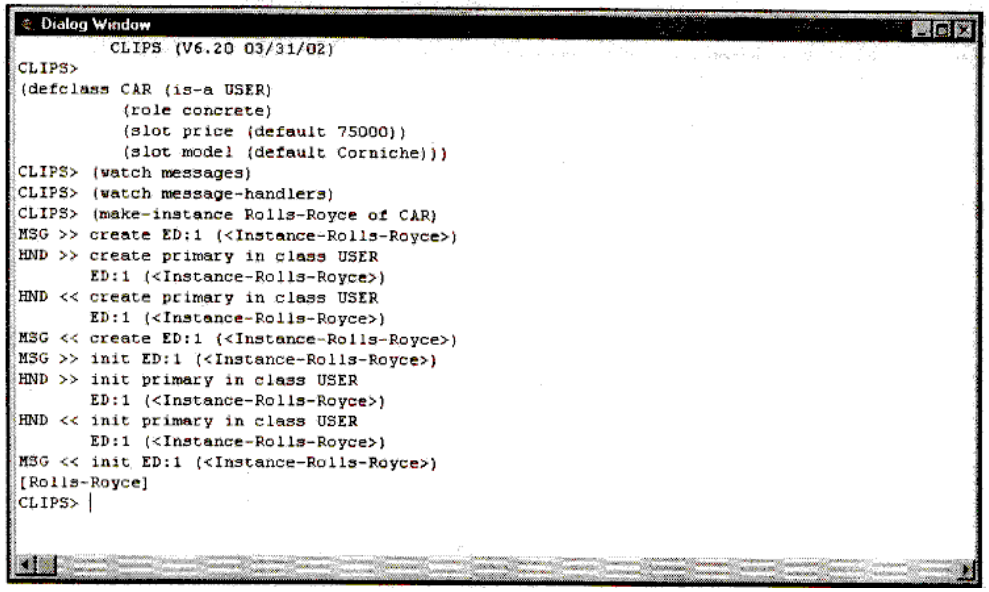

Рис. 11.17. Создание и инициализация экземпляра класса CAR

Включите просмотр поступающих сообщений с помощью команды (watch messages) и просмотр активизированных обработчиков командой (watch message-handlers) и создайте после этого какой-нибудь объект класса CAR. Вы должны увидеть результат, идентичный приведенному на рис. 11.17. Как видно из рисунка, сразу после создания объекту Rools-Royce класса CAR были посланы сообщения create и init.

Удаление объекта

При удалении объект класса, унаследованного от системного класса USER, получает сообщение delete, которое обрабатывается предопределенным системным обработчиком.

Определение 11.11. Синтаксис системного обработчика delete

```
(defmessage-handler USER delete primary  ( ) )
```

Этот обработчик отвечает за удаление объекта из системы. Пользователь должен самостоятельно послать сообщение delete объекту, который он хочет удалить. Обработчик возвращает значение TRUE, если объект успешно удален, в противном случае — FALSE. Пользователь может определить свой обработчик сообщения delete. Но определенный пользователем обработчик должен вызывать системный обработчик. Пример использования сообщения delete приведен на рис. 11.18.

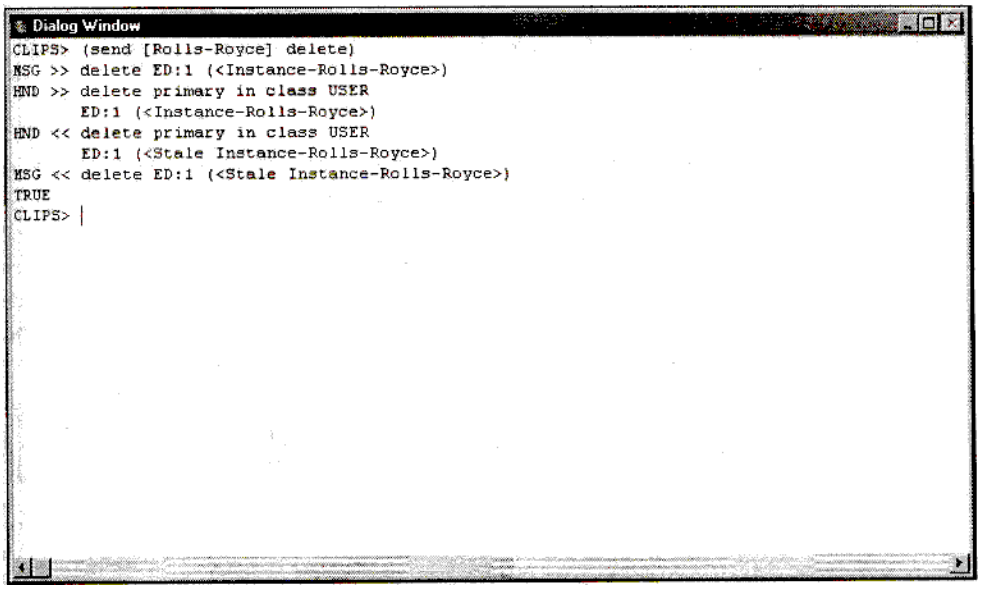

Рис. 11.18. Удаление экземпляра класса CAR

Отображение объекта

Для отображения содержимого слотов объекта предназначено сообщение print, которое также имеет предопределенный системный обработчик.

Определение 11.12. Синтаксис системного обработчика print

```
(defmessage-handler USER print Primary ())
```

Данный обработчик выводит название объекта, его класс и текущие значения всех слотов. Пример использования сообщения print приведен на рис.11.19.

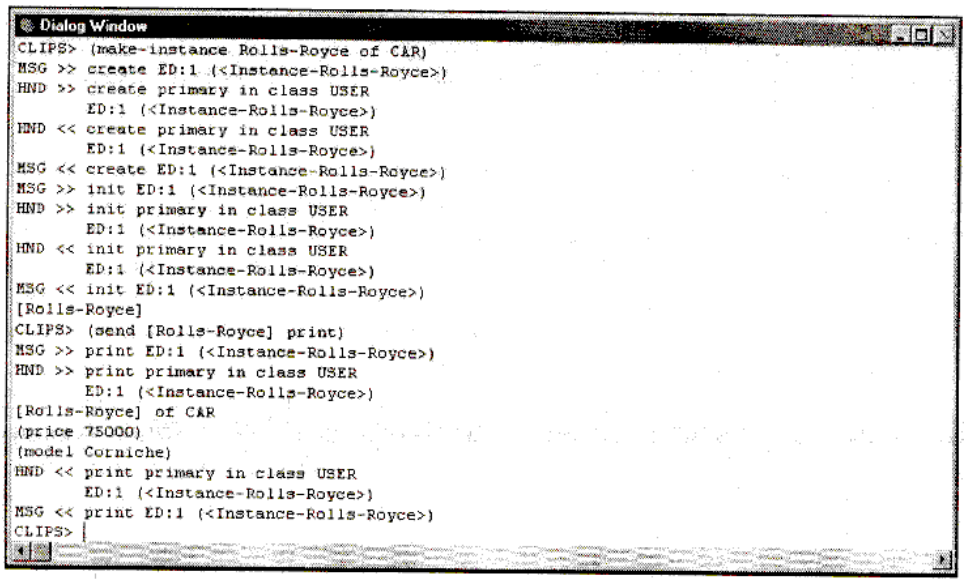

Рис.11.19. Использование системного обработчика print

Изменение объекта

CLIPS предоставляет два предопределенных системных обработчика для изменения значений слотов объекта: direct-modify и message-modify.

Определение 11.13. Синтаксис системных обработчиков direct-modify и message-modify

```
(defmessage-handler USER direct-modify primary
                    (?slot-override-expressions))
(defmessage-handler USER message-modify primary
 (?slot-override-expressions)
```

Обработчик direct-modify позволяет изменять значения слотов объекта без использования сообщений put-. Этот обработчик применяется функциями modify-instance и active-modify-instance.

Обработчик message-modify изменяет значения слотов объекта с помощью посылки сообщений put- для каждого слота и используется функциями message-modify-instance и active-message-modify-instance.

Копирование объекта

Так же, как и для изменения, CLIPS предоставляет два предопределенных системных обработчика для создания копии объекта: direct-duplicate и message-duplicate.

Определение 11.14. Синтаксис системных обработчиков direct-duplicate и message-duplicate

```
(defmessage-handler USER direct-duplicate primary
                    (?new-instance-name  ?slot-override-expressions))
(defmessage-handler USER message-duplicate primary
                    (?new-instance-name   ?slot-override-expressions)
```

Обработчик direct-duplicate производит копирование объекта без использования сообщений put- для установки значения заданных слотов. Значения слота из первичного объекта непосредственно копируются в заданные слоты нового объекта. Если имя нового объекта совпадает с уже существующим именем объекта, то существующий объект удаляется без применения сообщений. Это сообщение используется функциями duplicate-instance и active-duplicate-instance.

Обработчик message-duplicate копирует объект, используя сообщения. Значения слота из первичного объекта копируются в заданные слоты нового объекта с применением сообщений put- и get-. Если имя нового объекта совпадает с уже существующим именем объекта, то существующий

объект удаляется с помощью сообщения delete. После создания нового объекта ему посылается сообщение init. Это сообщение используется функциями message-duplicate-instance и active-message-duplicate-instance.

11.4. Диспетчеризация сообщений

При посылке сообщения объекту с помощью функции send CLIPS просматривает список предшествования класса для класса активного объекта и определяет набор обработчиков сообщений, которые применимы к данному сообщению. CLIPS использует роли (around, before, primary и after), заданные при определении этих обработчиков, для определения порядка, в котором данные сообщения должны быть выполнены, и после этого выполняет все обработчики в установленном порядке. Обработчик, связанный с классом, называется *более определенным,* чем обработчик того же сообщения, связанный с суперклассом. Весь этот процесс в целом называется *диспетчеризацией сообщений.*

Процесс диспетчеризации сообщений немного напоминает процесс родового связывания, описанный в предыдущей главе. Однако он имеет и некоторые существенные отличия, которые будут рассмотрены в данном разделе.

Обработчик сообщения считается применимым к сообщению, если его имя совпадает с сообщением, и он связан с классом, который находится в списке предшествования классов для объекта, получившего сообщения.

Множество всех применимых обработчиков сообщений сортируется по четырем группам, соответствующим ролям, и эти четыре группы дополнительно сортируются по определенности класса. Обработчики around, before и primary упорядочиваются от более определенного класса к более общему, тогда как обработчики after упорядочиваются от более общего к более определенному. Принцип выполнения последовательности обработчиков следующий:

- обработчики around начинают выполнение от более определенного к более общему (каждый обработчик around должен явно позволять выполнение других обработчиков);
- обработчики before выполняются последовательно от более определенного к более общему;
- обработчики primary начинают выполнение от более определенного к более общему (более определенные обработчики primary должны явно позволять выполнение более общих обработчиков);
- обработчики primary заканчивают выполнение от более общих к более определенным;
- обработчики after выполняются (друг за другом) от более общего к более определенному;
- обработчики around завершают выполнение от более общего к более определенному.

Должен существовать, по крайней мере, один применимый обработчик primary для сообщения. Иначе будет сгенерировано сообщение об ошибке.

Если один обработчик в процессе выполнения вызывает другой обработчик, то второй обработчик называется *скрытым* первым. Обработчик around скрывает все обработчики за исключением более определенных обработчиков around. Обработчик primary скрывает все более общие обработчики primary.

Обработка сообщения должна быть реализована с использованием декларативной технологии, насколько это возможно. Только роли обработчиков должны определять, какие обработчики получат выполнение. Для этого по возможности используйте только обработчики before и after и наиболее определенные обработчики primary. Это позволяет каждому обработчику сообщения быть полностью независимым от других обработчиков сообщений. Однако если необходимо использование обработчиков around или скрытых обработчиков primary, то такие обработчики должны брать на себя выполнение части процесса диспетчеризации сообщений, вызывая скрытые обработчики. Как уже упоминалось, такой подход называется *императивным*. Для выполнения скрытых обработчиков служат функции call-next-handler и override-next-handier. С их помощью обработчик может вызвать некоторый скрытый обработчик один или несколько раз.

Для иллюстрации последовательности вызовов скрытых обработчиков рассмотрим последовательность обработчиков сообщений.

Пример 11.40. Обработчики сообщений my-message классов USER и OBJECT

```
(defmessage-handler USER my-message  around  ( )
```

```
                    (call-next-handler))
(defmessage-handler USER my-message before ())
(defmessage-handler USER my-message ()
                    (call-next-handler))
(defmessage-handler USER my-message after ())
(defmessage-handler OBJECT my-message around ()
                    (call-next-handler))
(defmessage-handler OBJECT my-message before ())
(defmessage-handler OBJECT my-message ())
(defmessage-handler OBJECT my-message after ())
```

Диаграмма, приведенная на рис. 11.20, иллюстрирует порядок выполнения обработчиков, связанных с классами USER и OBJECT, при посылке сообщения my-message объекту класса наследника USER. Скобки показывают место начала и конца работы отдельного обработчика. Обработчики, находящиеся внутри скобок другого обработчика, являются скрытыми.

Если во время выполнения обработчика происходит ошибка, любой текущий выполняемый обработчик прерывается, обработчики, которые еще не начали выполнение, игнорируются, и функция send возвращает значение FALSE.

Если для пословного сообщения нет применимого обработчика primary или при вызове обработчика было передано неверное число параметров, CLIPS сгенерирует сообщение об ошибке и прервет выполнение программы.

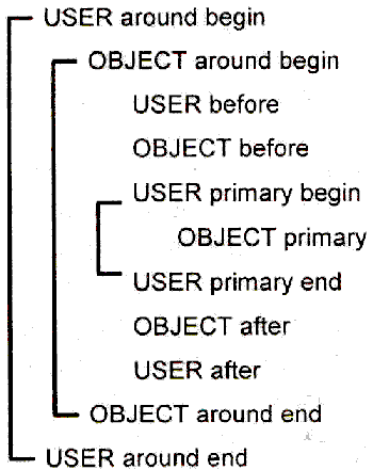

Рис.11.20. Последовательность выполнения обработчиков сообщений my-message, связанных с классами USER и OBJECT

Возвращаемое функцией send значение — это значение, полученное от наиболее определенного обработчика around или наиболее определенного обработчика primary, если обработчика around нет. Значение, возвращаемое обработчиком, является результатом вычисления последнего действия в этом обработчике.

Значения, возвращаемые обработчиками before и after, игнорируются. Эти обработчики применяются только для выполнения дополнительных полезных действий. Обработчик around может выбирать игнорировать или использовать значения, полученные от менее определенных обработчиков around или primary. Обработчик primary может выбирать игнорировать или использовать значения, полученные от менее определенных обработчиков primary.

11.5. Работа с объектами

Поскольку CLIPS реализует концепцию инкапсуляции данных, работа с объектами происходит посредством посылки им сообщений. Для этого существует системная функции send, которая содержит в качестве параметра объект назначения для сообщения, само сообщение и любые аргументы, которые передаются обработчику.

Определение 11.15. Синтаксис функции send

(send <объект> <имя-сообщения> [<аргументы>])

Существует два важных исключения, в которых работа с объектом выполняется без использования сообщений и функции send:

- объекты некоторых системных классов (например, вещественные и целые числа, символы, строки, составные значения, адреса фактов, внешние адреса) могут обрабатываться стандартным образом без использования сообщений;
- создание и инициализация объекта определенного пользователем класса выполняется с помощью системной функции make-instance.

В следующих разделах данной главы будут подробно рассмотрены типичные приемы работы с объектами, такие как создание и удаление объекта, работа со слотами объектов, копирование объектов и т. д.

11.5.1. Создание объекта

Так же, как и факты, объекты определенных пользователем классов должны быть явно созданы пользователем. Как и факты, все объекты, созданные пользователем, удаляются командой reset. Они могут быть загружены или сохранены в текстовый файл. Все операции, использующие объекты, выполняют передачу сообщений, применяя функцию send, за исключением операции создания объекта, если объект еще не существует.

Для создания и инициализации нового объекта служит функция make-instance. Она неявно посылает инициализирующие сообщения каждому новому объекту после его размещения в памяти. При желании пользователь может влиять на процесс инициализации с помощью обработчика-демона. Функция make-instance также позволяет переопределять значения слотов для изменения любого значения по умолчанию. Она автоматически приостанавливает процесс сопоставления образцов в правилах для всех активных объектов до тех пор, пока процесс создания объекта не будет завершен. Функция active-make-instance также позволяет создавать новые объекты определенных пользователем классов, но не вызывает задержки процесса сопоставления образцов.

Определение 11.16. Синтаксис функций make-instance и active-make-instance

```
(make-instance                <определение-объекта>)
(active-make-instance         <определение-объекта>)
<определение-объекта>         ::= [<имя-объекта>] of
                                  <имя-класса>
                                  <переопределения-слотов>
<переопределение-слота>       ::= (<имя слота> <значение>)
```

В случае успеха функция make-instance возвращает имя только что созданного объекта, в противном случае возвращает значение FALSE. Определение имени объекта является необязательным. Если имя объекта не задано, CLIPS воспользуется функцией gensym* для автоматической генерации имени.

В процессе выполнения функция make-instance совершает следующие действия:

1.Если объект с заданным именем уже имеется, то существующий объект получает сообщение delete, Т.е. (send <имя-объекта> delete). Если по какой-то причине эта операция завершается неудачно, создание нового объекта прерывается.

2.Создается новый неинициализированный объект заданного класса с заданным именем.

3.Значения всех переопределяемых слотов немедленно вычисляются и устанавливаются с помощью сообщения put-, т.е. (send <имя-объекта> put-<имя-слота> <выражение>*). Если в процессе этой операции возникнет какая-либо ошибка, новый объект будет удален.

4. Новый объект получает сообщение init: (send <имя-объекта> init). Предопределенный системный обработчик этого сообщения вызывает функцию init-siots. Данная функция использует значения по умолчанию из определения класса (если они есть) для всех слотов, которые не были переопределены. Установки слотов класса по умолчанию помещаются непосредственно в слот без использования сообщений. Если на этом этапе возникает ошибка, новый объект также удаляется.

Для иллюстрации работы описанного алгоритма определите в среде CLIPS класс и обработчики сообщения, представленные в примере 11.41.

Пример 11.41. Класс A и обработчики сообщений put-x и delete

```
(defclass A   (is-a USER)
              (role concrete)
              (slot x  (default 34)
                       (create-accessor write))
              (slot y (default abc)))
(defmessage-handler A put-x before (?value)
                          (printout t "Slot x set with message." crlf) )
(defmessage-handler A delete after ()
(printout t "Old instance deleted. " crlf))
```

После этого попробуйте создать объект с помощью функции make-instance, удалить его и создать объект с переопределением значения для некоторого слота. Обратите внимание на содержимое слотов объекта и выводимые на экран сообщения. Пример подобных операций с объектом приведен на рис. 11.21

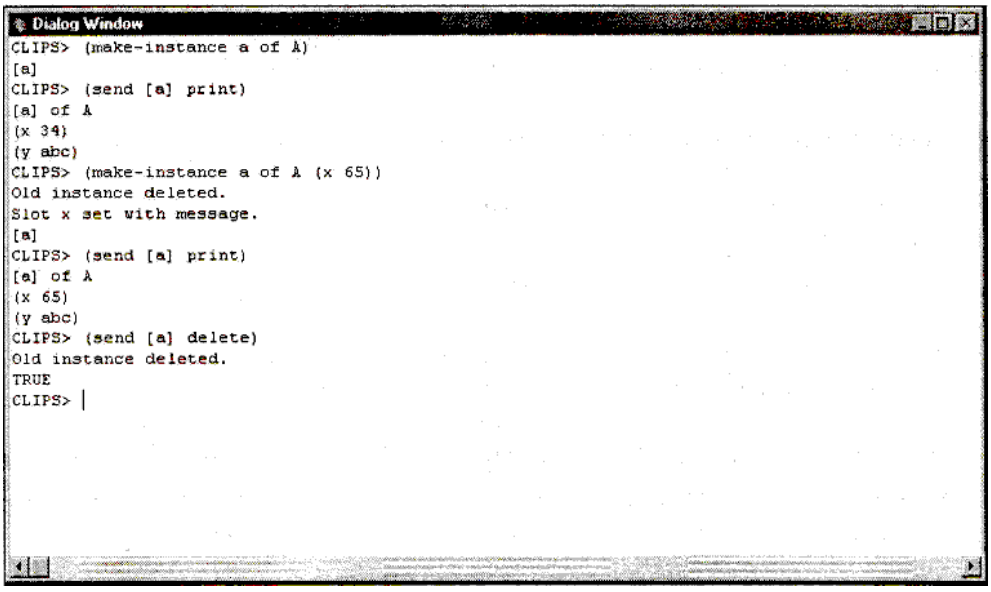

Рис. 11.21. Использование команды make-instance

Конструктор *definstances*

Подобно конструктору deffacts, конструктор definstances позволяет определять объекты, которые будут создаваться каждый раз при выполнении команды reset. При выполнении reset все текущие объекты получают сообщение delete, после чего CLIPS производит вызовы функции make-instance для каждого объекта, определенного в конструкторе definstances.

Определение 11.17. Синтаксис конструктора definstances

```
(definstances          <имя>  [active]  [<комментарии>]
                       <шаблоны-объектов>)
<шаблон-объекта>::=  (<определение-объекта>)
```

Конструктор definstances не может использовать классы, которые еще не были определены. Объекты, заданные в definstances, создаются по порядку, и если создание какого-либо объекта заканчивается неудачно, оставшиеся объекты не будут созданы. Кроме того, будет прервано выполнение других конструкторов definstances, если они существуют. Обычно для создания объектов definstances используют функцию make-instance (что вызывает задержку текущего процесса сопоставления образцов). Однако, если это не желательно, можно написать ключевое слово active после имени конструктора definstances, что укажет конструктору на необходимость воспользоваться функцией active-make-instance.

Создайте конструкторы, представленные в примере 11.42.

Пример 11.42. Использование конструктора definstances

```
(defclass A (is-a USER)
            (role concrete)
            (slot x (create-accessor write)
                    (default 1) ) )
(definstances A-OBJECTS
                  (a1 of A)
                  (of A (x 65)))
```

После этого включите режим отображения объектов с помощью команды (watch instances) и выполните команду (reset). Результат этих действий приведен на рис. 11.22.

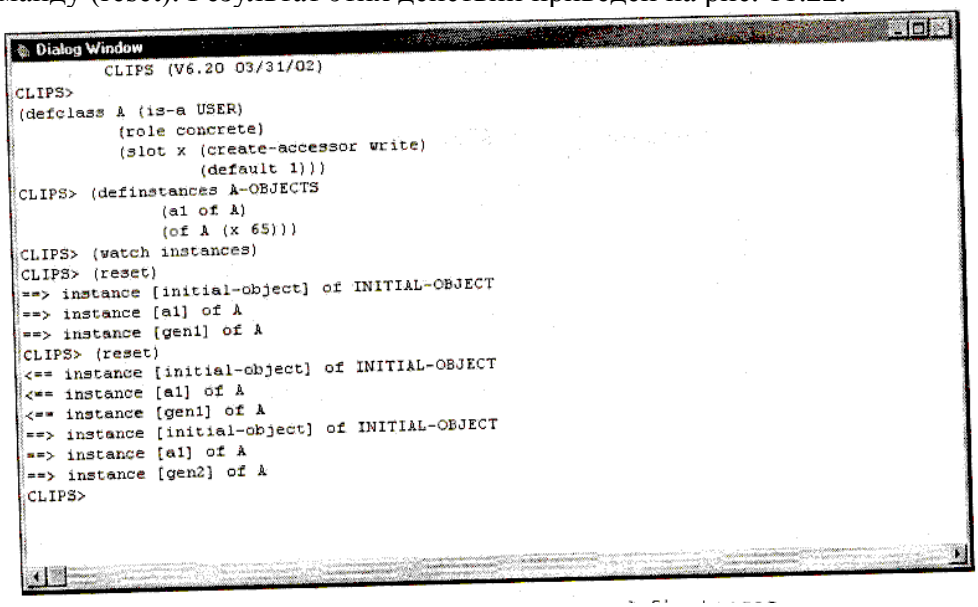

Рис. 11.22. Использование конструктора definstances

Во время загрузки и после выполнения команды clear CLIPS автоматически добавляет предопределенные конструкторы.

Определение 11.18. Синтаксис предопределенного класса и экземпляра этого класса

```
(defclass      INITIAL-OBJECT
               (is-a  USER)
               (role concrete)
               (pattern-match reactive))
(definstances initial-object
               (initial-object of INITIAL-OBJECT))
```

Класс INITIAL-OBJECT является предопределенным системным классом, прямым наследником класса USER. Класс INITIAL-OBJECT не может быть удален, однако может быть удален initial-object — объект этого класса. В предыдущих главах уже были подробно описаны значение и способы применения объекта initial-object в правилах.

Так же как и для конструктора deffacts, CLIPS предоставляет визуальный инструмент для манипуляции с определенными в данный момент в системе конструкторами definstances — **Definstances Manager** (Менеджер предопределенных объектов). Для его запуска выберите пункт **Definstances Manager** в меню **Browse.** Внешний вид окна менеджера предопределенных объектов приведен на рис. 11.23.

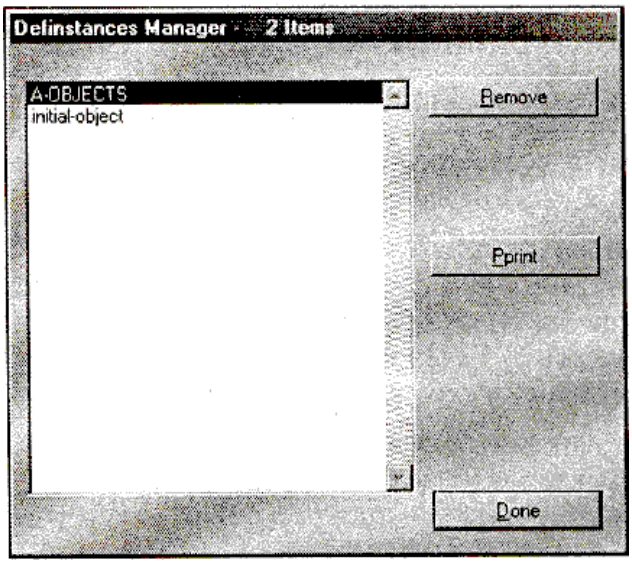

Рис. 11.23. Окно менеджера предопределенных объектов

Менеджер отображает все добавленные на текущий момент конструкторы definstances. В нашем случае это initial-object и только что добавленный нами A-OBJECTS. Менеджер позволяет выводить в основное окно CLIPS информацию об определениях, существующих в данный момент в системе конструкторов definstances с помощью кнопки **Pprint** (кроме definstances initial-object), и удалять любой существующий конструктор посредством кнопки **Remove.**

11.5.2. Переинициализация существующих объектов

Функция initialize-instance предоставляет возможность повторной инициализации существующих объектов значениями по умолчанию, заданными в определении класса, или новыми переопределениями слотов. Возвращаемое значение функции — имя объекта в случае успешного выполнения операции или значение FALSE в случаи ошибки. Параметр <объект> должен быть именем объекта, адресом объекта или строкой. Функция initialize-instance автоматически приостанавливает процесс сопоставления всех активных объектов правил до тех пор, пока операция не будет завершена. Если подобное поведение нежелательно, используйте функцию active-initialize-instance.

Определение 11.19. Синтаксис функции initialize-instance

(initialize-instance <объект> <переопределение-слота>*)

Функция initialize-instance в процессе выполнения совершает следующие действия:

1.Вычисляются все заданные переопределения слотов, после чего результаты помещаются в соответствующие слоты с помощью сообщений put- (send <объект> put- <имя слота> <значение>).

2.Предопределенный системный обработчик этого сообщения вызывает функцию init-slots. Данная функция использует значения по умолчанию из определения класса (если они есть) для всех слотов, которые не были переопределены. Установки слотов класса по умолчанию помещаются непосредственно в слот без использования сообщений.

Если при вызове функции не заданы переопределения ни для одного слота или в определении класса не задано ни одно значение по умолчанию, значение слотов объекта останется прежним. Пустые значения по умолчанию (nil) позволяют очищать слоты объекта с помощью вызова функции initialize-instance.

Если при выполнении функции происходит ошибка, объект не удаляется, но значения слотов могут стать несогласованными.

В качестве демонстрации работы функции initialize-instance создадим класс, представленный в примере 11.43.

Пример 11.43. Класс A

```
(defclass A (is-a USER)
        (role concrete)
        (slot x (default 34)
                (create-accessor write))
        (slot y (default nil)
                (create-accessor write))
        (slot z (create-accessor write) ))
```

Выполните следующую последовательность действий:

Пример 11.44. Использование функции initialize-instance

```
(make-instance  a  of A   (y abc)   (x  65))
(send   [a]   put-z   "Hello world.")
(send   [a]   print)
(initialize-instance  a)
(send   [a]   print)
```

Результат выполнения рассмотренных выше команд приведен на рис. 11.24.

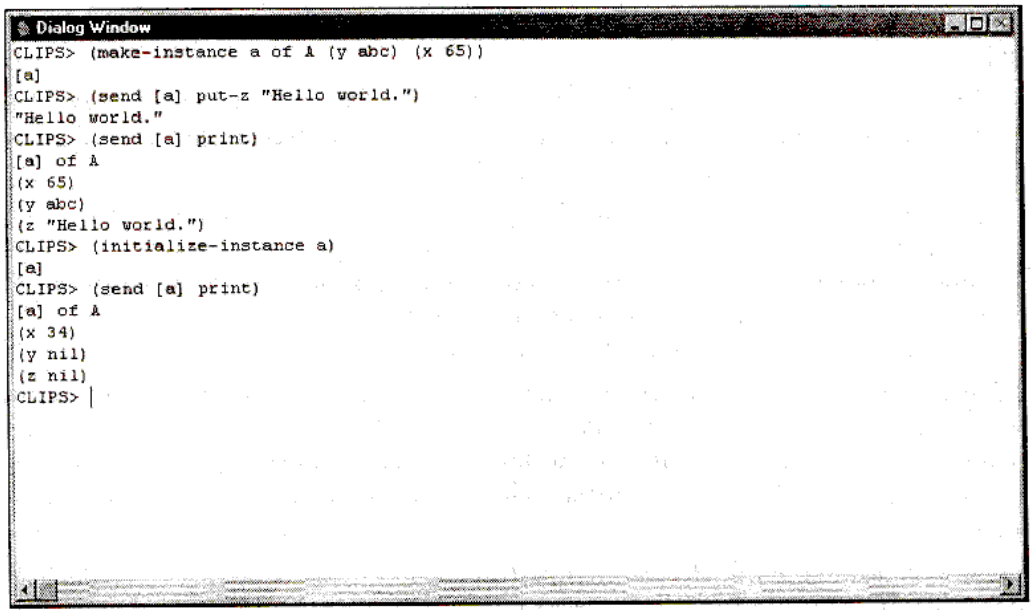

Рис. 11.24. Использование функции initialize-instance

11.5.3. Чтение значений слотов

Код, являющийся внешним по отношению к объекту, такой как правила или функции, может читать значения слотов объекта только при помощи сообщений. Обработчики сообщений, являясь частью определения класса, для чтения содержимого слотов могут использовать как сообщения, так и прямой доступ к слотам объекта. CLIPS предоставляет несколько функций, которые также могут неявно оперировать с объектом при помощи сообщений и могут быть вызваны только обработчиками, например, такими как dynamic-get. В *гл. 15* приводится описание способов проверки существования слотов объекта и их значений. Пример использования сообщений для чтения содержимого слотов объекта приведен на рис. 11.25.

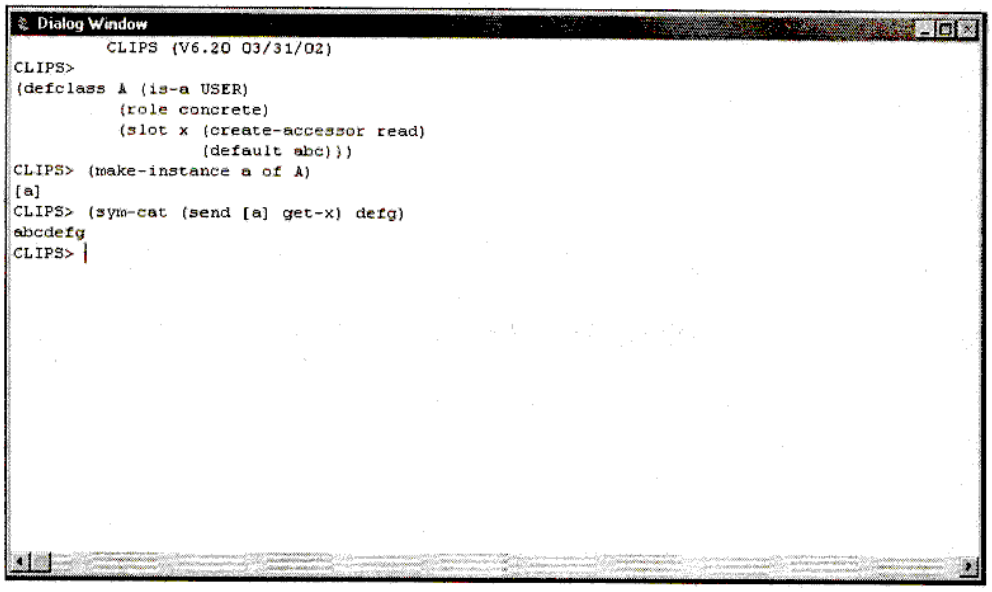

Рис. 11.25. Чтение значений слотов

11.5.4. Установка значений слотов

Правила, функции и любые другие конструкции CLIPS, которые являются внешним кодом по отношению к объекту, способны записывать новые значения слотов объекта только при помощи сообщений. Обработчики сообщений могут использовать для этих целей как сообщения, так и прямой доступ к слотам объекта. В случае прямого доступа к слоту для установки нового значения служит функция bind *(см. разд. 11.4.2)*. CLIPS предоставляет несколько функций, которые могут неявно оперировать объектом при помощи сообщений, но такие функции могут быть вызваны только обработчиками сообщений. Пример использования сообщений для записи нового значения в слот объекта приведен на рис. 11.26.

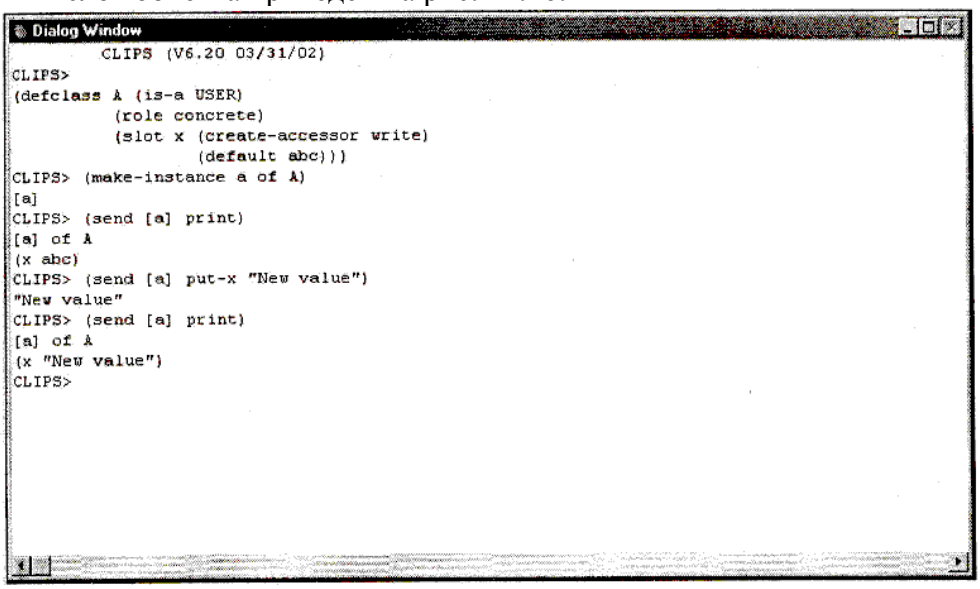

Рис. 11.26. Установка значений слотов

11.5.5. Удаление объектов

С помощью сообщения delete объект удаляется из системы. В обработчике сообщения может быть использована функция delete-instance *(см. гл. 15)* для удаления активного объекта.

Определение 11.20. Удаление объекта

(send <объект> delete)

Примеры использования этого сообщения уже несколько раз приводились ранее в данной главе.

11.5.6. Задержка сопоставления образцов при работе с объектами

При работе с объектами (например, создании, изменении или удалении объектов) возможна задержка процесса сопоставления образцов, выполняемая для правил, до конца выполнения операции. Такая задержка может быть произведена искусственно и при выполнении других действий, с помощью функции object-pattern-match-delay, которая действует идентично функции prong *(см. гл. 15)*. Однако процесс сопоставления образцов приостанавливается до завершения выполнения всех действий, заданных в функции object-pattern-match-delay. Основное назначение этой функции — предоставить контроль над выполнением процесса сопоставления образцов при осуществлении некоторых действий.

Определение 11.21. Синтаксис функции object-pattern-match-delay

(object-pattern-match-delay <действие>*)

Для демонстрации работы данной функции определите класс и правило, указанные в примере 11.21.

Пример 11.45. Определение класса A и правила, использующего объекты класса A

```
(defclass A (is-a USER)
          (role concrete)
          (pattern-match reactive)
(defrule match-A
          (object (is-a A))
          => )
```

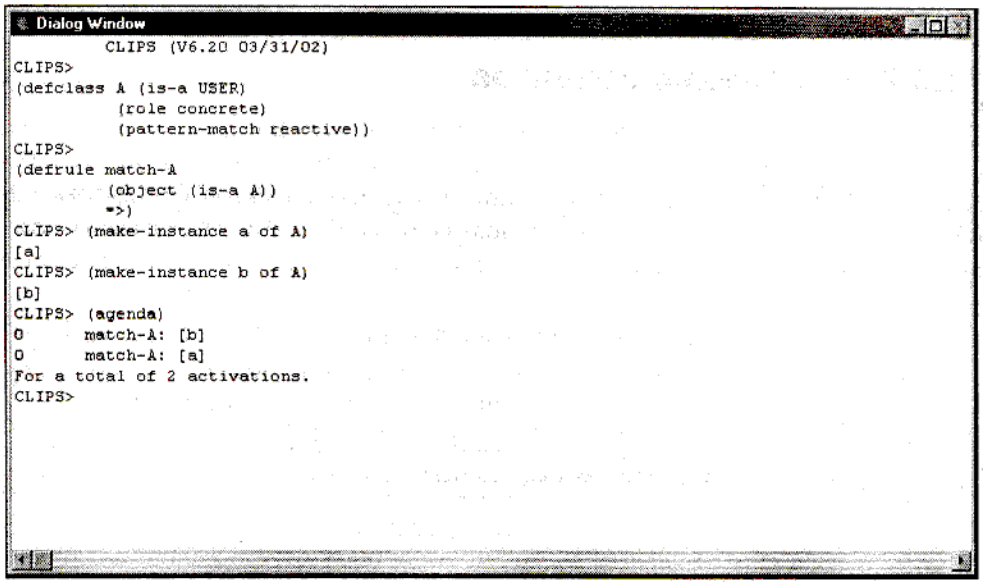

Рис. 11.27. Создание объектов и просмотр плана решения задачи

Создайте два объекта класса A и просмотрите план решения задачи, как показано на рис. 11.27. Дальнейшие действия представлены на рис. 11.28.

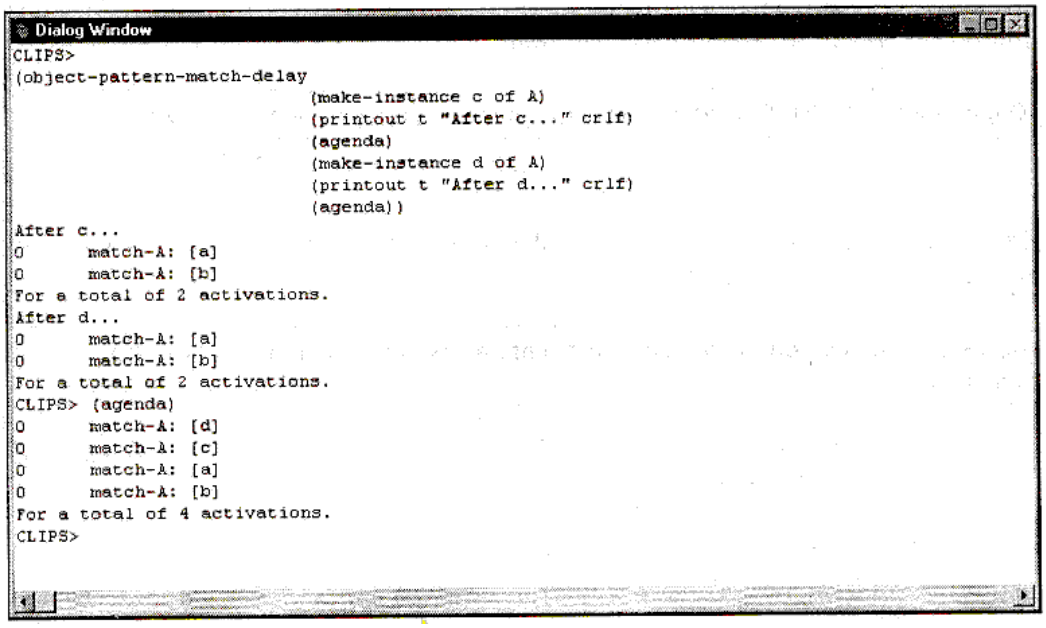

Рис. 11.28. Использование функции object-pattern-match-delay

11.5.7. Изменение объектов

Для изменения объектов предназначены четыре функции: modify-instance, active-modify-instance, message-modify-instance и active-message-modify-instance. Первые две позволяют обновлять слоты объектов без вызова сообщений put-, оставшиеся две используют сообщения. Каждая из этих функций возвращает значение TRUE в случае успеха и FALSE — в случае неудачи.

Функция modify-instance использует сообщение direct-modify для изменения значений слотов объекта. Процесс сопоставления образцов приостанавливается до тех пор, пока все изменения слотов не будут выполнены.

Определение 11.22. Синтаксис функции modify-instance

(modify-instance <объект> <переопределения-слота>*}

Пример использования функции modify-instance приведен на рис. 11.29.

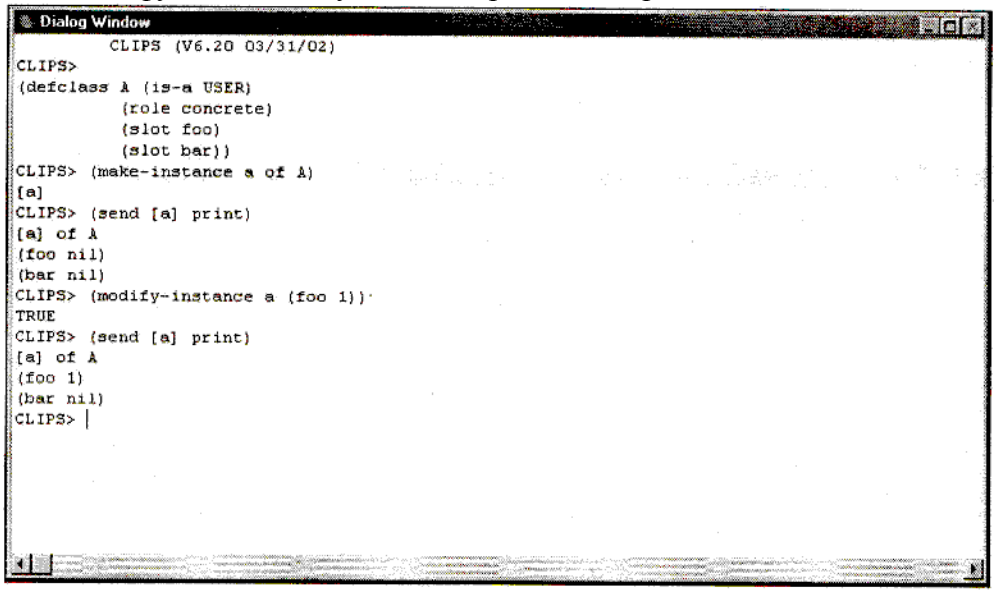

Рис. 11.29. Использование функции modify-instance

Функция active-modify-instance также работает с сообщением direct-modify для изменения значений слотов объекта. Однако при использовании этой функции процесс сопоставления образцов продолжается во время выполнения изменения слотов.

Определение 11.23. Синтаксис функции active-modify-instance

(active-modify-instance <объект> <переопределения-слота>*)

Функция message-modify-instance использует сообщение message-modify для изменения значений слотов объекта. Процесс сопоставления образцов приостанавливается до тех пор, пока все изменения слотов не будут выполнены.

Определение 11.24. Синтаксис функции message-modify-instance

(message-modify-instance <объект> <переопределения-слота>*)

Если остановка процесса сопоставления образцов нежелательна, можно применять функцию active-message-modify-instance, которая так же использует сообщение message-modify для изменения значений слотов объекта. Однако приостановка процесса сопоставления образцов при этом не происходит.

Определение 11.25. Синтаксис функции active-message-modify-instance

(active-message-modify-instance <объект> <переопределения-слота>*)

11.5.8. Дублирование объектов

Для дублирования объектов CLIPS предоставляет четыре функции: duplicate-instance, active-duplicate-instance, message-duplicate-instance и active-message-duplicate-instance. Они позволяют дублировать объекты и обновлять слоты объектов без вызова put-сообщений. Каждая из этих функций возвращает имя нового объекта в случае успеха и значение FALSE — в случае неудачи.

Функции дублирования могут опционально определять имя объекта, в который будет скопирован старый объект. Если это имя не задано, функция сгенерирует имя, используя функцию (gensym*). Если заданный объект уже существует, он будет удален с или без использования сообщений в зависимости от того, какая функция была вызвана.

Функция duplicate-instance использует сообщение direct-duplicate для изменения значений слотов объекта. Сопоставление образцов приостанавливается до тех пор, пока копирование всех слотов не будет выполнено.

Определение 11.26. Синтаксис функции duplicate-instance

(duplicate-instance <объект> [to <объект>] <переопределения-слота>*)

Пример использования функции duplicate-instance приведен на рис. 11.30.

Если необходимо выполнить дублирование объекта без приостановки процесса сопоставления образцов, можно воспользоваться функцией active-duplicate-instance. Она также использует сообщение direct-duplicate для изменения значений слотов объекта, но не останавливает при этом процесс сопоставления образцов.

Определение 11.27. Синтаксис функции active-duplicate-instance

(active-duplicate-instance <объект> [to <объект>]
<переопределения-слота>*)

Функция message-duplicate-instance использует сообщение message-duplicate для изменения значений слотов объекта. Сопоставление образцов приостанавливается до тех пор, пока все изменения слотов не будут выполнены.

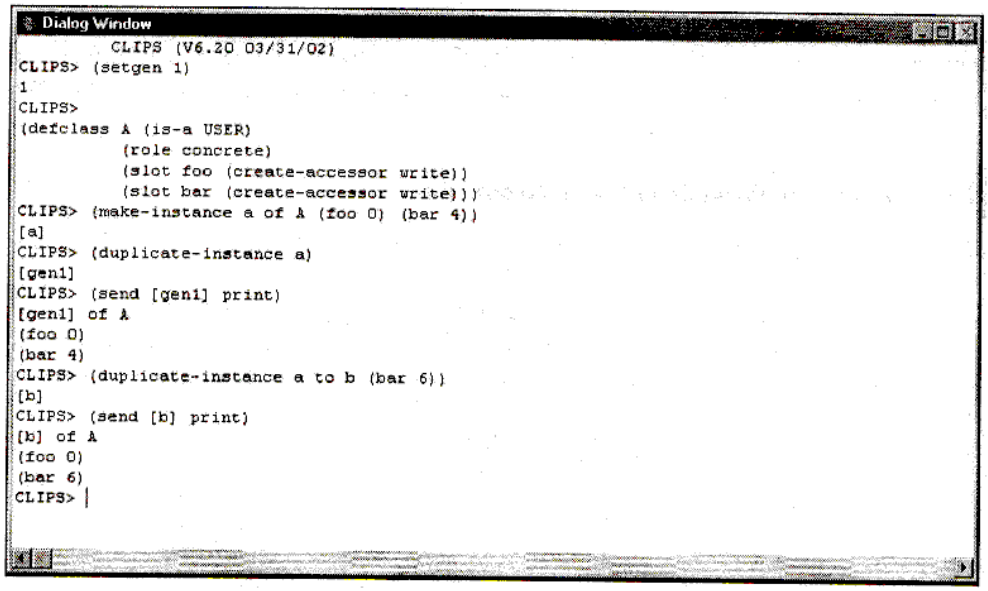

Рис. 11.30. Использование функции duplicate-instance

Определение 11.28. Синтаксис функции message-duplicate-instance

(message-duplicate-instance <объект> [to <объект>]
<переопределения-слота>*)

Функция active-message-duplicate-instance использует сообщение message-duplicate для изменения значений слотов объекта. Сопоставление образцов продолжается во время выполнения изменения слотов.

Определение 11.29. Синтаксис функции active-message-duplicate-instance

(active-message-duplicate-instance <объект> [to <объект>]
<переопределения-слота>*)

11.6. Наборы объектов

COOL предоставляет полезную возможность составления *наборов объектов* (instance-set) и выполнения некоторых действий над таким набором. Система запросов для составления наборов объектов в COOL состоит из шести функций, каждая из которых оперирует набором объектов, определенным с помощью заданного пользователем критерия.

В последующих разделах будут описаны способы формирования шаблонов наборов, запросов и действий над наборами. В приведенных ниже примерах предполагается, что определения, указанные в примере 11.46, уже присутствуют в системе.

Пример 11.46. Необходимые классы и объекты

```
(defclass      PERSON (is-a USER)
               (role abstract)
               (slot sex (access read-only)
                         (storage shared))
               (slot age (type NUMBER)
                         (visibility public)))
(defmessage-handler  PERSON put-age (?value)
               (dynamic-put age ?value))
(defclass      FEMALE (is-a PERSON)
               (role abstract)
```

```
                    (slot sex (source composite)
                              (default female)))
(defclass           MALE (is-a PERSON)
                    (role abstract)
                    (slot sex (source composite)
                              (default male)))
(defclass           GIRL (is-a FEMALE)
                    (role concrete)
                    (slot age (source composite)
                              (default 4)
                              (range 0.0 17.9)))))
(defclass           WOMAN (is-a FEMALE)
                    (role concrete)
                    (slot age (source composite)
                              (default 25)
                              (range 18.0 100.0)))
(defclass           BOY (is-a MALE)
                    (role concrete)
                    (slot age (source composite)
                              (default 4)
                              (range 0.0 17.9)))
(defclass           MAN (is-a MALE)
                    (role concrete)
                    (slot age (source composite)
                              (default 25)
                              (range 18.0 100.0)))

(definstances PEOPLE
              (Man-1 of MAN (age 18))
              (Man-2 of MAN (age 60))
              (Woman-1 of WOMAN (age 18))
              (Woman-2 of WOMAN (age 60))
              (Woman-3 of WOMAN)
              (Boy-1 of BOY (age 8))
              (Boy-2 of BOY)
              (Boy-3 of BOY)
              (Boy-4 of BOY)
              (Girl-1 of GIRL (age 8))
              (Girl-2 of GIRL))
```

11.6.1. Определение набора объектов

Набор объектов (instance-set) — это упорядоченная коллекция объектов, определенных пользователем классов. Каждый *член набора объектов* (instance-set member) — объект некоторого набора классов, называемых *ограничениями классов* (class restrictions). Различные ограничения классов могут задаваться для каждого члена набора объектов. Ограничения вместе с *переменными* (instance-set member variables), с которыми будут связываться соответствующие члены набора, называются *шаблоном набора* (instance-set templates). Функции запросов используют шаблон для генерации самого набора объектов. В ограничении классов может быть задан модуль *(см. гл. 12),* где определен класс. Если модуль не задан, будут использоваться определения классов, созданных в текущем модуле.

Определение 11.30. Синтаксис шаблона набора объектов

<шаблон-набора-объектов> ::= (<члены-шаблона-наборов-объекта>)

<член-шаблона-наборов-объекта> ::=(<переменная-набора-объектов> <ограничения-классов>)

<переменная-набора-объектов> ::= <простая-переменная>

<ограничения-классов> ::= <имена-классов>

В качестве примера можно привести шаблон, выбирающий пары объектов персон противоположного пола.

Пример 11.47. Шаблон, выбирающий пары персон противоположного пола

```
( (?man-or-boy BOY MAN)   (?woman-or-girl  GIRL WOMAN))
```

Этот шаблон можно представить также в эквивалентной форме.

Пример 11.48. Эквивалентная форма предыдущего шаблона

```
( (?man-or-boy MALE)   (?woman-or-girl   FEMALE))
```

В данном примере переменные ?man-or-boy и ?woman-or-giri ограничиваются только с помощью имен классов.

11.6.2. Создание набора объектов

COOL использует прямой перебор для генерации наборов объектов, которые соответствуют шаблону для существующих в системе объектов по перечисленным ниже правилам:

1.Если шаблону удовлетворяет более одного набора, то элементы набора изменяются справа налево.

2.Если в ограничениях классов задано более одного класса, они перебираются слева направо.

3.Экземпляры объектов, определенные в системе, рассматриваются в порядке их создания.

Для шаблона, приведенного в предыдущем разделе, CLIPS сгенерирует 30 наборов объектов в следующем порядке:

1.[Boy-1] [Girl-1]
2.[Boy-1] [Girl-2]
3.[Boy-1] [Woman-1]
4.[Boy-1] [Woman-2]
5.[Boy-1] [Woman-3]
6.[Boy-2] [Girl-1]
7.[Boy-2] [Girl-2]
8.[Boy-2] [Woman-1]
9.[Boy-2] [Woman-2]
10. [Boy-2] [Woman-3]
11.[Boy-3] [Girl-1]
12.[Boy-3] [Girl-2]
13.[Boy-3] [Woman-1]
14.[Boy-3] [Woman-2]
15.[Boy-3] [Woman-3]
16. [Boy-4] [Girl-1]
17. [Boy-4] [Girl-2]
18. [Boy-4] [Woman-1]
19- [Boy-4] [Woman-2]
20. [Boy-4] [Woman-3]
21. [Man-1] [Girl-1]
22. [Man-1] [Girl-2]
23. [Man-1] [Woman-1]
24. [Man-1] [Woman-2]
25. [Man-1] [Woman-3]
26. [Man-2] [Girl-1]
27. [Man-2] [Girl-2]
28. [Man-2] [Woman-1]
29. [Man-2] [Woman-2]
30. [Man-2] [Woman-3]

Пример 11.49. Шаблон, выбирающий пары персон женского пола

```
((?f1   FEMALE)   (?f2   FEMALE))
```

Шаблон из примера 11.49 приведет к генерации 25 наборов объектов:

10.[Girl-1] [Girl-1]
11.[Girl-1] [Girl-2]
12.[Girl-1] [Woman-1]
13.[Girl-1] [Woman-2]
14. [Woman-1] [Woman-2]
15. [Woman-1] [Woman-3]
16. [Woman-2] [Girl-1]
17. [Woman-2] [Girl-2]

14.[Girl-1] [Woman-3]
15.[Girl-2] [Girl-1]
16.[Girl-2] [Girl-2]
17.[Girl-2] [Woman-1]
18.[Girl-2] [Woman-2]
16. [Girl-2] [Woman-3]
17.[Woman-1] [Girl-1]
18.[Woman-1] [Girl-2]
19.[Woman-1] [Woman-1]
18. [Woman-2] [Woman-1]
19. [Woman-2] [Woman-2]
20. [Woman-2 [Woman-3]
21. [Woman-3] [Girl-1]
22. [Woman-3] [Girl-2]
23. [Woman-3] [Woman-1]
24. [Woman-3] [Woman-2]
25. [Woman-3] [Woman-3]

Порядок генерации приведенных выше шаблонов обусловлен в первую очередь тем, что объекты класса GIRL рассматриваются перед объектами класса WOMAN, т. к. сам класс GIRL был определен раньше класса WOMAN.

11.6.3. Определение запроса

Запрос — это определенное пользователем логическое выражение, применяемое к набору объектов для установления дополнительных ограничений для членов набора. Если значение логического выражения не равняется FALSE, то считается, что набор объектов удовлетворяет запросу.

Определение 11.31. Синтаксис запроса

<запрос> ::=(<логическое-выражение>)

Предположим, что необходимо найти пары противоположного пола одного возраста. Для этого следует использовать запрос, представленный в примере 11.50.

Пример 11.50. Запрос, накладывающий ограничение на возраст

```
(= (send ?man-or-boy get-age) (send ?woman-or-girl get-age))
```

В запросе можно обращаться к слотам членов набора объекта напрямую при помощи краткой нотации, скрывающей использование сообщений.

Определение 11.32. Синтаксис обращения к переменным

<переменная-набора-объектов>:<имя-слота>

С учетом сказанного, предыдущий пример может быть переписан так, как представлено в примере 11.51.

Пример 11.51. Измененный запрос, накладывающий ограничение на возраст

```
(= ?man-or-boy:age ?woman-or-girl:age)
```

11.6.4. Определение действий

Некоторые функции позволяют пользователю определить действие, которое будет выполнено для каждого набора объектов. В отличие от запросов действия должны использовать сообщения для чтения значений слотов. В случае если над объектом требуется выполнить несколько операций, используется функция prong *(см. гл. 15)* для их группировки.

Определение 11.33. Синтаксис определения действия

<действие> ::= <функция>

Обобщая приведенные выше примеры, в качестве иллюстрации использования действий над набором объектов воспользуемся выражением из примера 11.52.

Пример 11.52. Действия над набором объектов

```
(do-for- all-instances
        ((?man-or-boy MALE) (?woman-or-girl FEMALE))
        (= ?man-or-boy:age ?woman-or-girl:age)
        (printout t "(" ?man-or-boy "," ?woman-or-girl ")" crlf))
```

В данном примере:

- do-for-all-instances — одна из функций CLIPS, работающая с наборами объектов (функции будут рассмотрены ниже);
- ((?man-or-boy MALE) (?woman-or-girl FEMALE)) — шаблон, определяющий набор из пар объектов противоположного пола;
- (= ?man-or-boy:age ?wornan-or-girl :age) — запрос, отбирающий только пары одного возраста;
- (printout t "(" ?man-or-boy "," ?woman-or-girl ")" crlf) — действия, выводящие пары найденного набора на экран.

Результат выполнения данного выражения представлен на рис. 11.31.

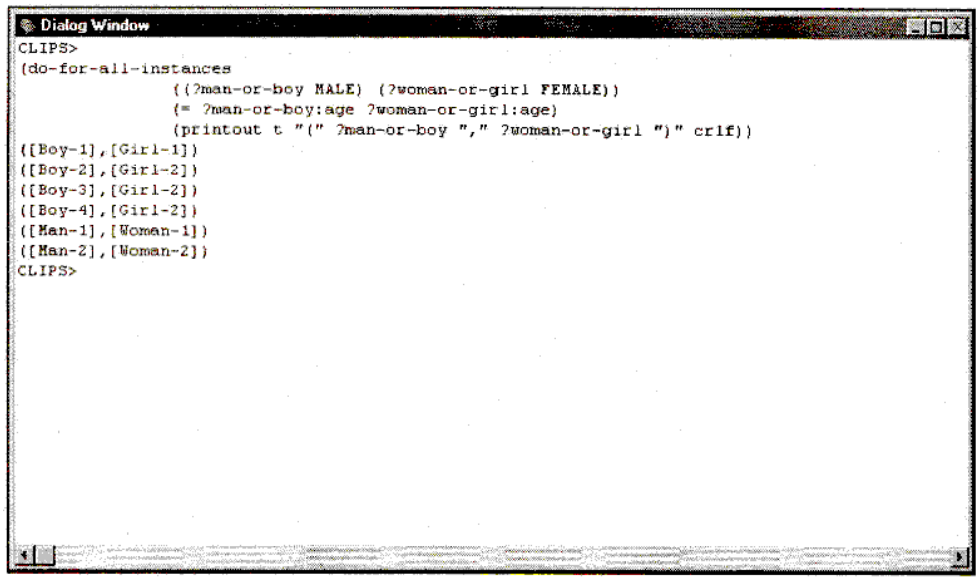

Рис. 11.31. Результат выполнения функции do-for-all-instances над заданным набором

11.6.5. Функции-запросы

CLIPS предоставляет *6* функций (табл. 11.2), способных оперировать с наборами объектов. Эти функции называются *функциями-запросами наборов объектов.*

Таблица 11.2. *Функции для операций над наборами объектов*

Функция	Назначение
any-instancep	Определяет, удовлетворяет ли запросу один или более наборов объектов
find-instance	Возвращает первый объект из набора удовлетворяющему запросу
find-all-instance	Группирует и возвращает все наборы объектов,

	удовлетворяющих запросу
do-for-instance	Выполняет действия для первого набора объектов, удовлетворяющих запросу
do-fo-all-instance	Выполняет действия для каждого набора объектов, удовлетворяющих запросу
delayed-do-for-all-instance	Группирует все наборы объектов, удовлетворяющие запросу, и выполняет действия над этой группой

Функция запроса набора объектов может быть вызвана в произвольном месте, где вызывается обычная функция CLIPS. Если при выполнении любой из этих функций происходит ошибка, функция тут же прерывается и возвращает значение false.

Пользователь имеет возможность прерывать выполнение некоторых из указанных функций и возвращать при этом какой-то результат. Для этого служат функции break и return, которые можно использовать внутри функций do-for-instance, do-for-all-instance и delayed-do-for-all-instances. Функция break фактически прерывает выполнение запроса, а функция return помогает вернуть некоторое значение.

Функция any-instancep выполняет проверку удовлетворения запроса каким-либо набором объектов. Она применяет запрос к каждому набору объектов, который соответствует шаблону. Если набор объектов удовлетворяет запросу, то функция немедленно прекращается и возвращает значение TRUE. В противном случае — FALSE.

Определение 11.34. Синтаксис функции any-instancep

(any-instancep <шаблон-набора-объектов> <запрос>)

Например, вызов функции any-instancep из примера 11.53 определяет, существует ли в системе объект класса man с возрастом больше 30 лет. С нашими данными функция вернет значение TRUE.

Пример 11.53. Использование функции any-instancep

```
(any-instancep   ((?man MAN))   (>  ?man:age  30))
```

Функция find-instance находит набор объектов, удовлетворяющий заданному запросу. Эта функция применяет запрос к каждому набору объектов, соответствующему шаблону. Если набор объектов удовлетворяет запросу, то работа функции немедленно прекращается, и она возвращает набор объектов в виде составного значения. Каждое поле составного значения является именем объекта из набора объектов. В случае если нужный набор не найден, функция возвращает составное значение нулевой длины.

Определение 11.35. Синтаксис функции find-instance

(find-instance <шаблон-набора-объектов> <запрос>)

Например, для поиска первой пары противоположного пола одинакового возраста необходимо выполнить вызов из примера 11.54.

Пример 11.54. Использование функции find-instance

```
(find-instance   ((?m MAN)   (?w WOMAN))   (=  ?m:age   ?w:age))
```

В нашем случае функция вернет значение ([Man-1] [woman-1]).

Функция find-all-instance находит все наборы, удовлетворяющие запросу. Она применяет запрос к каждому набору объектов, который соответствует шаблону. Набор, удовлетворяющий запросу, сохраняется в составном значении. Это значение возвращается, если запрос был применен ко всем

возможным наборам. Если в каждом наборе *n* объектов и *m* наборов удовлетворяют запросу, то длина возвращенного значения будет *nxm*. Первые *n* полей соответствуют именам объектов первого набора, вторые *n* — второму набору и т. д. Так как составное значение может занимать достаточно большое количество памяти, в связи с возможностью возникновения комбинаторного взрыва (большого числа возможных вариантов наборов, удовлетворяющих запросу), то эту функцию следует использовать с большой осторожностью.

Определение 11.36. Синтаксис функции find-all-instances

(find-all-instances <шаблон-набора-объектов> <запрос>)

Для того чтобы найти все пары мужчин и женщин одинакового возраста, можно вызвать функцию find-all-instances так, как представлено в примере 11.55.

Пример 11.55. Использование функции find-all-instances

```
(find-all-instances ((?mMAN) (?w WOMAN)) (= ?m:age ?w:age))
```

В нашем случае функция вернет составное значение, равное ([MAN1] [Woman-I] [Man-2] [Woman-2]).

Функция do-for-instance предназначена для выполнения некоторого заданного действия для первого набора, удовлетворяющего запросу. Функция do-for-instance применяет запрос к каждому набору объектов, который соответствует шаблону. Если набор объектов удовлетворяет запросу, выполняется определенное действие и функция немедленно прекращается. Возвращенное значение соответствует результату вычисления действия. Если функция не нашла набор объектов, удовлетворяющий запросу, то возвращается значение FALSE.

Определение 11.37. Синтаксис функции do-for-instance

(do-for-instance <шаблон-набора-объектов> <запрос> <действие>)

В качестве примера использования функции do-for-instance можно привести выражение, представленное в примере 11.56.

Пример 11.56. Использование функции do-for-instance

```
(do-for-instance
        ((?pl PERSON) (?p2 PERSON) (?p3 PERSON))
        (and ( = ?pl:age ?p2:age ?p3:age)
             (neq ?pl ?p2)
             (neq ?pl ?p3)
             (neq ?p2 ?p3))
         (printout t ?pl " " ?p2 " " ?p3 crlf))
```

Данный вызов функции выведет на экран сведения о первой тройке различных людей, имеющих одинаковый возраст. Вызовы функции neg в запросе устраняют перестановки с двумя и более одинаковыми членами. С введенными нами данными эта функция должна вывести на экран следующие объекты: [Girl-2] [Boy-2] [Boy-3].

Функция do-for-all-instance выполняет некоторое заданное действие для всех наборов объектов, удовлетворяющих запросу. Она применяет запрос к каждому набору объектов, который соответствует шаблону. Если набор объектов удовлетворяет запросу, выполняется действие, определенное пользователем. Значение, возвращенное функцией, соответствует результату вычисления действия над последним набором объектов. Если функция не нашла ни один набор объектов, удовлетворяющий запросу, то она возвращает значение FALSE.

Определение 11.38. Синтаксис функции do-for-all-instance

(do-for-all-instance <шаблон-набора-объектов> <запрос> <действие>)

Для того чтобы вывести на экран сведения о тройках людей, имеющих одинаковый возраст, предыдущий пример можно переделать.

Пример 11.57. Использование функции do-for-all-instance

```
(do-for-all-instance
        ((?pl PERSON)  (?p2 PERSON) (?p3 PERSON))
        (and (= ?pl:age ?p2:age ?p3:age)
             (neq ?pl ?p2)
             (neq ?pl ?p3)
             (neq ?p2 ?p3))
        (printout t ?pl " " ?p2 " " ?p3 crlf))
```

Помимо функции do-for-all-instance, CLIPS предоставляет еще одну функцию для осуществления некоторого действия над всеми наборами объектов — delayed-do-for-all-instance. Эта функция отличается от do-for-all-instance тем, что она группирует все наборы, которые удовлетворяют запросу, в промежуточное составное значение. Если функция не нашла ни один набор объектов, удовлетворяющий запросу, то она возвращает значение FALSE. В случае если нужные наборы были найдены, заданное действие выполняется для каждого набора из составного значения. Значение, возвращаемое функцией, является результатом заданного действия над последним набором. Так же, как и find-all-instances, эта функция может расходовать большое количество памяти. Функцию delayed-do-for-all-instance необходимо использовать вместо do-for-all-instance в случае, когда действие, применяемое к некоторому набору объектов, может изменить результат запроса для другого набора (конечно, кроме ситуаций, когда это подобное поведение не является желательным).

Определение 11.39. Синтаксис функции delayed-do-for-all-instance

```
(delayed-do-for-all-instance
                    <шаблон-набора-объектов> <запрос> <действие>)
```

В качестве примера рассмотрим ситуацию, когда необходимо удалить объект, представляющий самого старшего мальчика. В этом случае нужно использовать именно функцию delayed-do-for-all-instance. Действие (удаление) должно быть задержано до окончания проверки всех наборов, иначе проверяемый возраст может просто постепенно уменьшаться, и в таком случае может быть удалено больше сведений о мальчиках, чем нужно. Для проверки возраста будем использовать вложенную функцию any-instancep для поиска информации о каком-нибудь мальчике старше текущего. Полностью решение данной задачи приведено на рис. 11.32.

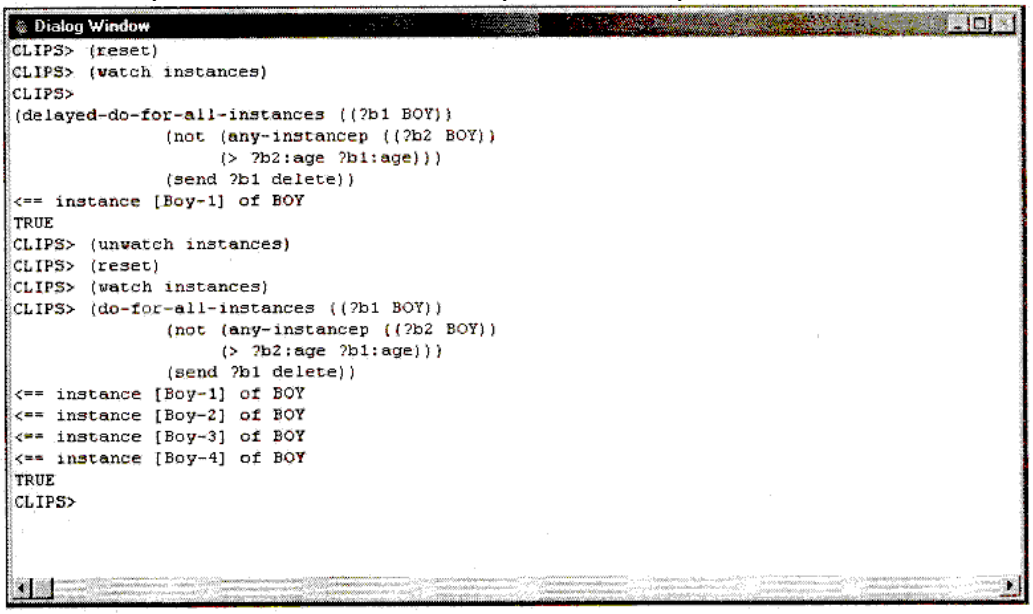

Рис. 11.32. Использование функции delayed-do-for-all-instances

ГЛАВА 12. Модули

CLIPS предоставляет возможность разбиения базы данных и решения задачи на отдельные независимые модули. Для создания таких модулей служит конструктор defmodule. С помощью модулей можно группировать вместе отдельные элементы базы знаний и управлять процессом доступа к этим элементам во время решения некоторой задачи. Подобный процесс управления доступа к данным напоминает механизм пространства имен, используемый в C++, и глобальные и локальные области видимости в языках C и Ada. Однако, в отличие от механизмов в перечисленных выше языках, области видимости в CLIPS строго иерархичны и однонаправлены: если модуль A может видеть данные модуля в, это не означает, что модуль в может видеть данные модуля A. С помощью управления ограничением доступа к данным, содержащимся в различных модулях, при решении сложных задач модули могут реализовывать концепцию *доски объявлений (blackboard strategy* — стратегия решения задач с использованием разнородных источников знаний, взаимодействующих через общее информационное поле). В этом случае отдельный модуль позволяет видеть строго определенный набор фактов и объектов правилам из других модулей. Кроме того, модули используются для управления потоком вычисления правил.

12.1. Создание модулей

Как уже упоминалось выше, для создания модулей служит конструктор defmodule.

Определение 12.1. Синтаксис конструктора defmodule

```
(defmodule  <имя-модуля>
            [<комментарии>]
            <спецификации-импорта-экспорта>*)

<спецификация-импорта-экспорта>     ::=
                                    (export <элемент-слецификации>) |
                                    (import <имя-модуля> <элемент-спецификации>
<элемент-спецификации>::= ?ALL |
                          ?NONE |
                          <конструктор> ?ALL |
                          <конструктор> ?NONE |
                          <конструктор> <имя-конструктора>
<конструкция>:= deftemplate | defclass | defglobal | deffunction | defqeneric
```

После своего создания модуль не может быть переопределен или удален (за исключением системного модуля MAIN, который пользователь может один раз переопределить). Единственный способ удалить существующий модуль — выполнить команду clear. Во время запуска системы и при вызове команды clear CLIPS автоматически создает предопределенный системный модуль, указанный ниже.

Определение 12.2. Предопределенный конструктор модуля MAIN

```
(defmodule MAIN)
```

Все предопределенные системные классы принадлежат системному модулю MAIN, однако нет необходимости экспортировать или импортировать системные классы в другие модули, они всегда находятся в области видимости определенных пользователем модулей. Предопределенный системный модуль MAIN не импортирует и не экспортирует никаких конструкций. Однако, в отличие от других модулей, пользователь может один раз переопределить модуль MAIN после запуска системы или выполнения команды clear.

12.2. Определения модулей в конструкторах

Для определения, в какой модуль будет помещена та или иная конструкция языка, созданная соответствующим конструктором, в конструкторе необходимо указать имя модуля. Конструкторы deffacts, deftemplate, defrule, deffunction, defgeneric, defclass и definstances для определения имени модуля позволяют включать его в имя соответствующей конструкции. Конструктор defglobal принимает имя модуля в специально отведенное для этого поле, которое следует сразу за ключевым словом defglobal. Конструктор defmessage-handier принимает имя модуля как часть определения класса, с которым связывается сообщение. Конструктор defmethod принимает имя модуля как часть определения родовой функции, которой принадлежит данный метод. Например, все приведенные ниже конструкторы будут помещены в модуль DETECTION.

Пример 12.1. Применение спецификации модуля

```
(defrule DETECTION::Find-Fault
          (sensor (name ?name) (value bad))
           =>
           (assert (fault (name ?name))))
(defglobal DETECTION ?*count* = 0)
(defmessage-handler DETECTION::COMPONENT get-charge ()
                        (* ?self:flux ?self:flow))
 (defmethod DETECTION::+ ((?x STRING) (?y STRING))
              (str-cat ?x ?y))
```

Выполните следующую последовательность действий:

Пример 12.2. Использование модулей

```
(clear)
(defmodule A)
(defmodule B)
(defrule  foo =>)
(defrule A::bar =>)
(list-defrules)
(set-current-module B)
(list-defrules)
```

Результат выполнения этих команд приведен на рис. 12.1.

Обратите внимание, что после определения нового модуля он становится текущим (имя текущего модуля можно получить с помощью функции get-current-module). Таким образом, правило foo было добавлено в текущий модуль B, т. к. при его создании модуль не был указан явно, а правило bar добавлено в модуль A, что явно указано в конструкторе. Сообщения, возникшие после определения правил, сообщают об определении в новых модулях фактов initial-fact, необходимых для безусловных правил. После этого, переключая текущий активный модуль с помощью команды set-current-module и используя команду list-defrules, можно убедиться, что правила находятся именно в тех модулях, в которых они должны находиться. Windows-версия CLIPS предоставляет еще один способ просмотра списка определенных пользователем модулей и изменения текущего модуля. Эта возможность реализована с помощью вложенного меню **Module,** содержащегося в меню **Browse.** Текущий модуль в этом меню отмечен флажком (рис. 12.2).

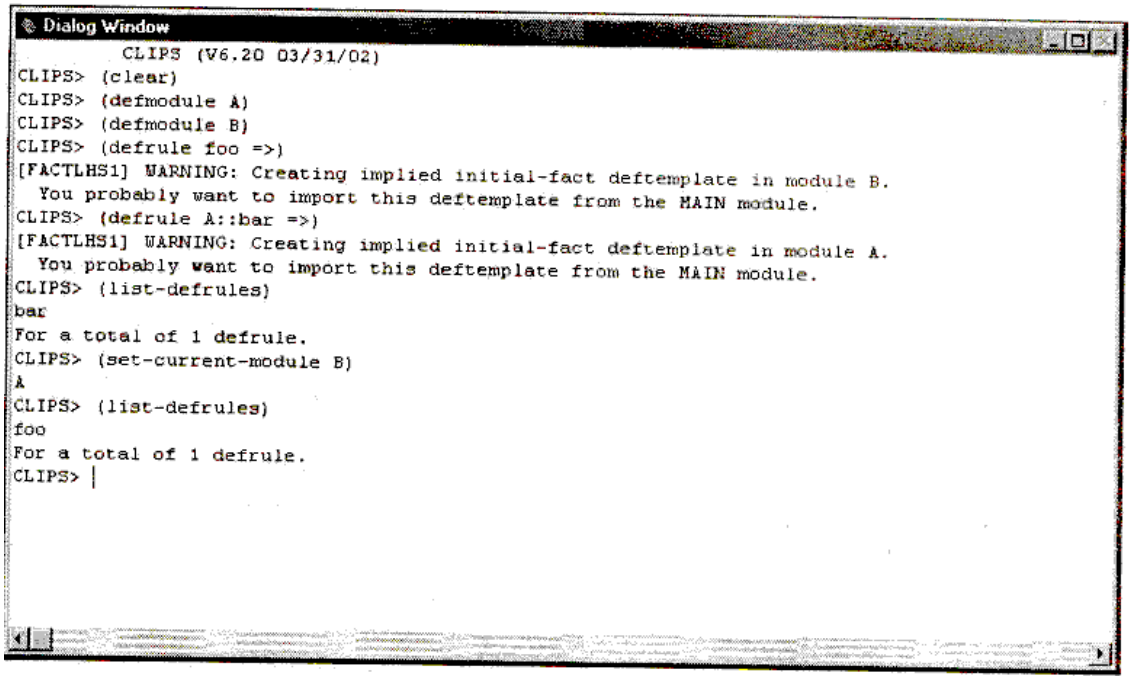

Рис. 12.1. Использование модулей

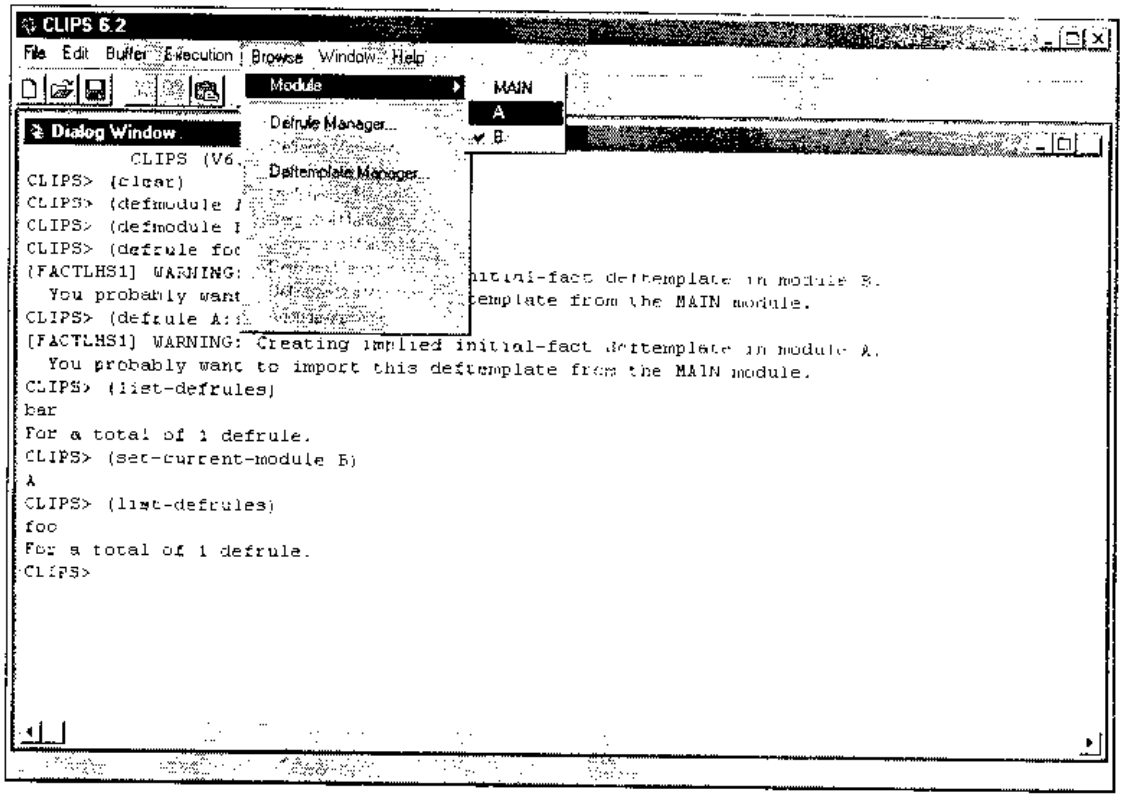

Рис. 12.2. Выбор активного модуля

12.3. Использование модулей в командах и функциях

Многие команды, например undefrule или ppdefrule, используют имя конструкции, которой оперируют. В предыдущих версиях CLIPS имени конструкции было вполне достаточно для однозначной идентификации. Однако после введения модулей стало возможным существование конструкций с одинаковыми именами в двух различных модулях. Модуль конструкции, используемой в команде, может быть задан явно или неявно.

Явное задание модуля выполняется с помощью имени модуля, разделенного с именем конструкции при помощи двойного двоеточия **::** . Имя модуля и символ **::** называются *спецификатором модуля* (module specifier). Например, запись MAIN**::** find-stuff ссылается на конструкцию find-stuff из модуля MAIN.

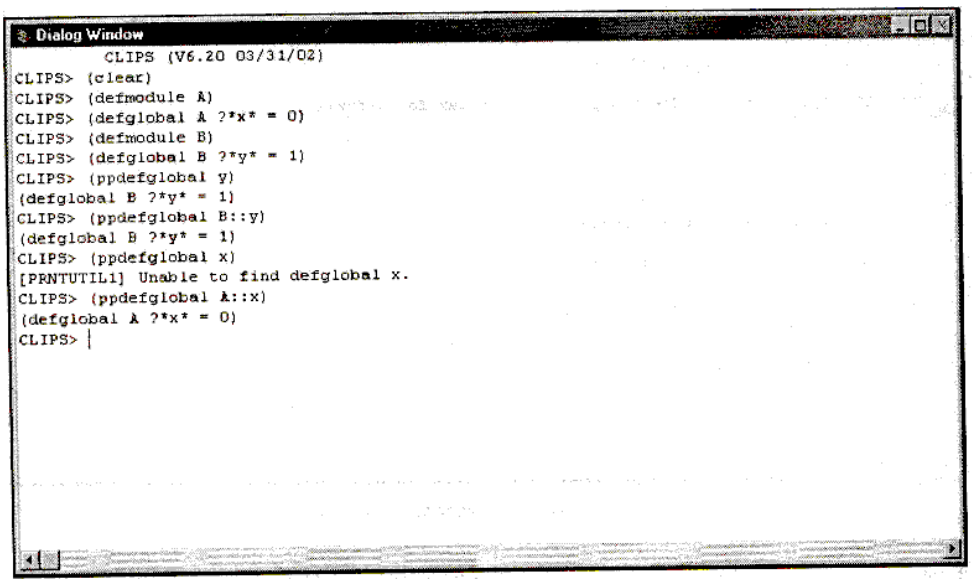

Рис. 12.3. Пример использования спецификатора модуля

Неявное задание модуля выполняется с помощью установки текущего активного модуля. Текущий модуль меняется при каждом определении нового модуля или при вызове функции set-current-module. Так как модуль MAIN автоматически добавляется в систему при загрузке системы, а также при каждом вызове функции clear, то MAIN является текущим модулем по умолчанию. Таким образом, имя find-stuff ссылается на конструкцию find-stuff из модуля MAIN. Пример явного и неявного задания модуля в командах приведен на рис. 12.3.

12.4. Импорт и экспорт конструкций

За исключением специально экспортированных и импортированных, конструкции, определенные в одном модуле, не могут использоваться в другом модуле. Конструкция называется видимой или находящейся в пределах области видимости модуля, если она может использоваться в модуле. Например, если пользователь хочет указать в модуле в конструктор deftemplate с именем foo, определенный в модуле A, то модуль A должен экспортировать deftemplate foo, а модуль B должен импортировать deftemplate foo их модуля A. Подобная ситуация приведена на рис. 12.4.

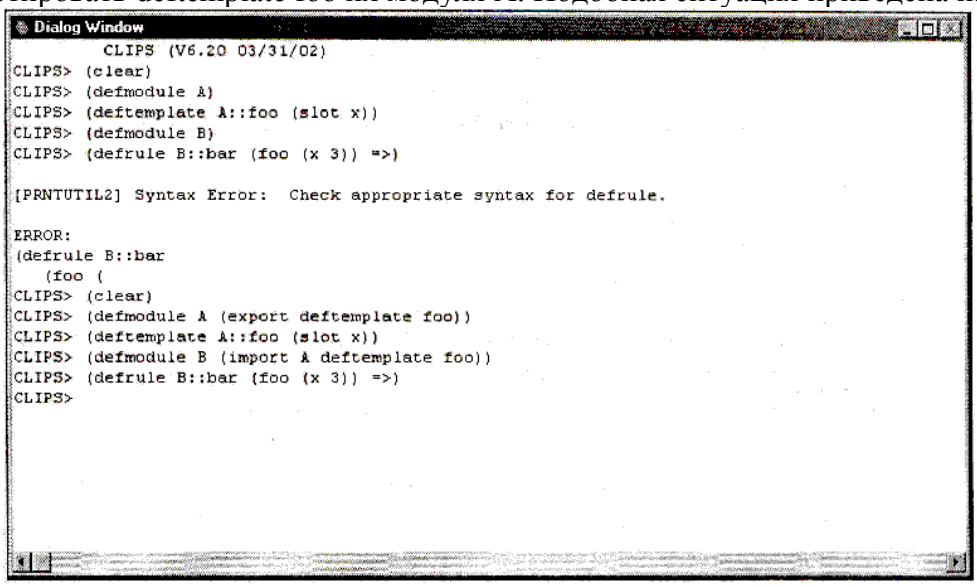

Рис. 12.4. Импорт/экспорт шаблонов

CLIPS не допускает существования двух конструкторов с одинаковыми именами, видимых в одном модуле.

Спецификация экспорта в определении модуля служит для определения, какие именно конструкции данного модуля могут импортироваться другими модулями. Экспортироваться способны только следующие конструкции: deftemplates, defclasses, defglobals, deffunctions, и defgenerics. Модуль может экспортировать любую видимую конструкцию данных типов. При этом не обязательно, чтобы эта конструкция была непосредственно определена в данном модуле.

В CLIPS существует три типа спецификации экспорта.

- Во-первых, модуль может экспортировать все видимые в нем конструкции. Это осуществляется с помощью ключевого слова export и следующего за ним ключевого слова ?ALL.
- Во-вторых, модуль может экспортировать все видимые в нем конструкции заданного типа. Для этого используется ключевое слово export, тип конструкции и ключевое слово ?ALL.
- В-третьих, модуль может экспортировать некоторые отдельные конструкции заданного типа. Это осуществляется с помощью ключевого слова export, типа конструкции, списка из одного или более имен видимых конструкций заданного типа, которые необходимо экспортировать.

В приведенном ниже примере модуль A экспортирует все видимые в нем конструкции, модуль B — все конструкции deftemplate, а модуль C — три отдельных конструкции defglobal.

Пример 12.3. Пример экспорта конструкций

```
(defmodule A (export ?ALL))
(defmodule B (export deftempiate ?ALL))
(defmodule C (export defglobal foo bar yak))
```

Вместо ключевого слова ?ALL в спецификации экспорта может использоваться ключевое слово ? NONE. В этом случае модуль не будет экспортировать либо вообще никаких конструкций, либо не будет экспортировать никаких конструкций заданного типа.

Конструкции defmethod и defmessage-handler никогда не экспортируются явно. Экспорт конструктора defgeneric автоматически приводит к экспорту всех ассоциированных с ним конструкторов defmethod. Экспорт конструктора defclass — к автоматическому экспортированию всех связанных с классом обработчиков (конструкторов defmessage-handler). Конструкции deffacts, definstances и defrules вообще не могут быть экспортированы.

Спецификация импорта в определении модуля служит для определения, какие конструкции из других модулей могут использоваться в данном модуле. Импортироваться могут только следующие конструкции: deftemplates, defclasses, defglobals, deffunctions и defgenerics.

В CLIPS существует три типа спецификации импорта.

- Во-первых, модуль может импортировать все конструкции, видимые в некотором заданном модуле. Это осуществляется с помощью ключевого слова import, имени модуля, из которого будет производиться импорт, и ключевого слова ?ALL.
- Во-вторых, модуль может импортировать все конструкции заданного типа, видимые в некотором заданном модуле. Для этого используется ключевое слово import, имя модуля, тип конструкции и ключевое слово ?ALL.
- В-третьих, модуль может импортировать некоторые отдельные конструкции заданного типа. Это осуществляется с помощью ключевого слова import, имени модуля, из которого будет производиться импорт, типа конструкции, списка из одного или более имен видимых конструкций заданного типа, которые необходимо импортировать.

В приведенном ниже примере модуль A импортирует все видимые конструкции модуля D, модуль B — все конструкции deftemplate из модуля D, а модуль C — три отдельных конструкции defglobal, также определенные в модуле D.

Пример 12.4. Пример импорта конструкций

```
(defmodule A (import D ?ALL))
(defmodule B (import D deftemplate ?ALL)}
(defmodule C (import D defglobal foo bar yak))
```

Вместо ключевого слова ?ALL в спецификации импорта может использоваться ключевое слово ? NONE. В этом случае модуль не будет импортировать либо вообще никаких конструкций, либо не будет импортировать никаких конструкций заданного типа.

Конструкции defmethod и defmessage-handler никогда явно не импортируются. Импорт конструкторов defgeneric приводит к импортированию всех ассоциированных с ним конструкторов defmethod. Импорт конструкторов defclass приводит к автоматическому импортированию всех связанных с классом обработчиков (конструкторов defmessage-handler). Конструкции deffacts, definstances и defrules не могут быть импортированы.

Модуль должен быть определен до того, как он будет использован в спецификации импорта. Кроме того, указанные в спецификации импорта конструкции должны экспортироваться соответствующим модулем.

12.5. Импорт и экспорт фактов и объектов

В CLIPS факты и объекты принадлежат не тому модулю, в котором они были созданы, а тому, в котором был определен соответствующий конструктор deftemplate или defclass. Таким образом, факты и объекты становятся видимыми в тех модулях, которые импортируют соответствующие конструкции deftemplate или defclass (рис. 12.5). Это позволяет разбивать базу знаний таким образом, чтобы правила или другие конструкции могли видеть только необходимые им факты и объекты.

Следует отметить, что конструкцию deftemplate, определяющую initial-fact, и конструкцию defclass, определяющую initial-object, необходимо явно импортировать из модуля MAIN. Без этого правила, содержащие initial-fact или initial-object в левой части (например, безусловные правила или правила, содержащие not в качестве первого условного элемента) не будут активированы.

Имя объекта должно быть уникально в пределах модуля, но в области видимости одного модуля могут находиться одновременно несколько объектов с одинаковым именем. Поэтому в CLIPS, начиная с версии 6.0, синтаксис задания имени объекта был расширен, что позволило определять модуль, в котором определен объект (обратите внимание, что квадратные скобки в данном случае являются частью синтаксиса языка CLIPS, а не обозначают необязательность элементов определения).

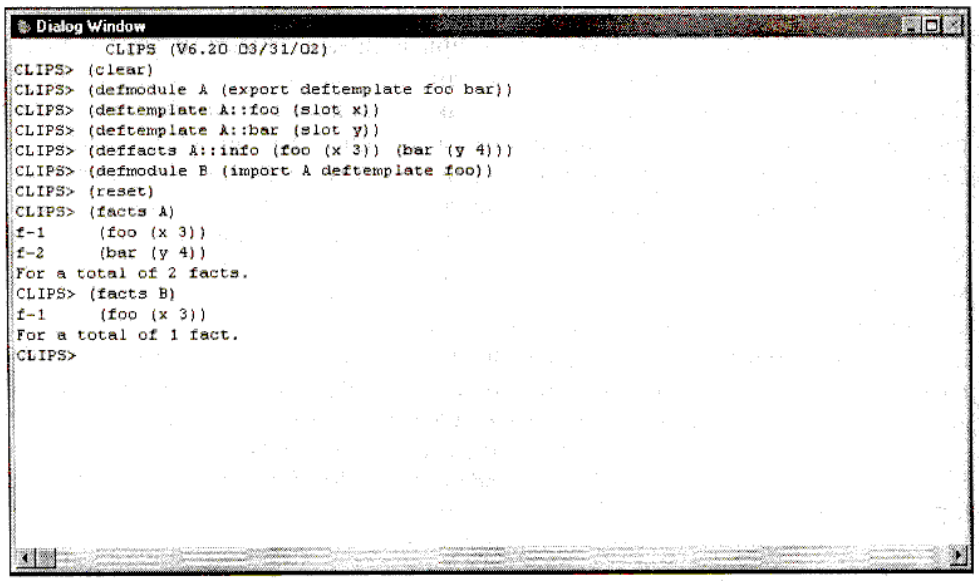

Рис. 12.5. Импорт/экспорт фактов

Определение 12.3. Синтаксис определения имени объекта

<имя-объекта> ::= [<имя>] |
[::<имя>] |
[<модуль >::<имя>]

Использование для поиска объекта только имени, например [Rolls-Royce], приведет к поиску объекта в текущем модуле. Указание символа **::** перед именем объекта, например [**::**Rolls-Royce], позволит выполнить поиск объекта в текущем модуле, а затем во всех модулях, определенных в списке импортирования текущего модуля. Использование имени модуля перед именем

объекта, например [CARS::Rolls-Royce], приведет к поиску объекта в заданном модуле. Независимо от того, какой способ ссылки на объект используется, класс данного объекта должен находиться в области видимости модуля.

12.6. Модули и выполнение правил

Каждый модуль имеет свой собственный процесс сопоставления образцов для своих правил и свой план решения задачи. По команде run начинает выполняться план решения задачи модуля, на который в данный момент установлен фокус. Команды reset и clear автоматически устанавливают фокус на модуль MAIN. Выполнение правил продолжается до тех пор, пока в плане решения задачи не останется применимых правил, и другой модуль не получит фокус, либо правая часть одного из выполняемых правил не вызовет функцию return. После того как в плане решения задачи модуля, имеющего фокус, заканчиваются правила, текущей модуль удаляется из *стека фокусов* (focus stack) и находящийся в стеке следующий модуль получает фокус. Перед выполнением правила текущим становится модуль, в котором данное правило определено. Управлять стеком фокусов можно с помощью команды focus. На рис. 12.6 приведен пример использования модулей, правил и команды focus.

Текущее состояние стека фокусов можно просматривать с помощью команды get-focus-stack или list-focus-stack. Пользователи Windows-версии системы CLIPS могут просматривать содержимое стека фокусов и его изменение в процессе решения задачи посредством окна **Focus.** Это окно вызывается с помощью пункта **Focus Window** меню **Windows.** Внешний вид окна **Focus** показан на рис. 12.7.

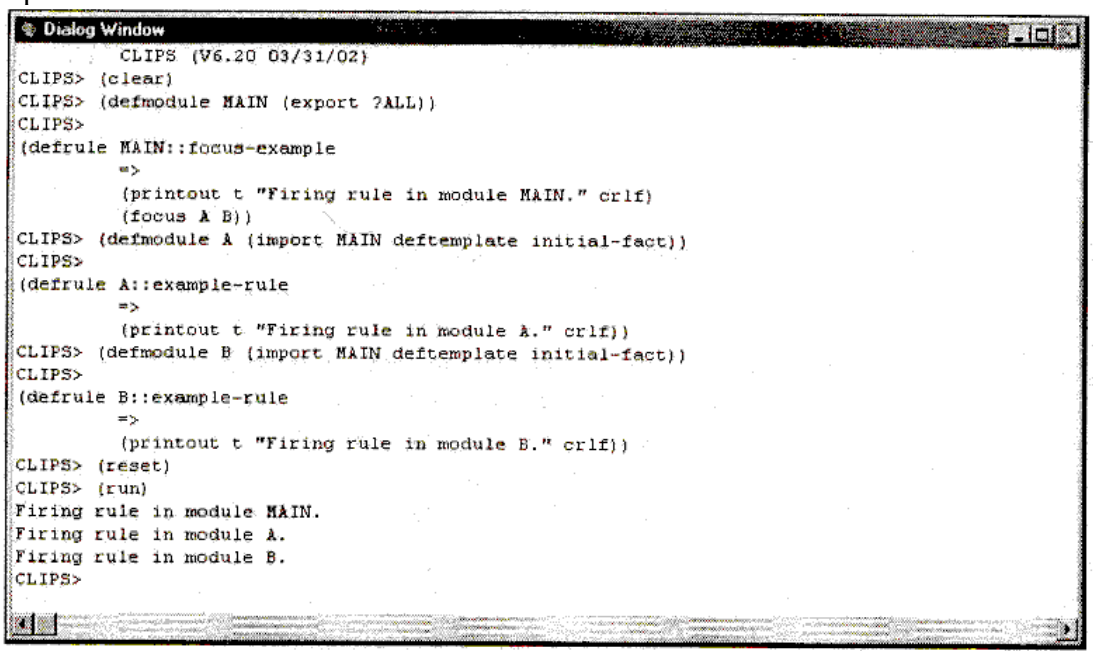

Рис. 12.6. Использование правил в различных модулях

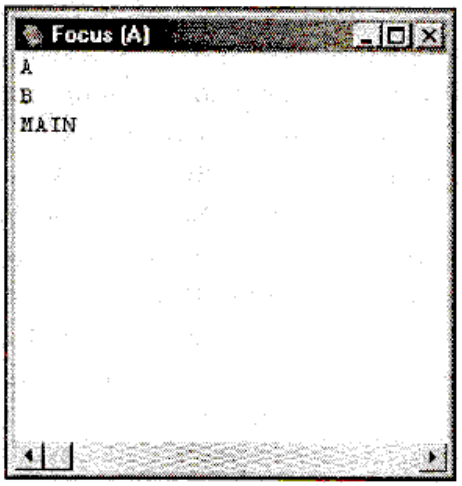

Рис. 12.7. Окно **Focus** для модуля A

ГЛАВА 13Ограничения

Эта глава посвящена вопросу ограничений, применяемых в фактах или объектах, и типам проверки значений слотов и полей. Кроме того, атрибуты ограничений используются в левой части правил для определения дополнительных условий запуска правил, проверяемых во время процесса сопоставления образцов.

CLIPS поддерживает два типа проверки ограничений — *статическую* и *динамическую.* Если включен режим статической проверки ограничений, нарушения ограничений обнаруживаются при вызове функций и создании различных новых конструкций. Статическая проверка также обеспечивает проверку на соответствие переменных в левой части правил. Если включен режим динамической проверки ограничений, проверка осуществляется при появлении любых новых данных (таких как факты или объекты). Фактически, можно считать, что статическая проверка выполняется при загрузке программы в память, а динамическая — при ее выполнении. По умолчанию в CLIPS включена статическая проверка ограничений, а динамическая выключена. Эту установку можно изменить с помощью функций set-static-constraint-checking И set-dynamic-constraint-checking. Кроме того, пользователи Windows-версии среды CLIPS могут устанавливать режимы проверки ограничений с помощью диалогового окна **Execution Options,** открываемого одноименной командой через меню **Execution.** Внешний вид этого диалогового окна приведен на рис. 13.1.

Если режим динамической проверки ограничений не включен, информация об ограничениях, ассоциированная с различными конструкциями, не сохраняется в бинарный файл, создаваемый командой bsave.

Обобщенный синтаксис атрибутов ограничений можно представить в следующем виде.

Определение 13.1. Синтаксис атрибутов ограничений

<атрибуты-ограничений> ::= <атрибут-типа> |
<константный-атрибут> |
<атрибут-диапазона> | <атрибут-мощности>

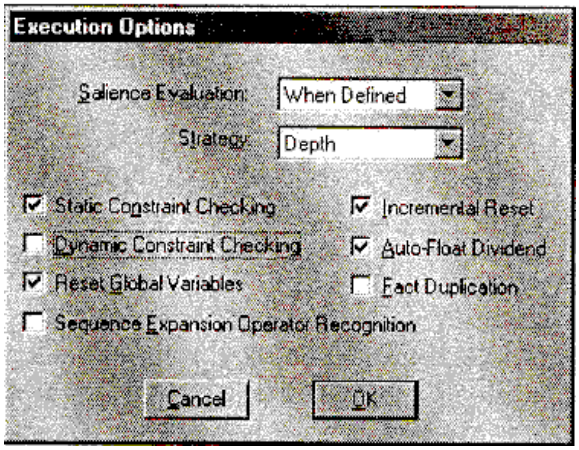

Рис. 13.1. Диалоговое окно **Execution Options**

13.1. Атрибут типа

Атрибут типа позволяет ограничивать типы значений, помещаемых в слоты объектов или поля фактов.

Определение 13.2. Синтаксис атрибута ограничения типа

<атрибут-типа> := (type <спецификация-типа>)
спецификация-типа> := <допустимые-типы> | ?VARIABLE
<допустимый-тип> ::= SYMBOL | STRING | LEXEME |
INTEGER | FLOAT | NUMBER |
INSTANCE-NAME | INSTANCE-ADDRESS |
INSTANCE | EXTERNAL-ADDRESS | FACT-ADDRES

Использование в качестве атрибута типа NUMBER эквивалентно использованию двух типов: INTEGER и FLOAT. Таким же образом тип LEXEME эквивалентен паре типов SYMBOL и STRING, а тип INSTANCE эквивалентен INSTANCE-NAME и INSTANCE-ADDRESS. Использование ключевого слова ? VARIABLE позволяет сохранять в слот значения различных типов.

13.2. Константный атрибут

Константный атрибут позволяет задавать константные значения определенного типа, которые могут сохраняться в данном слоте. Список значений константного атрибута может принимать два значения: либо список значений некоторых констант заданного типа, либо символ ? VARIABLE, который означает, что допустимым является любое значение заданного типа. Атрибут allowed-values отличается от остальных константных атрибутов. Он позволяет задавать список допустимых значений любых типов. Обратите внимание на разницу между атрибутами, например, (allowed-symbols red green blue) И (allowed-values red green blue). Атрибут allowed-symbols строго определяет, что заданные с его помощью значения имеют тип SYMBOL. Атрибут allowed-values ограничивает различные допустимые значения атрибута, невзирая на тип. Общий синтаксис константного атрибута можно представить следующим образом.

Определение 13.3. Синтаксис константного атрибута

<константный-атрибут> :: =(allowed-symbols <список-symbol-значений>) |
(allowed-strings <список-string-значений>) |
(allowed-lexemes <список-1exeme-значений> |
(allowed-integers <список-integer-значений>) |
(allowed-floats <список-f1oat;-значений>) |
(allowed-numbers <список-number-значений>) |
(allowed-instance-names < список -instance-значений) |
(allowed-values <список-значений>)
<список- symbo1-значений> ::=< symbol >+ | ?VARIABLE
<список-string-значений> ::=< string >+ | ?VARIABLE
<список-1exeme-значений> ::=< 1exeme >+ | ?VARIABLE
<список-integer-значений> ::=< integer >+ | ?VARIABLE
<список-f1oat-значений> ::=< f1oat >+ | ?VARIABLE
<список-number-значений> ::=< number >+ | ?VARIABLE
<cписок-instance-значений> ::=< instance -name>+ | ?VARIABLE
<список-значений> ::=<constant>+ | ?VARIABLE

Указание атрибута allowed-lexemes позволяет использовать значения как типа SYMBOL, так и типа STRING. Значения этих типов должны совпадать с одним из значений типа LEXEME из списка значений. Преобразование типа из SYMBOL в STRING или из STRING в SYMBOL При этом не выполняется. Аналогично, применение атрибута allowed-numbers позволяет использовать значения как типа INTEGER, так и типа FLOAT. При использовании атрибута allowed-numbers в CLIPS версии 6.0 выполняется преобразование типов из INTEGER в FLOAT или из FLOAT в INTEGER (таким образом, указание этого атрибута не эквивалентно использованию пары атрибутов allowed-integers и allowed-floats). В CLIPS версии 5.1 атрибут allowed-instances не поддерживался, вместо него использовался атрибут allowed-instance-names.

13.3. Атрибут диапазона

Атрибут диапазона позволяет задавать диапазон для слотов, содержащих числовые значения. Если слот, для которого задан атрибут диапазона, содержит не числовое значение, проверка на удовлетворение этого атрибута не выполняется.

Определение 13.4. Синтаксис атрибута диапазона

<атрибут-диапазона> : : = (range <граница-диапазона> <граница-диапазона>)
<граница-диапазона> ::= <число> | ? VARIABLE

Для задания границ диапазона могут использоваться числа как целого, так и вещественного типа. Первая граница диапазона определяет минимальное допустимое значение, которое может

содержать соответствующий слот, вторая граница диапазона — максимальное. Для выполнения сравнения там, где это необходимо, целые числа временно переводятся в вещественные. В случае использования ?VARIABLE в качестве первой границы диапазона, минимальным значением, которое может принимать соответствующий слот, считается отрицательная бесконечность (- ∞). В случае если ключевое слово ?VARIABLE использовалось в качестве второй границы диапазона, максимальным допустимым значением слота считается положительная бесконечность (+ ∞). Атрибут диапазона range не может быть использован вместе с константными атрибутами allowed-values, allowed-numbers, allowed-integers, или allowed-floats.

13.4. Атрибут мощности

Атрибут мощности служит для ограничения числа полей, сохраняемых в составном поле. Этот атрибут нельзя использовать с простыми полями.

Определение 13.5. Синтаксис атрибута мощности

<атрибут-мощности> ::= (cardinality <граница-мощности> <граница-мощности>)
<граница-мощности> ::= <целое-значение> | ?VARIABLE

В качестве границ мощности можно использовать только целые числа. Первое значение границы мощности определяет минимальное допустимое число полей, которые могут быть сохранены в слот, а второе значение границы мощности — максимальное число полей. В случае использования ключевого слова ?VARIABLE в качестве первой границы мощности, минимальное допустимое количество полей, сохраненных в заданный слот, будет равняться нулю. Если в качестве второй границы атрибута мощности использовалось ?VARIABLE, то максимальным допустимым числом полей слота считается положительная бесконечность (+ ∞). В случае если для составного слота не задан атрибут мощности, то по умолчанию считается, что допустимая мощность данного слота находится в диапазоне от нуля до бесконечности.

В CLIPS версии 5.1 поддерживались атрибуты min-number-of-elements и max-number-of-elements. В версии 6.0 эти атрибуты уже не поддерживаются. Вместо них необходимо применять атрибут мощности.

13.5. Получение значений по умолчанию с помощью атрибутов ограничений

Если для слотов объекта или факта значения по умолчанию явно не заданы, то эти значения могут автоматически получаться из атрибутов ограничений. Для этого используются следующие правила:

1. Тип по умолчанию выбирается из списка допустимых типов, заданных атрибутом type. При этом используется следующий порядок приоритетов типов**:** SYMBOL, STRING, INTEGER, FLOAT, INSTANCE-NAME, INSTANCE-ADDRESS, FACT-ADDRESS, EXTERNAL-ADDRESS.

2.Если тип по умолчанию имеет заданные константные ограничения (например, атрибут allowed-integers для типа INTEGER), то первое значение, заданное в константном ограничении, используется в качестве значения по умолчанию для данного слота.

3.Если значение по умолчанию нельзя определить на шаге 2, а тип слота по умолчанию INTEGER или FLOAT, и для данного слота задан атрибут диапазона, то, если нижняя граница диапазона не равна значению ?VARIABLE, это значение используется в качестве значения по умолчанию. В случае если нижняя граница диапазона равна значению ?VARIABLE, а верхняя неравна ?VARIABLE, то в качестве значения по умолчанию используется верхняя граница диапазона.

4.Если значение по умолчанию нельзя определить на шаге 2 и 3, то в качестве значения по умолчанию для слота берутся значения по умолчанию для типов. Это nil для типа SYMBOL, "" для типа STRING, 0 для INTEGER, 0.0 для FLOAT, [nil] для INSTANCE-NAME, указатель на несуществующий объект для INSTANS-ADDRESS, указатель на несуществующий факт для FACT-ADDRESS и NULL для EXTERNAL-ADDRESS.

5. Шаги 1—4 используются для определения значения по умолчанию для простых слотов. Значение по умолчанию для составных слотов — это составное значение нулевой длины.

Однако если составной слот имеет атрибут ограничения мощности, задающий минимальную мощность слота, большую 0, то в качестве значения по умолчанию для этого слота используется составное значение с количеством полей, равным минимальной мощности поля. Каждое поле при этом содержит значение по умолчанию, определенное с помощью шагов 1—4.

13.6. Примеры нарушения ограничений

В этом разделе приведено несколько примеров, иллюстрирующих некоторые типы нарушений ограничений, которые обнаруживает CLIPS.

Введите в CLIPS следующий конструктор deftemplate.

Пример 13.1. Шаблон с ограничениями

```
(deftemplate bar
             (slot a  (type SYMBOL INTEGER))
             (slot b  (type INTEGER FLOAT))
             (slot c  (type SYMBOL STRING)))
```

Теперь попробуем ввести правило, использующее созданный шаблон.

Пример 13.2. Использование шаблона с ограничениями в правиле

```
(degrule error
          (bar (a ?x) )
          (bar (b ?x) )
         (bar (c ?x))
         =>)
```

Результат выполнения этих действий приведен на рис. 13.2.

Появление переменной ?x в первом образце правила error определяет допустимые типы этой переменной по ограничениям типов для слота a. Переменная ?x может содержать значения либо типа SYMBOL, либо типа INTEGER. Использование переменной ?x во втором образце правила добавляет дополнительное ограничение на тип переменной ?x. Теперь переменная может принимать значения только типа INTEGER. Указание переменной в третьем образце правила приводит к ошибке, т. к. слот с может содержать значения либо типа SYMBOL, либо типа STRING, а переменная ? x , которая ставится в соответствие слоту с, должна содержать значения типа INTEGER.

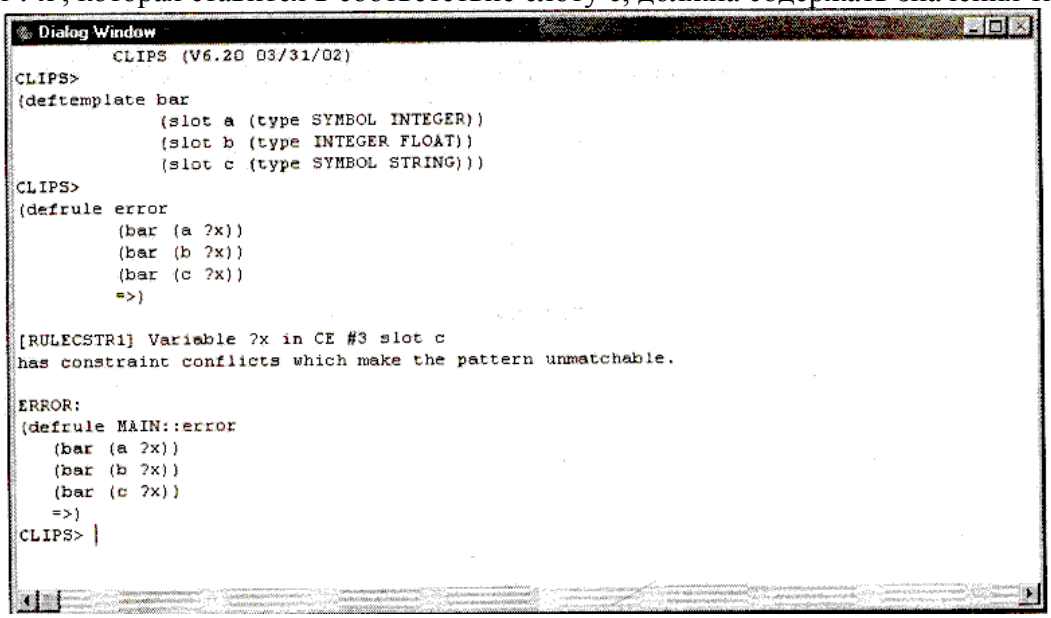

Рис. 13.2. Использование шаблона с ограничениями в правиле

Рассмотрим пример 13.3.

Пример 13.3. Еще одно ошибочное применение ограничений

```
(deftemplate foo (multislot x (cardinality ?VARIABLE 3))
(deftemplate bar (multislot y (cardinality ?VARIABLE 2))
(deftemplate woz (multislot z (cardinality 7 ?VARIABLE))
(defrule error
        (foo (x $?x))
        (bar (y $?y))
       (woz (z $?x $?y)) =>)
```

Результат выполнения этих команд приведен на рис. 13.3.

В данном случае ошибка возникает по следующим причинам. Составная переменная ?x, используемая в первом образце правила, может содержать максимум 3 поля (согласно заданным ограничениям мощности). Составная переменная ?y из второго образца правила может содержать максимум 2 поля. Объединение обеих переменных дает составную переменную, содержащую максимум 5 полей. Поскольку слот z в третьем образце правила error имеет минимальную мощность, равную 7, переменные ?x и ?y не могут удовлетворять ограничениям мощности для слота z, что вызывает соответствующую ошибку.

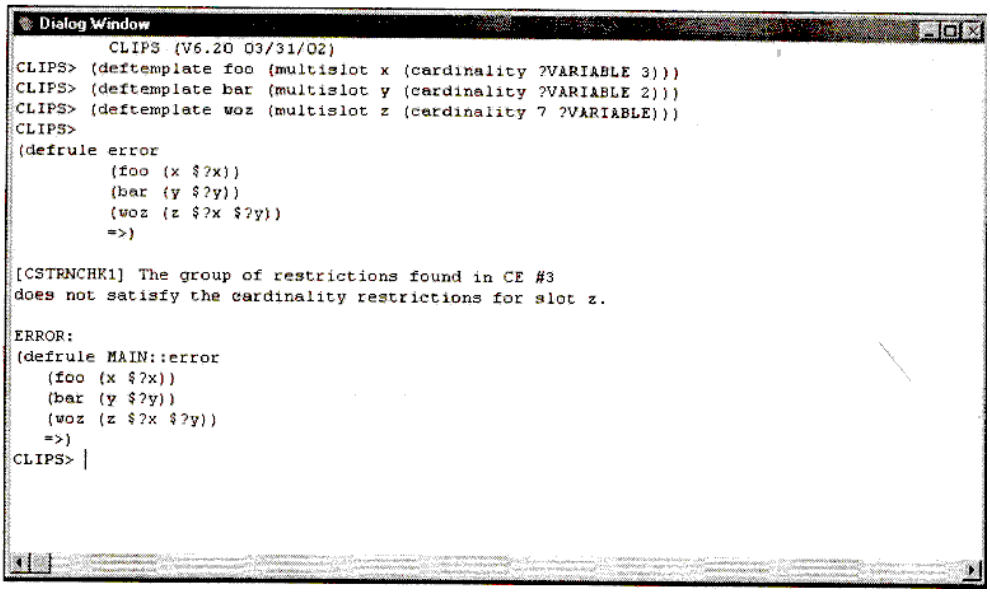

Рис. 13.3. Еще одно ошибочное применение ограничений

Введите в CLIPS следующие команды:

Пример 13.4. И еще одно ошибочное применение ограничений

```
(deftemplate foo (slot x (type SYMBOL)))
(defrule error
        (foo (x ?x))
       (test (> ?x 10)) =>)
```

Результат ввода этих команд изображен на рис. 13.4.

Переменная ?x из первого образца правила, согласно ограничениям типа для слота x, должна содержать значения типа SYMBOL. Однако функция > использует только числовые аргументы, поэтому CLIPS выводит соответствующее сообщение об ошибке.

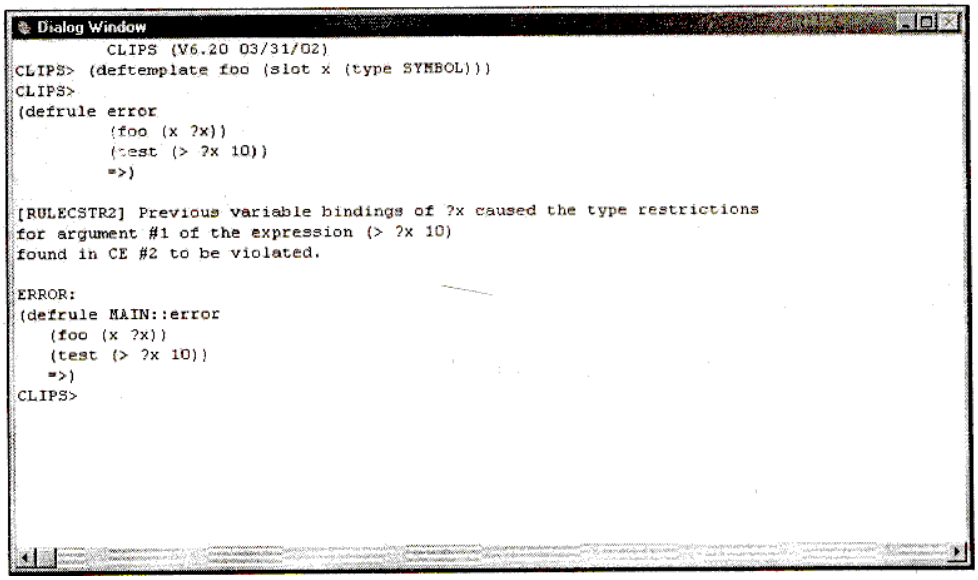

Рис. 13.4. И еще одно ошибочное применение ограничений

ГЛАВА 14. Разработка экспертной системы CIOS

Данная глава целиком посвящена примеру создания экспертной системы CIOS (Circuit Input/Output Simplification). Назначением системы CIOS является построение и оптимизация *таблиц истинности* (boolean decision table) заданных логических схем. В отличие от примера, приведенного в *гл. 9,* рассматриваемая в этой главе экспертная система будет использовать не так уж много правил. Данный пример демонстрирует, насколько эффективной может оказаться интеграция объектно-ориентированных возможностей COOL с правилами и механизмом логического вывода CLIPS. Именно наличию продуманной иерархии классов, объекты которых представляют собой элементы логической схемы, экспертная система обязана такому небольшому числу правил. Большая часть обработки информации происходит *в* объектах благодаря использованию обработчиков сообщений разных типов.

Помимо объектно-ориентированных возможностей CLIPS система CIOS использует также родовые функции, которые превосходно подошли для связывания логических элементов разных типов.

Большинство конструкций языка, используемых при создании экспертной системы CIOS, уже были рассмотрены в предыдущих главах. Однако в данной главе все-таки встречаются еще не рассмотренные функции языка CLIPS. Краткое описание назначения таких функций будет приводиться по мере необходимости. Полное описание большинства таких функций можно будет найти в *гл. 15.*

14.1. Постановка задачи

Основной задачей, для решения которой предназначена экспертная система CIOS, является построение и оптимизация таблиц истинности заданных логических схем. Логической схемой будем называть объединение некоторых примитивных логических элементов. Полный список и описание логических элементов, используемых в данном примере, приведены в табл. 14.1.

Таблица 14.1. *Примитивные логические элементы*

Логический элемент	Описание
SOURCE	Источник— элемент, представляющий собой вход логической схемы. Имеет один выход и не имеет входов. Считается, что первоначальный сигнал появляется в этом элементе и сразу передается следующему логическому элементу
LED	Индикатор — элемент, моделирующий выход логической схемы. Является противоположностью источнику, имеет один вход и не имеет выходов. Считается, что при получении сигнала индикатор просто отображает его
SPLITTER	Разделитель. Данный элемент предназначен для деления сигнала. Имеет один вход и два логических выхода. Полученный сигнал без изменения передается на оба выхода разделителя
NOT	Логическое НЕ. Имеет один вход и один выход. Меняет полученный сигнал на противоположный
AND	Логическое И. Имеет два входа и один выход. Возвращает 1, если оба полученных сигнала равны 1. В противном случае возвращает 0
OR	Логическое ИЛИ. Имеет два входа и один выход. Возвращает 1, если хотя бы один из полученных сигналов равен 1. В противном случае возвращает 0

NAND	Отрицание логического И. Имеет два входа и один выход. Возвращает 0, если оба полученных сигнала равны 1. В противном случае возвращает 1
XOR	Исключающее ИЛИ. Имеет два входа и один выход. Возвращает 1, если полученные сигналы не равны. В противном случае возвращает 0

При составлении логической схемы из примитивных элементов обязательными являются следующие правила:

- выход каждого логического элемента связан с одним и только одним входом другого элемента;
- вход каждого логического элемента связан с одним и только одним выходом другого элемента;
- исходные сигналы подаются на источники логической схемы, не имеющие входов;
- исходный сигнал перемещается по логической схеме, за условную единицу времени — *такт;*
- по завершении такта обработанный сигнал оказывается на логических индикаторах схемы.

Таблицей истинности называется таблица, содержащая всевозможные наборы сигналов, принимаемые источниками логической схемы, и обработанный результат, полученный на индикаторах схемы для каждого набора. Например, для простейшей логической схемы, приведенной на рис. 14.1, таблица истинности представлена в табл. 14.2.

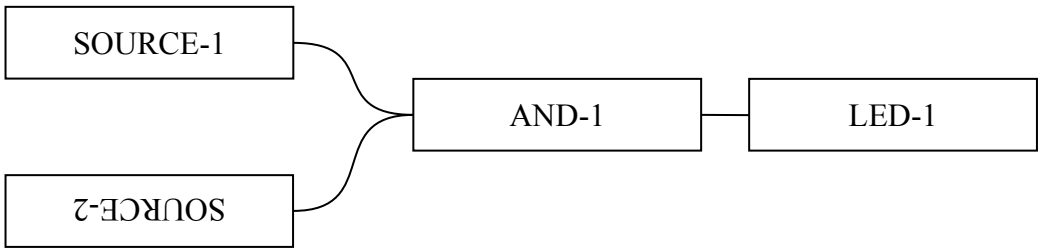

Рис. 14.1. Простейшая логическая схема

Таблица 14.2. *Пример простейшей таблицы истинности*

SOURCE-1	SOURCE-2	LED-1
0	0	0
0	1	0
1	0	0
1	1	1

Количество столбцов таблицы истинности очевидно равно числу источников и индикаторов схемы. Количество строк — числу 2, возведенному в степень числа источников, т. е. при одном источнике $2^1 = 2$, при двух $2^2 = 4$, при трех $2^3 = 8$ и т. д. Таким образом, таблица истинности может принимать весьма внушительные размеры. Уже при 6-ти источниках количество строк в ней равняется 64.

Очевидно, что возможны ситуации, когда изменение значения одного из входов не влияет на результат при постоянных значениях остальных входах. В этом случае таблицу можно оптимизировать, объединив такие строчки и заменив значение невлияющего на результат входа на символ *. В представленном выше примере можно объединить первую и вторую строку таблицы, т. к. при любом значении, подаваемом на источник SOURCE-2 в случае, если на источник SOURCE-1 подается 0, результат будет равен 0. Таким образом, оптимизированная таблица примет вид, приведенный в табл. 14.3.

Таблица 14.2. *Оптимизированная таблица истинности*

SOURCE-1	SOURCE-2	LED-1
0	*	0
1	0	0
1	1	1

В данном примере преимущества оптимизированных таблиц истинности не очень заметно. Однако при составлении таких таблиц для достаточно больших логических схем преимущества очевидны. Эффект, полученный от оптимизации таблицы истинности, сильно зависит от конкретной анализируемой логической схемы.

Следует отметить, что оптимизация таблиц истинности неоднозначна. Но конечный результат (количество строк в итоговой таблице) всегда одинаков. Наш простой пример можно было оптимизировать так, как представлено в табл. 14.4.

Таблица 14.4. *Еще один пример оптимизированной таблицы истинности*

SOURCE-1	SOURCE-2	LED-1
*	0	0
0	1	0
1	1	1

Для разработки и тестирования экспертной системы CIOS будем использовать логические схемы, приведенные на рис. 14.2—14.4. Сокращения даны в табл. 14.5.

Таблица 14.5. *Сокращения названий логических элементов*

Название логического элемента	Используемое сокращение
SOURCE	S
LED	L
SPLITTER	P
NOT	N
AND	A
OR	O
NAND	D
XOR	X

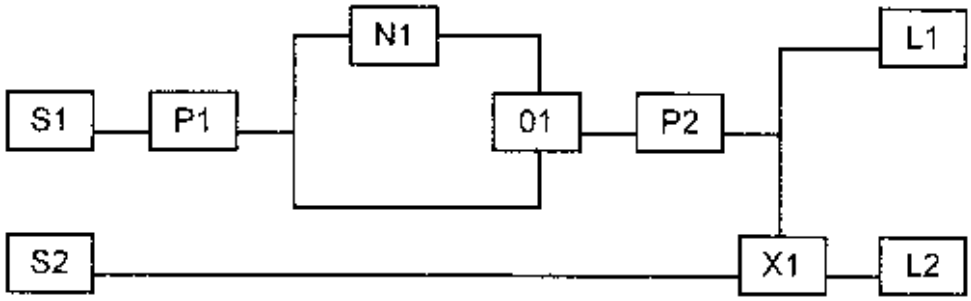

Рис. 14.2. Схема 1

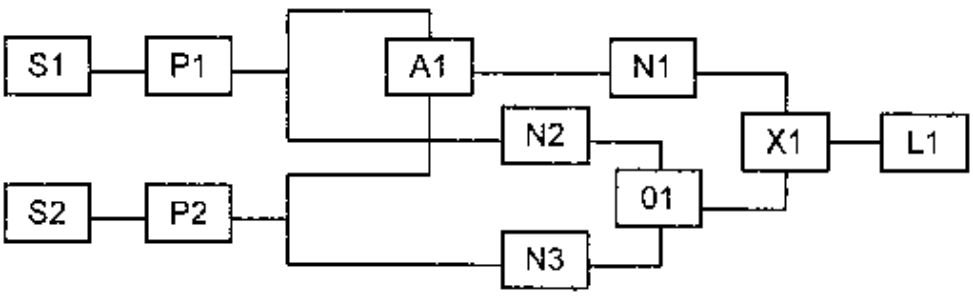

Рис. 14.3. Схема 2

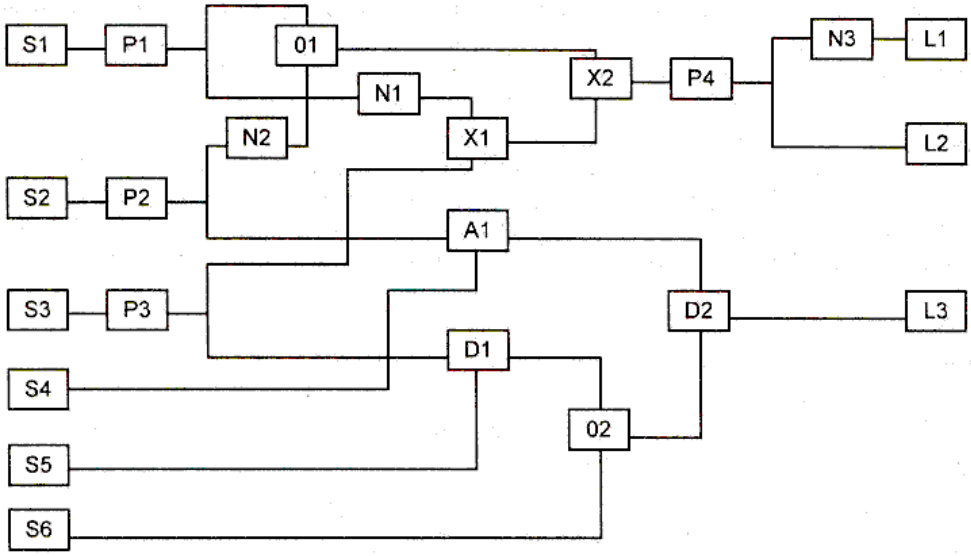

Рис. 14.4. Схема 3

14.2. Алгоритм решения задачи

Для решения поставленной задачи построения и оптимизации таблиц истинности логических схем будем использовать следующий алгоритм.

1.Инициализация заданной логической схемы и установление необходимых соединений между ее элементами.

2.Получение результата работы логической схемы, в случае если на все ее источники подаются значения, равные 0.

3.Далее, последовательно перебираются всевозможные комбинации входных сигналов для всех источников логической схемы и вычисляются результаты ее работы. Перебор всевозможных комбинаций входных сигналов осуществляется с помощью циклического (рефлексного) кода Грея (binary cyclic Gray code). Этот код является особым методом представления двоичных чисел, в котором каждое последующее число отличается от предыдущего только на один символ. Например, коды Грея для чисел от 0 до 7 будут: 0 = 000, 1 = 001, 2 = 011, 3 = 010, 4 = 110, 5 = 111, 6 = 101, 7 = 100. Использование кода Грея позволяет значительно оптимизировать работу экспертной системы по времени исполнения, т. к. для получения результата обработки каждого следующего варианта входных сигналов необходимо изменить значение только одного источника.

4.В процессе получения результатов работы логической схемы для всевозможных комбинаций входных сигналов экспертная система производит поиск двух строк таблицы, которые можно объединить согласно алгоритму оптимизации, приведенному в *разд. 14.1.*

5.После получения результатов работы логической схемы для всех комбинаций входных сигналов и их оптимизации экспертная система выводит на экран найденную таблицу истинности.

14.3. Представление логических элементов

При анализе заданной логической схемы огромное значение имеет способ представления ее элементов в рабочей памяти экспертной системы. Например, от способа представления элементов зависят методы и алгоритмы, используемые для вычисления результатов работы логической схемы при заданном наборе входных сигналов. Для того чтобы при разработке правил экспертной системы сконцентрироваться на алгоритме, приведенном в *разд. 14.2,* воспользуемся объектно-ориентированными возможностями, предоставляемыми системой CLIPS.

Для представления каждого логического элемента создадим соответствующий класс, объект которого будет самостоятельно выполнять обработку поступившего сигнала и передавать полученный результат на вход элемента, соединенного с выходом. Составленная таким образом логическая схема будет самостоятельно вычислять результаты своей работы при попадании соответствующих сигналов на вход источников.

При реализации классов, представляющих собой логические элементы, воспользуемся возможностью множественного наследования. Заметим, что каждый логический элемент

является компонентом, обладающим некоторым числом входов (от 0 до 2) и выходов (от 0 до 2). Основываясь на этом замечании, создадим набор классов и обработчиков сообщений.

Пример 14.1. Классы, используемые для создания логических элементов

```
(defclass COMPONENT
            (is-a USER)
            (slot ID#  (create-accessor write))
          )
 (defclass NO-OUTPUT
            (is-a USER)
            (slot number-of-outputs (access read-only)
                                    (default 0)
                                    (create-accessor read)) )
(defmessage-handler NO-OUTPUT compute-output ( ) )
(defclass ONE-OUTPUT
          (is-a NO-OUTPUT)
          (slot number-of-outputs (access read-only)
                                  (default 1)
                                  (create-accessor read))
          (slot output-1 (default UNDEFINED)
                         (create-accessor write))
          (slot output-1-link (default GROUND)
                              (create-accessor write))
          (slot output-1-link-pin (default 1)
                                  (create-accessor write))
      )
 (defmessage-handler ONE-OUTPUT put-output-1 after (?value)
                        (send ?self:output-1-link
                              (sym-cat put-input- ?self:output-1-link-pin)
                              ?value)
            )
  (defclass TWO-OUTPUT
            (is-a ONE-OUTPUT)
            (slot number-of-outputs (access read-only)
                                    (default 2)
                                    (create-accessor read))

          (slot output-2 (default UNDEFINED)
                         (create-accessor write))
          (slot output-2-link (default GROUND)
                              (create-accessor write))
          (slot output-2-link-pin (default 1)
                                  (create-accessor write))
        )
(defmessage-handler TWO-OUTPUT put-output-2 after (?value)
                        (send ?self:output-2-link
                              (sym-cat put-input- ?self:output-2-link-pin)
                              ?value)
        )
(defclass NO-INPUT
          (is-a USER)
          (slot number-of-inputs (access read-only)
                                 (default 0)
                                 (create-accessor read)) )
(defclass ONE-INPUT
          (is-a NO-INPUT)
          (slot number-of-inputs (access read-only)
                                 (default 1)
                                 (create-accessor read))
           (slot input-1 (default UNDEFINED)
                         (visibility public)
                         (create-accessor read-write))
           (slot input-1-link (default GROUND)
```

```
                                (create-accessor write))
            (slot input-1-link-pin (default 1)
                                (create-accessor write)) )
(defmessage-handler ONE-INPUT put-input-1 after (?value)
                        (send ?self compute-output)
                    )
(defclass TWO-INPUT
            (is-a ONE-INPUT)
            (slot number-of-inputs (access read-only)
                                (default 2)
                                (create-accessor read))
             (slot input-2 (default UNDEFINED)
                        (visibility public)
                        (create-accessor write))
             (slot input-2-link (default GROUND)
                                (create-accessor write))
            (slot  input-2-link-pin   (default  1)
                                 (create-accessor write))
          )
(defmessage-handler TWO-INPUT put-input-2  after   (?value)
                        (send  ?self compute-output)
          )
```

Класс COMPONENT является абстрактным потомком класса USER. Этот класс содержит единственный слот ID# для идентификационного номера компонента и определяет для него акцессор записи. Класс будет являться суперклассом для всех классов логических элементов.

Набор классов NO-OUTPUT, ONE-OUTPUT и TWO-OUTPUT предназначен для реализации свойств элемента без выхода, с одним выходом и двумя выходами соответственно. Каждый следующий класс наследуется от предыдущего и добавляет к нему реализацию одного логического выхода.

Класс NO-OUTPUT имеет единственный слот number-of-outputs со значением по умолчанию 0 и доступен только для чтения. Этот слот предназначен для хранения числа выходов элемента, унаследованного от этого класса. Кроме того, класс NO-OUTPUT определяет обработчик compute-output, который унаследует и переопределит все логические элементы. Именно благодаря наличию этого обработчика объекты классов логических элементов смогут самостоятельно определять результат обработки полученных сигналов.

Класс ONE-OUTPUT переопределяет слот number-of-outputs и назначает ему значение по умолчанию, равное 1. Кроме того, класс определяет набор слотов, предназначенных для описания логического выхода элемента, и связи этого выхода с последующими элементами схемы. Слот output-1 предназначен для хранения значения, образующегося на выходе элемента. По умолчанию слот имеет значение UNDEFINED. Слот output-1-link содержит имя объекта, с которым связан данный выход, и по умолчанию имеет значение GROUND. Слот output-1-link-pin содержит номер конкретного логического входа элемента, с которым связан данный выход, и по умолчанию равен 1. Класс ONE-OUTPUT определяет after-обработчик put-outout-1, который сразу после присвоения значения слоту output-1 передает это значение на вход элемента, связанного с данным выходом.

Класс TWO-OUTPUT тоже переопределяет слот number-of-outputs и присваивает ему значение по умолчанию, равное 2, а также добавляет набор слотов, предназначенных для описания второго логического выхода: output-2, output-2-link и output-2-link-pin и after-обработчик put-outout-2. Следует обратить внимание на тот факт, что слоты output-1, output-1-link и output-l-link-pin и обработчик put-outout-1 не переопределяются, а наследуются от класса ONE-OUTPUT.

Набор классов NO-INPUT, ONE-INPUT и TWO-INPUT предназначен для реализации свойств элемента без входа, с одним входом и двумя входами соответственно. Каждый следующий класс наследуется от предыдущего и добавляет к нему реализацию еще одного логического входа. Назначение слотов, описывающих логический вход элементов input-x, input-x-link, input-x-link-pin, идентично назначению слотов, использующихся в классах ONE-OUTPUT и TWO-OUTPUT. Кардинальное отличие классов, реализующих логику входов элемента, заключается в том, что вместо обработчика put-outout-x классы реализуют after-обработчик put-inout-x. Этот обработчик вызывает обработчик сообщения compute-output сразу после получения нового значения.

После определения классов COMPONENT, NO-OUTPUT, ONE-OUTPUT, TWO-OUTPUT, NO-INPUT, ONE-INPUT и TWO-INPUT можно приступать к созданию классов, реализующих логику отдельных элементов схемы. Начнем с простейших по своей функциональности элементов — SOURCE и LED.

Пример 14.2. Классы логических элементов SOURCE и LED

```
(defclass SOURCE
          (is-a NO-INPUT ONE-OUTPUT COMPONENT)
          (role concrete)
          (slot output-1 (default UNDEFINED)
                         (create-accessor write))
       )

(defclass LED
          (is-a ONE-INPUT NO-OUTPUT COMPONENT)
          (role concrete)
       )
```

Класс SOURCE реализует поведение источника. Этот логический элемент имеет один выход и не имеет входов. Класс LED является реализацией логического индикатора. Он имеет один вход и не имеет выходов. В дальнейшем при создании экспертной системы многократно придется решать проблему сбора результата обработки входных сигналов со всех индикаторов логической схемы. Для решения этой задачи предназначена следующая функция:

Пример 14.3. Функция LED-response

```
(deffunction LED-response   ( )
                (bind  ?response   (create$))
                (do-for-all-instances   ((?ledLED))   TRUE
                        (bind  ?response   (create$   ?response
                                                      (send  ?led get-input-1))))
                ?response
             )
```

Эта функция собирает составное поле из значений, хранящихся во всех объектах класса LED текущей логической схемы, с помощью функции do-for-all-instances.

Приступим к реализации более сложных логических элементов. Их реализация будет напоминать классы SOURCE и LED, за исключением того, что остальные логические элементы должны переопределять обработчик compute-output, который будет производить обработку сигнала, полученного на входе элемента.

Пример 14.4. Класс NOT-GATE

```
(defclass NOT-GATE
          (is-a ONE-INPUT ONE-OUTPUT COMPONENT)
          (role concrete)
         )
(deffunction not # (?x) (- 1 ?x) )
(defmessage-handler NOT-GATE compute-output ()
                        (if (integerp ?self : input-1) then
                           (send ?self put-output-1 (not # ?self: input-1) ))
                    )
```

Обратите внимание, что в реализации обработчика compute-output используется функция not#, которая и выполняет необходимые вычисления. Определение класса NOT-GATE чрезвычайно компактно, однако, благодаря использованию продуманной иерархии классов и множественного наследования, класс обладает всей необходимой функциональностью. Классы остальных логических элементов реализуем аналогично.

Пример 14.5. Классы остальных логических элементов

```
(defclass AND-GATE
        (is-a TWO-INPUT ONE-OUTPUT COMPONENT)
        (role concrete)
    )
(deffunction andt (?x ?y)
            (if (and ( ! = ?x 0) ( ! = ?y 0 ) ) then 1 else 0 ) )
(defmessage-handler AND-GATE compute-output ()
                    (if (and  (integerp ?self: input-1)
                              (integerp ?self : input-2) ) then
                        (send ?self put-output-1
                              (and# ?self: input-1 ?self: input-2))
                    )

(defclass OR- GATE
        (is-a TWO- INPUT ONE-OUTPUT COMPONENT)
        (role concrete)
    )
(deffunction or# (?x ?y)
            (if (or ( ! = ?x 0) ( ! - ?y 0 ) ) then 1 else 0 ) )
(defmessage-handler OR-GATE compute-output ()
                    (if (and (integerp ?self : input- 1)
                             (integerp ?self : input-2) ) then
                        (send ?self put-output-1
                              (or# ?self :input-l ?self: input-2) ))
                )

(defclass NAND-GATE
        (is-a TWO- INPUT ONE-OUTPUT COMPONENT)
        (role concrete)
    )
(deffunction nand# (?x ?y)
            (if (not (and ( ! = ?x 0) ( ! = ?y 0 ) ) ) then 1 else 0 ) )
(defmessage-handler NAND-GATE compute-output ( )
                                (if (and (integerp ?self : input-1)
                                         (integerp ?self: input-2) ) then
                                         (send ?self put-output-1
                                               (nand# ?self : input-1 ?self: input-2)))
                    )
(defclass XOR-GATE
        (is-a TWO-INPUT ONE-OUTPUT COMPONENT)
        (role concrete)
    )
(deffunction xor# (?x ?y)
            (if (or (and (= ?x 1) (= ?y 0 ) )
                    (and (= ?x 0) (= ?y 1 ) ) )  then 1 else 0 ) )
(defmessage-handler XOR-GATE compute-output ()
                    (if (and (integerp ?self: input-1)
                             (integerp ?self: input-2) ) then
                        (send ?self put-output-1
                              (xor# ?self :input-l ?self: input-2) ))
        )
(defclass SPLITTER
        (is-a ONE-INPUT TWO-OUTPUT COMPONENT)
        (role concrete)
    )
(defmessage-handler SPLITTER compute-output ()
                                (if (integerp ?self: input-1) then
                    (send  ?self put-output-1   ?self:input-1)
                    (send  ?self put-output-2   ?self:input-1))
                )
```

На этом разработку классов, необходимых для представления логических элементов, можно считать завершенной.

14.4. Связь логических элементов

После создания набора классов, необходимого для представления всех используемых логических элементов, встает вопрос объединения этих элементов в логическую схему.

Для ввода в систему набора элементов, необходимых для создания заданной логической схемы, будем использовать отдельный файл с конструктором definstances. Ниже приведен пример такого конструктора для схемы, изображенной на рис. 14.2.

Пример 14.6. Набор объектов логических элементов

```
(definstances circuit
              (S-l of SOURCE)
              (S-2 of SOURCE)
              (P-l of SPLITTER)
              (P-2 of SPLITTER)
              (N-l of NOT-GATE)
              (O-1 of OR-GATE)
              (X-l of XOR-GATE)
              (L-l of LED)
              (L-2 of LED))
```

Для связи этих элементов между собой можно разработать несколько функций, для каждого типа соединений. Однако гораздо лучшим решением в данном случае будет создание родовой функции с методами, учитывающими различие соединяемых объектов, полученных в качестве параметров.

Пример 14.7. Родовая функция connect

```
(defgeneric  connect)
(defmethod connect   ((?out ONE-OUTPUT)    (?in ONE-INPUT))
            (send  ?out put-output-1-link  ?in)
            (send ?out put-output-1-link-pin 1)
            (send  ?in    put-input-1-link  ?out)
            (send ?in    put-input-1-link-pin 1)
        )
(defmethod connect ((?out ONE-OUTPUT) (?in TWO-INPUT) (?in-pin INTEGER))
             (send ?out put-output-1-link ?in)
             (send ?out put-output-1-link-pin ?in-pin)
             (send ?in  (sym-cat put-input- ?in-pin -link) ?out)
             (send ?in  (sym-cat put-input- ?in-pin -link-pin) 1)
        )
(defmethod connect ((?out TWO-OUTPUT) (?out-pin INTEGER) (?in ONE-INPUT))
            (send ?out (sym-cat put-output- ?out-pin -link) ?in)
            (send ?out (sym-cat put-output- ?out-pin -link-pin) 1)
            (send ?in put-input-1-link ?out)
            (send ?in put-input-1-link-pin ?out-pin)
      )
(defmethod connect ((?out TWO-OUTPUT) (?out-pin INTEGER)
                          (?in TWO-INPUT) (?in-pin INTEGER))
            (send ?out (sym-cat put-output- ?out-pin -link) ?in)
            (send ?out (sym-cat put-output- ?out-pin -link-pin) ?in-pin)
            (send ?in  (sym-cat put-input- ?in-pin -link) ?out)
            (send ?in  (sym-cat put-input- ?in-pin -link-pin) ?out-pin)
      )
```

Приведенные выше методы родовой функции connect учитывают все возможные типы соединений логических аргументов. Используя эту родовую функцию, можно написать простую процедуру инициализации логической схемы, приведенной на рис. 14.2.

Пример 14.8. Инициализация логической схемы

```
(deffunction connect-circuit ()
            (connect [S-1]  [P-1])
            (connect [S-2]  [X-1]  2)
            (connect [P-1]  1 [N-1])
            (connect [P-1]  2  [O-1]  2)
            (connect N-1]  [O-1]  1)
            (connect [O-1]  [P-2])
            (connect [P-2]  1  [L-1])
            (connect [P-2]  2  [X-1]  1)
            (connect [X-1]  [L-2])
      )
```

Применение описанного выше метода связывания элементов в логическую схему позволит разделить данные и методы их обработки. Каждая отдельная схема может быть расположена в отдельном файле и подгружаться к экспертной системе с помощью команды load.

14.5. Дополнительные функции и переменные

Для реализации логики экспертной системы CIOS, помимо описанных выше классов, объектов, функций и родовых функций, понадобятся также некоторые дополнительные элементы.

Прежде всего, добавим в систему следующие глобальные переменные:

Пример 14.9. Необходимые глобальные переменные

```
(defglobal ?*gray-code*       = (create$)
           ?*sources*         = (create$)
           ?*max-iterations* = 0)
```

Переменная max-iterations служит для хранения максимального числа итераций при переборе всевозможных комбинаций входных сигналов системы. В переменной sources хранятся имена всех источников текущей логической схемы. Переменная gray-code предназначена для хранения текущего кода Грея.

Ниже приведена вспомогательная функция, определяющая номер элемента в коде Грея, значение которого необходимо изменить для перебора всех возможных вариантов входных сигналов.

Пример 14.10. Функция change-which-bit

```
(deffunction change-which-bit  (?x)
            (bind  ?i  1)
            (while  (and  (evenp  ?x)  (!= ?x  0))  do
                    (bind  ?x  (div  ?x  2))
                    (bind  ?i  (+  ?i  1))
                )
              ?i
            )
```

Кроме того, поскольку фактическое определение функции, производящей связь логических элементов в схему, осуществляется в отдельном файле, необходимо предварительное объявление этой функции.

Пример 14.11. Предварительное объявление функции connect-circuit

```
(deffunction connect-circuit ())
```

14.6. Реализация правил экспертной системы

После добавления в CLIPS всех перечисленных определений и конструкций, можно приступить к созданию правил, реализующих работу алгоритма, описанного в *разд. 14.2*.

Начнем с реализации правила, инициализирующего только что загруженную логическую схему.

Пример 14.12. Правило startup

```
(defrule startup
        =>
        ( connect-circuit )
        (bind ?*sources* (find-all-instances ((?x SOURCE)) TRUE))
        (do-for-all-instances ( (?x SOURCE)) TRUE
                (bind ?*gray-code* (create$ ?*gray-code* 0) ) )
        (bind ?*max-iterations* (round (** 2 (length ?*sources*) ) ) )
        (assert (current-iteration 0) )
    )
```

Правило startup не имеет явных условных элементов, поэтому, как было описано в *гл. 6,* правило будет активизировано первым фактом initial-fact. Сразу после запуска правила вызывается функция connect-circuit, инициализирующая текущую логическую схему. После этого правило получает имена всех элементов источников в переменную sources. Затем правило startup создает нулевой код Грея для начала процесса перебора всех вариантов входных сигналов, вычисляет максимальное количество итераций и добавляет факт с номером текущей итерации.

Теперь необходимо запустить итерационный процесс перебора комбинаций входных сигналов. Для этого служит правило compute-response-1st-time.

Пример 14.13. Правило compute-response-1st-time

```
(defrule compute-response-1st-time
        ?f <- (current-iteration 0)
        =>
        (do-for-all-instances ((?source SOURCE)) TRUE
                        (send ?source put-output-1 0))
        (assert (result ?*gray-code* =(str-implode (LED-response))))
        (retract ?f)
        (assert (current-iteration 1)
    )
```

Это правило предназначено для первого шага перебора всевозможных входных сигналов. Оно передает нулевой сигнал всем источникам логической схемы, что вызывает автоматическое вычисление результата. Далее полученный результат сохраняется в факте вида:

(result текущее-состояние-источников текущее-состояние-индикаторов)

После этого правило удаляет факт (current-iteration 0), который и запускает правило и добавляет факт (current-iteration 1).

Пример 14.14. Правило compute-response-other-times

```
(defrule compute- response-other-times
        ?f <- (current-iteration ?n&~0&:(< ?n ?*max-iterations*) )
        =>
        (bind ?pos (change-which-bit ?n) )
        (bind ?nv) (- 1 (nth ?pos ?*gray-code*) ) )
        (bind ?*gray-code* (replace$ ?*gray-code* ?pos ?pos ?nv) )
        (send (nth ?pos ?*sources*) put-output-1 ?nv)
        (assert (result ?*gray-code* = (str-implode (LED-response) ) ) )
        (retract ?f)
        (assert (current-iteration =(+ ?n 1)))
    )
```

Логика работы правила compute-response-other-times подобна работе правила compute-response-1st-time, за исключением того, что в начале с помощью вызова функции change-which-bit оно определяет, какой из источников схемы должен поменять входящий сигнал, и меняет его. После этого правило получает результат работы логической схемы и сохраняет его в соответствующем факте. Кроме того, правило увеличивает число итераций на единицу. Правило выполняется до тех пор, пока число итераций не достигнет предельного значения. Еще одним условием выполнения правила является то, что одна итерация уже была совершена (число итераций не равно 0).

Пример 14.15. Правило marge-responses

```
(defrule merge-responses
        (declare  (salience 10))
        ?fl <-  (result $?b   ?x  $?e  ?response)
        ?f2 <-  (result  $?b  ~?x  $?e  ?response)
        =>
        (retract  ?fl  ?f2)
        (assert (result ?b * ?e ?response))
    )
```

Правило merge-responses предназначено для оптимизации вычисляемой таблицы истинности и имеет более высокий приоритет, поэтому выполняется сразу после того, как в системе появятся два факта, удовлетворяющие заданной маске.

Пример 14.16. Правило print-header

```
(defrule print-header
        (declare (salience -10) )
        =>
        (do-for-all-instances ((?x SOURCE)) TRUE
                              (format t " %3s " (sym-cat ?x)))
        (printout t " | ")
        (do-for-all-instances ((?x LED)) TRUE
                              (format t " %3s " (sym-cat ?x)))
        (format t "%n")
        (do-for-all-instances ((?x SOURCE)) TRUE
                              (printout t "-------") )
        (printout t "- + -")
        (do-for-all-instances ((?x LED)) TRUE
                              (printout t "-------"))
        (format t "%n")
        (assert (print-results))
    )
```

Приведенное выше правило print-header предназначено для вывода на экран заголовка таблицы истинности. Затем правило добавляет в систему факт print-results, активизирующий правило print-result. Заголовок содержит список всех источников системы и список индикаторов, разделенных вертикальной чертой. Кроме того, для большего удобства восприятия правило отделяет заголовок таблицы от ее содержания дополнительной строкой. Это правило имеет более низкий приоритет, чем остальные правила экспертной системы, и не имеет явных условных элементов. Поэтому выполняется только после завершения перебора всевозможных комбинаций входных сигналов логической схемы.

Пример 14.17. Правило print-result

```
(defrule print-result
        (print-results)
        ?f <- (result $?input ?response)
        (not (result $?input-2 ?response-2&:
                        (< (str-compare ?response-2 ?response) 0) ) )
        =>
        (retract ?f)
```

```
    (while (neq ?input (create$)) do
           (printout t "  " (nth 1 ?input) "  ")
           (bind ?input (rest$ ?input)))
    (printout t " | ")
    (bind ?response (str-explode ?response))
    (while (neq ?response (create$)) do
           (printout t "  " (nth 1 ?response) "  ")
           (bind ?response (rest$ ?response)))
    (printout t crlf)
)
```

Правило print-result выводит на экран оптимизированную таблицу истинности, сортируя при этом ее строки.

14.7. Листинг программы

Разработку экспертной системы CIOS можно считать завершенной. Данный раздел содержит полный листинг программы с подробными комментариями. Если у вас еще не сложилась целостная картина, как работает экспертная система CIOS, из каких частей она состоит, внимательно изучите приведенный код.

Пример 14.18. Полный листинг программы

```
;=========================================================
;   Пример экспертной системы на языке CLIPS
;
;   Приведенная ниже экспертная система способна находить
;   и оптимизировать таблицы истинности заданных логических схем.
;
;=========================================================
;   Необходимые классы
;=========================================================
;---------------------------------------------------------
;   Класс COMPONENT является суперклассом для всех классов логических элементов
(defclass COMPONENT
        (is-a USER)
        (slot ID# (create-accessor write))
    )
;---------------------------------------------------------
;   Класс NO-OUTPUT реализует логику работы элемента без логических выходов
(defclass NO-OUTPUT
        (is-a USER)
        (slot number-of-outputs (access read-only)
                                (default 0)
                                (create-accessor read))
    )
;   Предварительное объявление обработчика, осуществляющего обработку полученного сигнала
(defmessage-handler NO-OUTPUT compute-output ( ) )
;---------------------------------------------------------
;   Класс ONE-OUTPUT реализует логику работы элемента с одним логическим выходом
(defclass ONE-OUTPUT
        (is-a NO-OUTPUT)
        (slot number-of-outputs (access read-only)
                                (default 1)
                                (create-accessor read))
       ; значение выхода
       (slot output-1 (default UNDEFINED)
                      (create-accessor write))
```

```
        ; название элемента, с которым связан выход
        (slot output-1-link (default GROUND)
                            (create-accessor write))
        ; номер входа, с которым связан выход
        (slot output-1-link-pin (default 1)
                                (create-accessor write))
        )
;   Обработчик для передачи обработанного сигнала на вход следующего элемента
(defmessage-handler ONE-OUTPUT put-output-1 after (?value)
                    (send ?self:output-1-link
                          (sym-cat put-input- ?self:output-l-link-pin)
                          ?value)
                )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;   Класс TWO-OUTPUT реализует логику работы элемента с двумя логическими выходами
(defclass TWO-OUTPUT
          (is-a ONE-OUTPUT)
          (slot  number-of-outputs   (access  read-only)
                                     (default 2)
                                     (create-accessor read) )
          ; значение выхода
          (slot output-2 (default UNDEFINED)
                         (create-accessor write) )
         ; название элемента, с которым связан выход
         (slot output-2-link (default GROUND)
                             (create-accessor write) )
          ; номер входа, с которым связан выход
          (slot output-2-link-pin (default 1)
                                  (create-accessor write) )
        )
;   Обработчик для передачи обработанного сигнала на вход следующего элемента
(defmessage-handler TWO-OUTPUT put-output-2 after (?value)
                    (send ?self: output-2-link
                          (sym-cat put-input- ?self: output-2-link-pin)
                          ?value)
               )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;   Класс NO-INPUT реализует логику работы элемента без логических входов
(defclass NO-INPUT
          (is-a USER)
          (slot number-of-inputs (access read-only)
                                 (default 0)
                                 (create-accessor read) )
       )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;   Класс ONE-INPUT реализует логику работы элемента с одним логическим входом
(defclass ONE-INPUT
          (is-a NO- INPUT)
          (slot number-of-inputs (access read-only)
                                 (default 1)
                                 (create-accessor read) )
           ; значение входа
           (slot input-1 (default UNDEFINED)
                         (visibility public)
                         (create-accessor read-write) )
         ; название элемента, с которым связан вход
         (slot input-1-link (default GROUND)
                            (create-accessor write) )
         ;номер выхода, с которым связан вход
         (slot input-1-link-pin (default 1)
                                (create-accessor write)) )
```

```
;    Обработчик, активизирующий процесс вычисления результата работы схемы
;    после изменения данного входа
(defmessage-handler ONE-INPUT put-input-1 after (?value)
                           (send ?self compute-output)
                  )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    Класс TWO-INPUT реализует логику работы элемента с двумя логическими входами
(defclass TWO-INPUT
              (is-a ONE-INPUT)
              (slot number-of-inputs (access read-only)
                                          (default 2}
                                          (create-accessor read))
             ; значение входа
              (slot input-2 (default UNDEFINED)
                             (visibility public)
                             (create-accessor write))
            ; название элемента, с которым связан вход
            (slot input-2-link (default GROUND)
                                (create-accessor write))
             ; номер выхода, с которым связан вход
                                      (slot input-2-link-pin (default 1)
                                                            (create-accessor write))
         )
;    Обработчик, активизирующий процесс вычисления результата работы схемы
;    после изменения данного входа
(defmessage-handler TWO-INPUT put-input-2 after (?value)
                           (send ?self compute-output)
                )
;= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
;    Классы, реализующие логические элементы
;= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    Класс, реализующий логику работы элемента SOURCE, имеет один выход и не имеет входов
(defclass SOURCE
              (is-a NO-INPUT ONE-OUTPUT COMPONENT)
              (role concrete)
              (slot output-1 (default UNDEFINED)
                               (create-accessor write))
         )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    Класс, реализующий логику работы элемента LED, имеет один вход и не имеет выходов
(defclass LED
             (is-a ONE-INPUT NO-OUTPUT COMPONENT)
             (role concrete)
        )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    Класс, реализующий логику работы элемента NOT, имеет один вход и один выход
(defclass NOT-GATE
            (is-a ONE-INPUT ONE-OUTPUT COMPONENT)
            (role concrete)
        )
;    Функция, вычисляющая значение элемента NOT в зависимости от полученного аргумента
(deffunctiori not# (?x) (- 1 ?x) )
;    Обработчик, выполняющий вычисления элемента NOT при изменении входных сигналов
(defmessage-handler NOT-GATE compute-output ()
                                          (if (integerp ?self:input-1) then
                                             (send ?self put-output-1 (not# ?self:input-1)))
                )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
;    Класс, реализующий логику работы элемента AND, имеет два входа и один выход
 (defclass AND-GATE
            (is-a TWO-INPUT ONE-OUTPUT COMPONENT)
            (role concrete)
        )
;  Функция, вычисляющая значение элемента AND в зависимости от полученного аргумента
(deffunction and! (?x ?y)
                 (if (and (! = ?x 0) (!= ?y 0)) then 1 else 0))
;    Обработчик, выполняющий вычисления элемента AND при изменении входных сигналов
(defmessage-handler AND-GATE compute-output ()
                             (if (and (integerp ?self:input-1)
                                         (integerp ?self:input-2)) then
                                           (send ?self put-output-1
                                                  (and# ?self:input-1 ?self:input-2)))
                     )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    Класс, реализующий логику работы элемента OR, имеет два входа и один выход
(defclass OR-GATE
            (is-a TWO- INPUT ONE-OUTPUT COMPONENT)
            (role concrete)
      )
;    Функция, вычисляющая значение элемента OR в зависимости от полученного аргумента
(deffunction or# (?x ?y)
                 (if (or ( ! =  ?x 0) (I- ?y 0 ) )  then 1 else 0 ) )
;    Обработчик, выполняющий вычисления элемента OR при изменении входных сигналов
(defmessage-handler OR-GATE compute-output ()
                             (if (and (integerp ?self : input-1)
                                       (integerp ?self : input-2) ) then
                                         (send ?self put-output-1
                                                 (or# ?self : input-1 ?self: input-2) ))
                  )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    Класс, реализующий логику работы элемента NAND, имеет два входа и один выход
 (defclass NAND-GATE
            (is-a TWO-INPUT ONE-OUTPUT COMPONENT)
            (role concrete)
        )
;    Функция, вычисляющая значение элемента NAND в зависимости от полученного аргумента
 (deffunction nand# (?x ?y)
                 (if (not (and ( ! =  ?x 0) ( ! =  ?y 0») then 1 else 0 ) )
;    Обработчик, выполняющий вычисления элемента NAND при изменении входных сигналов
(defmessage-handler NAND-GATE compute-output ()
                           (if (and (integerp ?self: input-1)
                                     (integerp ?self: input-2) ) then
                                     (send ?self put-output-1
                                            (nand# ?self: input-1 ?self: input-2) ))
                   )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    Класс,реализующий логику работы элемента XOR, имеет два входа и один выход
(defclass XOR-GATE
             (is-a TWO- INPUT ONE-OUTPUT COMPONENT)
             (role concrete)
        )
;    Функция, вычисляющая значение элемента XOR в зависимости от полученного аргумента
(deffunction xor# (?x ?y)
```

```
                (if (or (and (= ?x 1) (= ?y 0 ) )
                        (and (= ?x 0} (= ?y 1 ) ) )  then 1 else 0 ) )
;    Обработчик, выполняющий вычисления элемента XOR при изменении входных сигналов
(defmessage-handler XOR-GATE compute-output (}
                          (if (and (integerp ?self : input-1)
                                   (integerp ?self : input-2) ) then
                                      (send ?self put-output-1
                                            (xor# ?self: input-1 ?self: input-2) ))
               )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    Класс, реализующий логику работы элемента SPLITTER, имеет один вход и два выхода
(defclass SPLITTER
          (is-a ONE-INPUT TWO-OUTPUT COMPONENT)
          (role concrete)
       )
;    Обработчик, выполняющий вычисления элемента SPLITTER при изменении входных сигналов
(defmessage-handler SPLITTER compute-output ()
                          (if (integerp ?self: input-1) then
                               (send ?self put-output-1 ?self: input-1)
                               (send ?self put-output-2 ?self: input-1) )
                 )
;= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
;    Методы родовой функции
;= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    Предварительное объявление родовой функции
(defgeneric connect)
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    Соединение элемента, имеющего один выход, с элементом, имеющим один вход
(defmethod connect ((?out ONE-OUTPUT) (?in ONE-INPUT))
            (send ?out put-output-1-link ?in)
            (send ?out put-output-1-link-pin 1)
            (send ?in put-input-1-link ?out)
            (send ?in put-input-1-link-pin 1)
       )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    Соединение элемента, имеющего один выход, с элементом, имеющим два входа
(defmethod connect ((?out ONE-OUTPUT) (?in TWO- INPUT) (?in-pin INTEGER))
            (send ?out put-output-1-link ?in)
            (send ?out put-output-1-link-pin ?in-pin)
            (send ?in  (sym-cat put-input- ?in-pin -link) ?out)
            (send ?in  (sym-cat put-input- ?in-pin -link-pin) 1)
          )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    Соединение элемента, имеющего два выхода, с элементом, имеющим один вход
(defmethod connect ((?out TWO-OUTPUT) (?out-pin INTEGER) (?in ONE-INPUT)
            (send ?out (sym-cat put-output- ?out-pin -link) ?in)
            (send ?out (sym-cat put-output- ?out-pin -link-pin) 1)
            (send ?in put-input-1-link ?out)
            (send ?in put-input-1-link-pin ?out-pin)
          )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    Соединение элемента, имеющего два выхода, с элементом, имеющим два входа
(defmethod connect ((?out TWO-OUTPUT) (?out-pin INTEGER)(?in TWO- INPUT) (?in-pin INTEGER))
            (send ?out (sym-cat put-output- ?out-pin -link) ?in)
```

```
            (send ?out (sym-cat put-output- ?out-pin -link-pin) ?in-pin)
            (send ?in  (sym-cat put-input- ?in-pin -link) ?out)
            (send ?in  (sym-cat put-input- ?in-pin -link-pin) ?out-pin)
        )
;=========================================================
;   Глобальные переменные
;=========================================================
(defglobal  ?*gray-code*       = (create$) ;   Переменная для хранения текущего кода Грея
            ?*sources*         = (create$) ;   Список источников текущей логической схемы
            ?*max-iterations* =  0)  ; Максимальное число итераций для текущей логической схемы
;=========================================================
;   Вспомогательные функции
;=========================================================
;---------------------------------------------------------
;   Определяет номер сигнала, который необходимо изменить для получения
;   следующего кода Грея
(deffunction change-which-bit (?x)
            (bind ?i 1)
              (while (and (evenp ?x) (!= ?x 0)) do
                    (bind ?x (div ?x 2) )
                    (bind ?i (+ ?i 1) )
               )
          ?i
       )
;---------------------------------------------------------
;   С помощью функции do-for-all-instances определяет обработанный сигнал с индикаторов
;   логической схемы
(def function LED- response ()
              (bind ? response (create$) )
              (do-for-all-instances ( (?led LED) ) TRUE
                   (bind ?response (create$ ?response
                                            (send ?led get-input-1) ) ) )
             ?response
             )
;---------------------------------------------------------
;   Предварительное объявление функции, необходимой для объединения элементов
;   логической схемы
deffunction connect-circuit ())
;=========================================================
;   Правила
;=========================================================
;---------------------------------------------------------
;   Инициализация логической схемы и запуск системы
(defrule startup
           =>
         ; инициализация текущей логической схемы
          (connect-circuit)
          ; получение имен всех источников текущей логической схемы
          (bind ?*sources* (find-all-instances ((?x SOURCE)) TRUE))
          ; создает нулевой код Грея
          (do-for-all-instances ((?x SOURCE)) TRUE
                    (bind ?*gray-code* (create$ ?*gray-code* 0)))
         ; определение максимального числа итераций
          (bind ?*max-iterations* (round (** 2 (length ?*sources*))
          ; обнуление количества сделанных итераций
```

```
            (assert (current-iteration 0))
            )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    Запуск процесса перебора всевозможных входных сигналов текущей логической системы
(defrule compute-response-1st-time
              ; если это первая итерация, то
              ?f <- (current-iteration 0)
              =>
              ; помещение во все источники нулевого сигнала
              (do-for-all-instances ((?source SOURCE)) TRUE
                                   (send ?source put-output-1 0))
              ; получение результата работы логической схемы
              (assert (result ?*gray-code* =(str-implode (LED-response)) ) )
              ; увеличение количества итераций на 1
              (retract ?f)
              (assert (current-iteration 1))
            )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    Перебор всевозможных входных сигналов текущей логической системы
(defrule compute-response-other-times
          ; если это не первая итерация и количество итераций еще не превышено
           ?f <- (current-iteration ?n&~0&:(< ?n ?*max-iterations*))
           =>
           ; вычисление номера источника, сигнал которого нужно менять
            (bind ?pos (change-which-bit ?n))
            ; получение следующего кода Грея
            (bind ?nv (- 1 (nth ?pos ?*gray-code*)))
            (bind ?*gray-code* (replace$ ?*gray-code* ?pos ?pos ?nv))
            ; изменение сигнала на заданном источнике на противоположный
            (send (nth ?pos ?*sources*) put-output-1 ?nv)
            ; получение результата работы логической схемы
            (assert (result ?*gray-code* =(str-implode (LED-response))))
            ; увеличение количества итераций на 1
            (retract ?f)
            (assert (current-iteration = ( + ?n 1) ) )
         )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    Оптимизация таблицы истинности
(defrule merge-responses
          ; более высокий приоритет позволяет производить оптимизацию
          ; в процессе построения таблицы истинности
            (declare (salience 10))
          ; если в текущей таблице есть две строки, которые можно объединить
            ?f1 <- (result $?b ?x $?e ?response)
            ?f2 <- (result $?b ~?x $?e ?response)
           =>
          ; то удалить такие строки
            (retract ?f1 ?f2)
            ; и вставить обобщенную строку
            (assert (result ?b * ?e ?response) )
            )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    Вывод заголовка таблицы истинности
(defrule print-header
          ; более низкий приоритет запрещает применение этого правила
```

```
        ; до окончания перебора всевозможных вариантов входных сигналов
         (declare (salience -10) )
         =>
        ; вывод списка источников
        (do-for-all-instances ( ( ? x  SOURCE)) TRUE
                                  (format t " %3s " (sym-cat ?x) ) )
         ; вывод разделительной линии
        (printout t " | ")
        ; вывод списка индикаторов
        (do-for-all-instances ( (?x LED)) TRUE
                                  (format t " %3s " (sym-cat ?x) ) )
         (format t "%n")
        ; вывод разделительной линии, отделяющей заголовок
        (do-for-all-instances ( ( ? x  SOURCE)) TRUE
                                  (printout t " ----- ") ) (printout t "-+-")
         (do-for-all-instances ( ( ? x  LED)) TRUE
                                  (printout t " ----- ") )
         (format t "%n")
        ; запрос на печать таблицы истинности
        (assert (print-results) )
      )
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;    Вывод таблицы истинности
(defrule print-result
        ; если заголовок уже напечатан
          (print-results)
         ; еще остались не выведенные строки
           ?f <- (result $?input ?response)
         ; выбор наименьшей по порядку строки
        (not (result $?input-2 ?response-2&:
                (< (str-compare ?response-2 ?response) 0) ))
         =>
        ; удаление выбранной строки
          (retract ?f)
        ; вывод выбранной строки
        (while (neq ?input (create$) ) do
                  (printout t "  " (nth 1 ? input) "
                  (bind ?input (rest$ ? input) ))
         (printout t " | ")
         (bind ?response (str-explode ?response) )
         (while (neq ?response (create$) ) do
                  (printout t "  " (nth 1 ?response)
                  (bind ?response (rest$ ?response) ) )
         (printout t crlf)
        )
```

Создайте файл cios.CLP, содержащий текст переведенной выше программы. Как уже не раз упоминалось, среда CLIPS воспринимает только символы английского алфавита, поэтому комментарии, приведенные в листинге, необходимо опустить.

14.8. Тестирование системы

Для проверки экспертной системы будем использовать логические схемы, приведенные на рис. 14.2—14.4. Создайте три файла: scheme- 1.CLP, scheme-2.CLP и scheme-3.CLP. Содержание этих файлов приведено ниже.

Пример 14.19. Содержимое файла scheme-1.CLP

```
(definstances circuit
              (S-l of SOURCE)
              (S-2 of SOURCE)
              (P-l of SPLITTER)
              (P-2 of SPLITTER)
              (N-l of NOT-GATE)
              (O-1 of OR-GATE)
              (X-l of XOR-GATE)
              (L-l of LED)
              (L-2 of LED)
              )
(deffunction connect-circuit ()
              (connect [S-l] [P-l])
              (connect [S-2] [X-l] 2)
              (connect [P-l] 1 [N-l])
              (connect [P-l]  2  [O-l]  2)
              (connect [N-1]  [O-l]  1)
              (connect [O-l]  [P-2])
              (connect [P-2]  1  [L-l])
              (connect [P-2]  2  [X-l]  1)
              (connect [X-l] [L-2])
```

Пример 14.20. Содержимое файла scheme-2.CLP

```
(definstances circuit
              (S-l of SOURCE)
              (S-2 of SOURCE)
              (P-l of SPLITTER)
              (P-2 of SPLITTER)
              (A-l of AND-GATE)
              (N-1 of NOT-GATE)
              (N-2 of NOT-GATE)
              (N-3 of NOT-GATE)
              (O-1 of OR-GATE)
              (X-l of XOR-GATE)
              (L-l of LED)
              )
(deffunction connect-circuit ()
              (connect [S-l] [P-l])
              (connect [S-2] [P-2])
              (connect [P-l] 1 [A-l]  1)
              (connect [P-l]  2  [N-2])
              (connect [P-2] 1 [A-l]  2)
              (connect [P-2] 2 [N-3])
              (connect [A-1]  [N-l])
              (connect [N-2]  [O-1]  1)
              (connect [N-3]  [O-l]  2)
              (connect [N-1]  [X-l]  1)
              (connect [O-l] [X-1]  2)
              (connect [X-1]  [L-l])
        )
```

Пример 14.21. Содержимое файла scheme-3.CLP

```
(definstances circuit
                (S-l of SOURCE)
                (S-2 of SOURCE)
                (S-3 of SOURCE)
                (S-4 of SOURCE)
                (S-5 of SOURCE)
                (S-6 of SOURCE)
                (P-l of SPLITTER)
                (P-2 of SPLITTER)
                (P-3 of SPLITTER)
                (P-4 of SPLITTER)
                (N-1 of NOT-GATE)
                (N-2 of NOT-GATE)
                (N-3 of NOT-GATE)
                (O-1 of OR-GATE)
                (O-2 of OR-GATE)
                (X-l of XOR-GATE)
                (X-2 of XOR-GATE)
                (A-l of AND-GATE)
                (D-l of NAND-GATE)
                (D-2 of NAND-GATE)
                (L-l of LED)
                (L-2 of LED)
                (L-3 of LED)
                )
(deffunction connect-circuit ()
                (connect [S-l] [P-l])
                (connect [S-2] [P-2])
                (connect [S-3] [P-3])
                (connect [S-4] [A-1]  2)
                (connect [S-5] [D-1]  2)
                (connect [S-6] [O-2]  2)
                (connect [P-l] 1 [O-l]  1)
                (connect [P-l]  2  [N-1])
                (connect [P-2]  1  [N-2])
                (connect [P-2] 2 [A-l]  1)
                (connect [P-3] 1 [X-1]  2)
                (connect [P-3] 2 [D-1]  1)
                (connect [N-1]  [X-1]  1)
                (connect [N-2]  [O-l]  2)
                (connect [O-l] [X-2]  1)
                (connect [X-1]  [X-2]  2)
                (connect [A-1]  [D-2]  1)
                (connect [D-l] [O-2]  1)
                (connect [X-2]  [P-4])
                (connect [O-2]  [D-2]  2)
                (connect [P-4]  1  [N-3])
                (connect [P-4]  2  [L-2])
                (connect [D-2]  [L-3])
                (connect [N-3]  [L-1])
        )
```

14.9. Запуск программы

Для запуска программы очистите среду CLIPS командой (clear) и выполните команду (load "cios.CLP"). Если при создании файла не было допущено ошибок, вы должны увидеть сообщения, представленные на рис. 14.5.

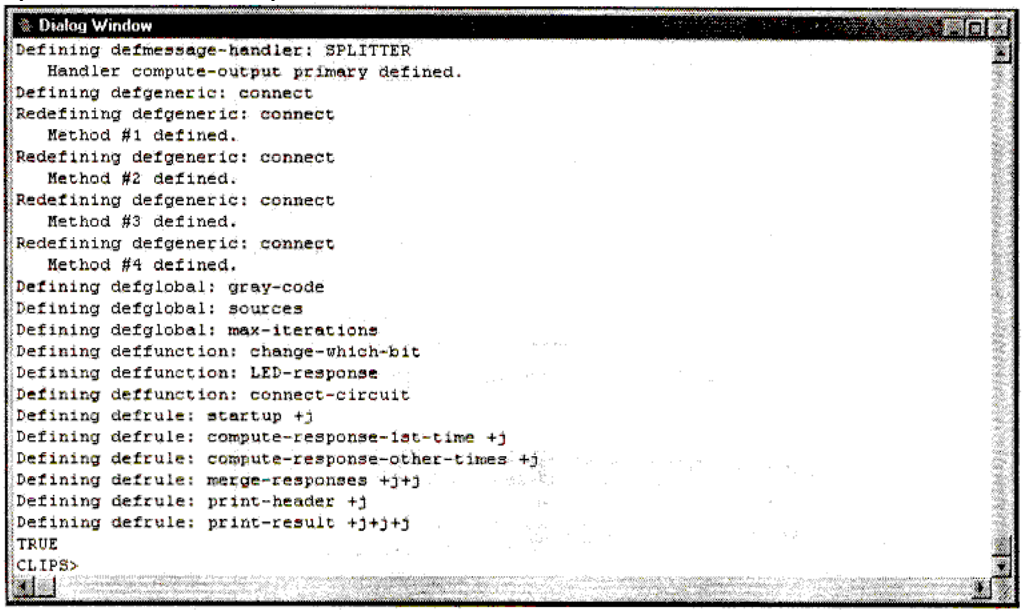

Рис. 14.5. Загрузка экспертной системы

Рис. 14.5 демонстрирует успешный вариант загрузки экспертной системы. Обратите внимание, что функция load вернула значение TRUE. Если это не так, значит, в синтаксисе определений файла cios.CLP была допущена ошибка.

После удачной загрузки следует убедиться, что необходимые определения присутствуют в системе. Легче всего это выполнить с помощью соответствующих менеджеров. Внешний вид этих менеджеров приведен на рис. 14.6— 14.9.

Для запуска экспертной системы необходимо загрузить файл с определением какой-нибудь логической схемы, например, командой (load "scheme-1").

После этого необходимо выполнить команды reset и run для запуска основного цикла выполнения правил. Пример работы системы показан на рис. 14.10.

Для повторного запуска экспертной системы нужно еще раз выполнить команды load, reset и run. На рис. 14.11 и 14.12 приведены результаты работы системы для схем 2 и 3 соответственно.

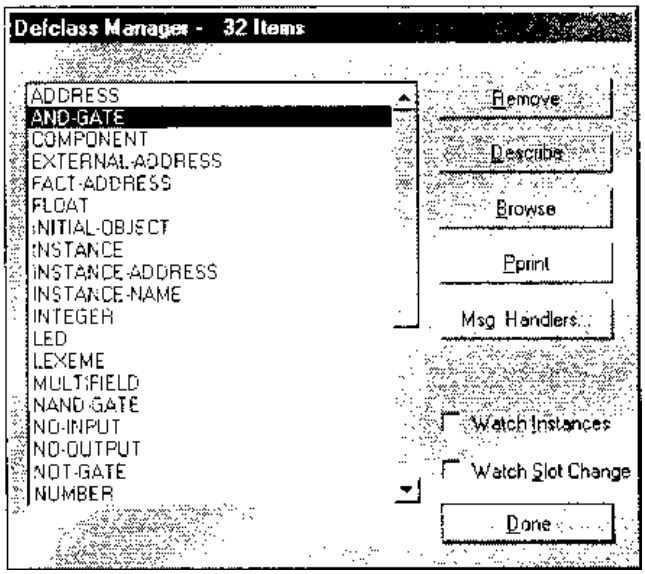

Рис. 14.6. Классы экспертной системы

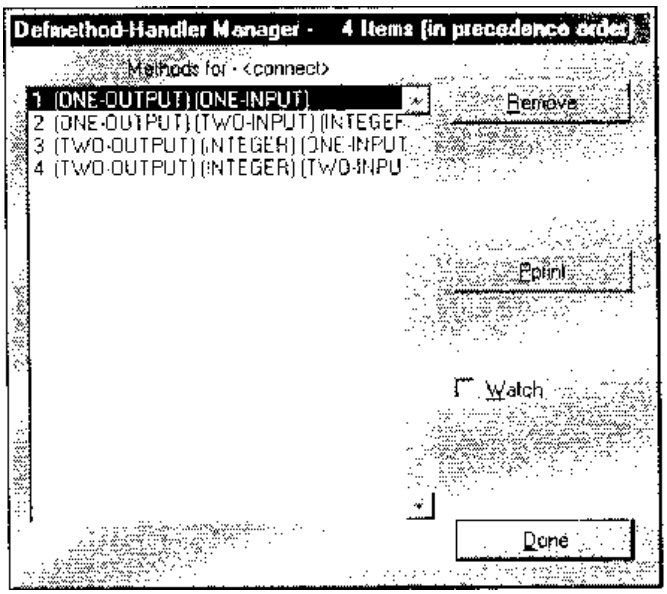

Рис. 14.7. Методы экспертной системы

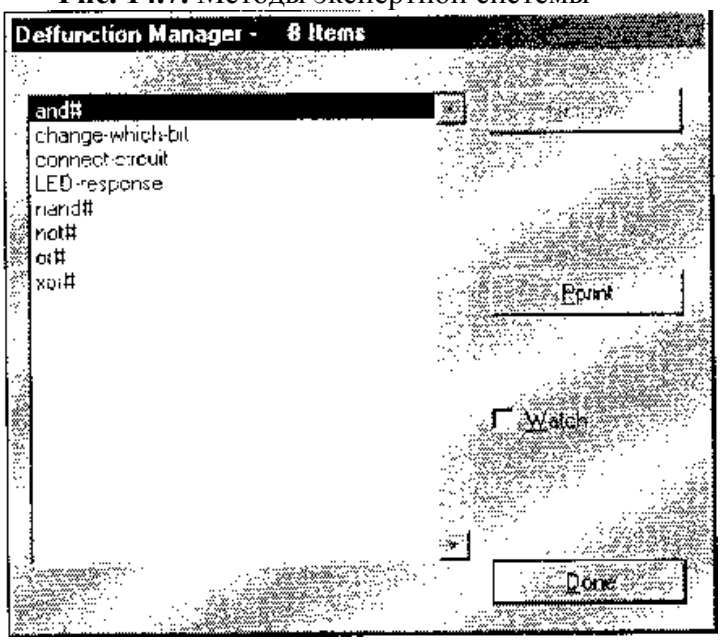

Рис. 14.8. Функции экспертной системы

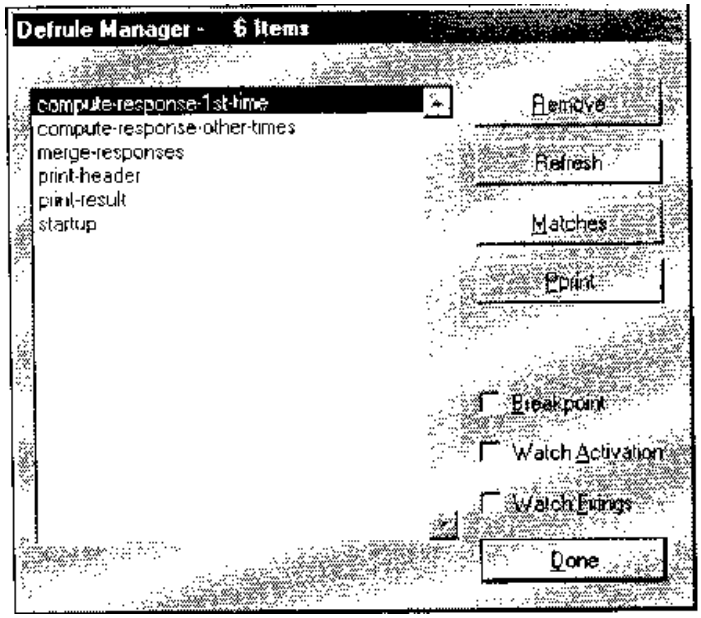

Рис. 14.9. Правила экспертной системы

Для того чтобы лучше понять принципы работы экспертной системы, попробуйте выполнить ее по шагам с помощью команды (run 1), наблюдая за изменениями в списке фактов, объектов, переменных и плане решения задачи. Для этого сделайте видимыми окна **Facts Window, Instances Window, Globals Window и Agenda Window,** воспользовавшись соответствующими пунктами меню **Window.** Пример такого пошагового нахождения решения приведен на рис. 14.13.

Пример построения экспертной системы, приведенный в данной главе, является наглядным подтверждением того, насколько эффективным может оказаться использование объектно-ориентированных возможностей и родовых функций в среде CLIPS.

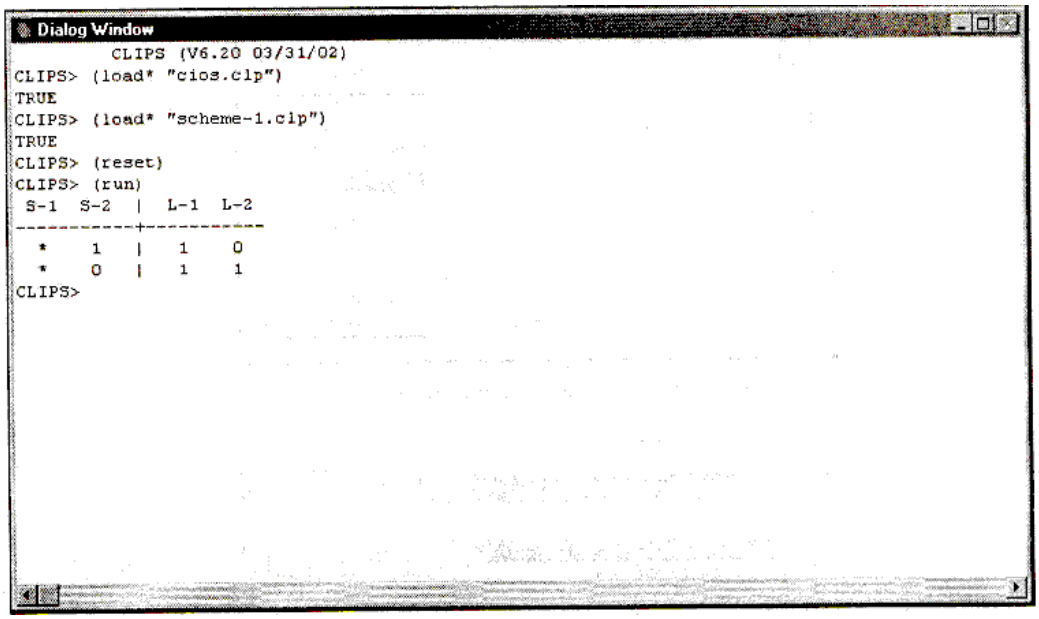

Рис. 14.10. Оптимизированная таблица истинности для логической схемы 1

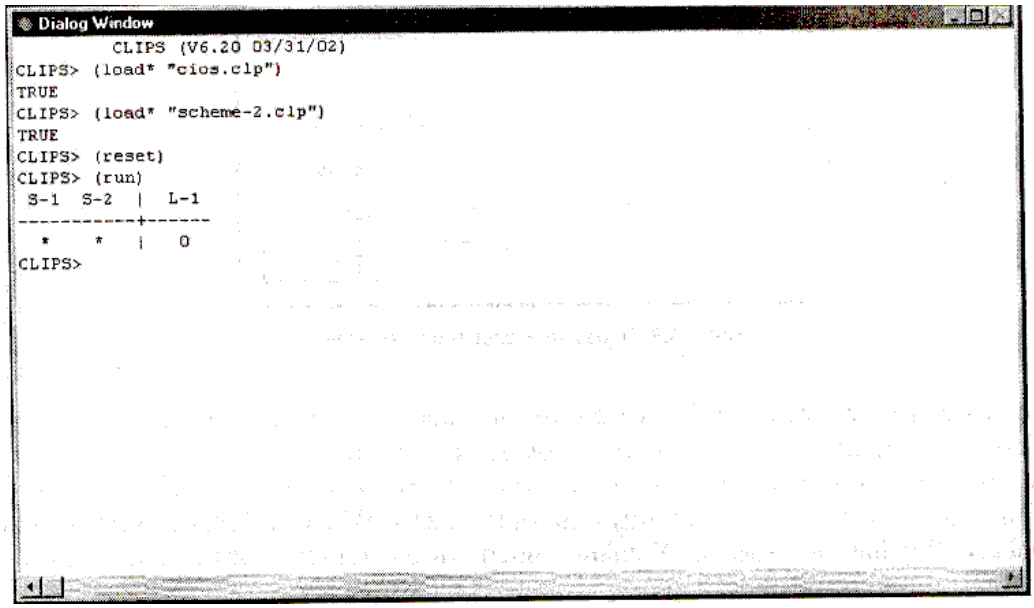

Рис. 14.11. Оптимизированная таблица истинности для логической схемы 2

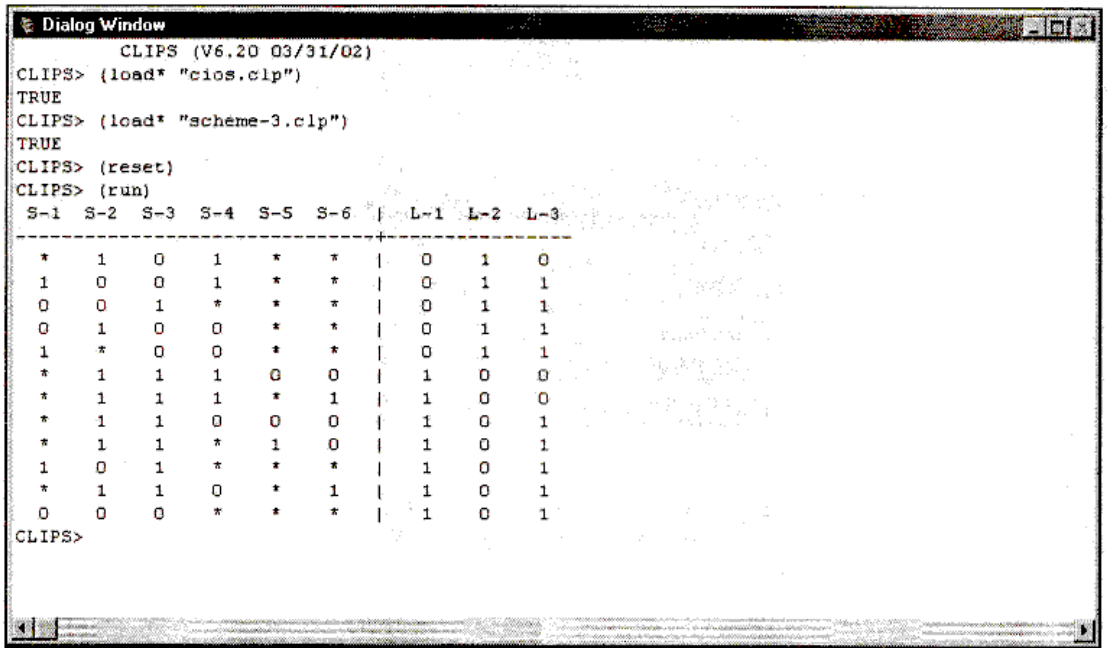

Рис. 14.12. Оптимизированная таблица истинности для логической схемы 3

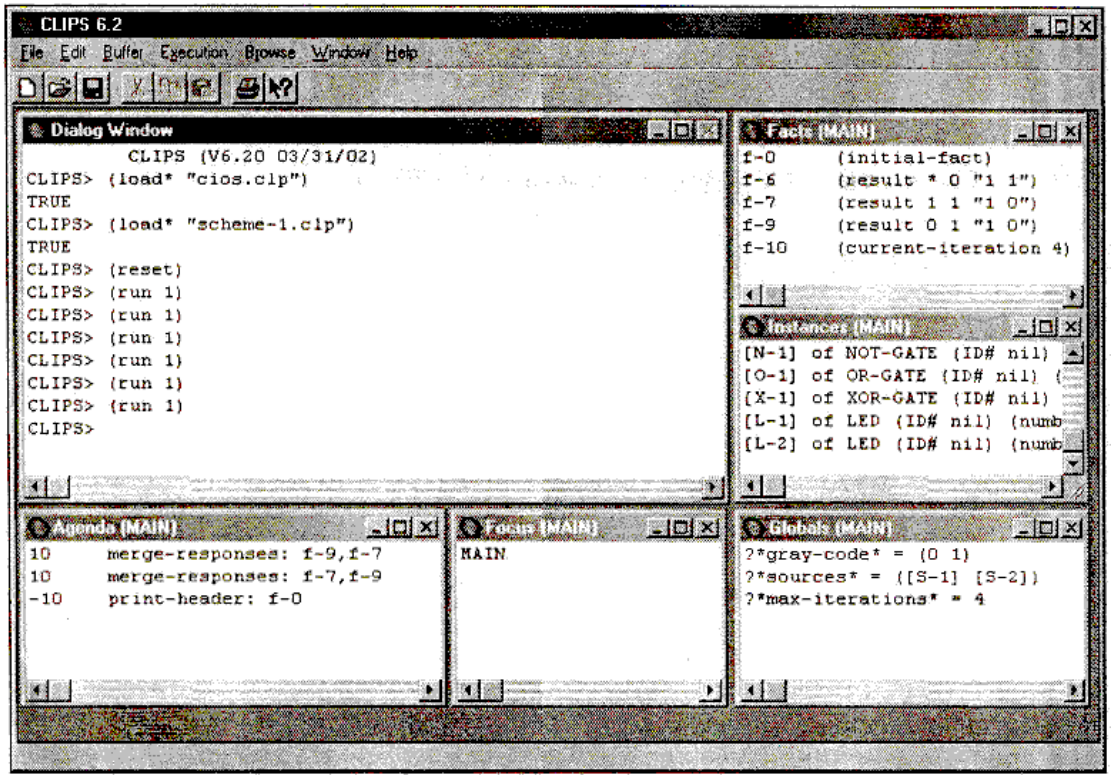

Рис. 14.13. Пошаговое выполнение правил

ЧАСТЬ V.Функции и команды CLIPS.

ГЛАВА 15Основные функции CLIPS

Функциями в CLIPS называются предопределенные системные действия, обрабатывающие заданный набор аргументов и возвращающие некоторый результат. Функции могут вводиться в диалоговом окне CLIPS с клавиатуры или использоваться в правилах, обработчиках сообщений, определенных пользователем в функциях или родовых функциях.
CLIPS предоставляет довольно большой набор функций, способный удовлетворить любые потребности пользователя, среди которых всевозможные логические и математические функции, функции работы со строками и составными величинами, функции ввода/вывода, процедурные функции, функции для работы с методами родовых функций, функции, ориентированные на поддержку объектно-ориентированных возможностей, а также функции для работы с конструкторами deftemplate, deffacts, defrule, defglobal, deffunction, defgeneric, defmethod, defmodule.
Данная глава посвящена описанию наиболее важных функций среды CLIPS. Желающие получить полную информацию обо всех доступных функциях CLIPS и методах их работы могут обратиться к книгам *"CLIPS Reference Manual, Volume I, Basic Programming Guide" и "CLIPS Reference Manual, Volume II, Advanced Programming Guide"*.

15.1. Логические функции

CLIPS предоставляет довольно богатый набор логических функций, описанию которых целиком посвящен этот раздел.
Среди всевозможных логических функций отдельной подгруппой выделяются предикатные функции, назначение которых заключается в тестировании своего единственного аргумента на принадлежность к тому или иному типу. Ниже приведен обобщенный синтаксис предикатных функций CLIPS.

Определение 15.1. Предикатные функции

(<имя-предикатной-функции> <выражение>)
При выполнении предикатной функции вычисляется выражение, переданное ей в качестве единственного параметра, и проверяется соответствие параметра некоторому типу. Если принадлежность к проверяемому типу подтверждается, функция возвращает значение TRUE, в противном случае — FALSE. Полный список предикатных функций приведен в табл. 15.1.

Таблица 15.1. *Предикатные функции*

Функция	Описание
numberp	Проверка, относится ли аргумент к типу float или integer
floatp	Проверка, относится ли аргумент к типу float
integerp	Проверка, относится ли аргумент к типу integer
lexemep	Проверка, относится ли аргумент к типу symbol или string
stringp	Проверка, относится ли аргумент к типу string
symbolp	Проверка, относится ли аргумент к типу symbol
wordp	Синоним функции symbolp
evenp	Проверка целого числа на четность
oddp	Проверка целого числа на нечетность
multifieldp	Проверка, является ли аргумент составным полем
sequencep	Синоним функции multifieldp
pointerp	Проверка, относится ли аргумент к типу external-address

Другими полезными логическими функциями являются eq и neq, синтаксис которых приведен ниже.

Определение 15.2. Функции eq и neq

(eq <выражение1> <выражение2>+)

(neq <выражение1> <выражение2>+)

Функция eq возвращает значение TRUE, если ее первый аргумент равен второму и всем последующим аргументам (если они присутствуют). В противном случае функция возвращает значение FALSE. Важной особенностью функции является то, что она сравнивает как значения аргументов, так и их типы. Например, результатом выполнения выражения (eq 3 3.0) будет значение FALSE, т. к. число 3 принадлежит типу integer, а число 3.0 — типу float.
Функция neq, напротив, возвращает значение TRUE, если ее первый аргумент не равен второму и последующим аргументам, и значение FALSE — в противном случае. В примере 15.1 показано использование функций eq и neq.

Пример 15.1. Использование функций eq и neq

```
(eq foo bar mumble foo)
(eq foo foo foo foo)
(eq 3 4)
(neq foo bar yak bar)
(neq foo foo yak bar)
(neq 3 a)
```

Результат приведенных выше вызовов функций eq и neq представлен на рис. 15.1.

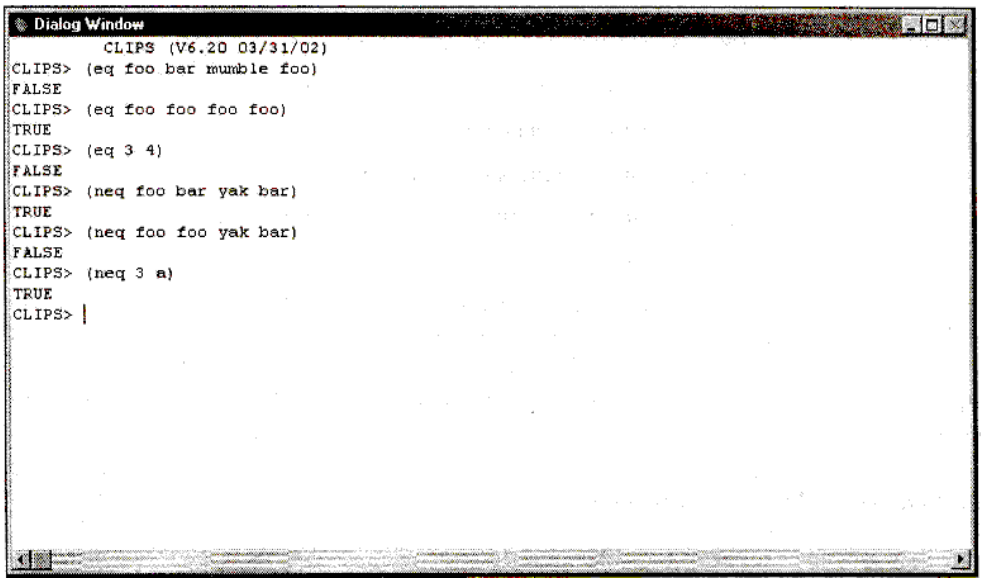

Рис. 15.1. Использование функций eq и neq

Помимо функций eq и neq, CLIPS предоставляет несколько функций, предназначенных для сравнения чисел. Общий синтаксис таких функций приведен ниже.

Определение 15.3. Функции сравнения чисел

(<имя-функции> <численное-выражение1> <численное-выражение2>+)

Функции этой группы вычисляют полученные в качестве параметров выражения и проверяют, выполняется ли заданное условие между первым и всеми последующими аргументами. Если соответствующее условие выполняется, функция возвращает значение TRUE, в противном случае — FALSE. Все функции данной группы сравнивают только числовые значения и при необходимости преобразуют значения типа integer в тип float. Список и описание функций, предназначенных для сравнения чисел, приведен в табл. 15.2.

Таблица 15.2. *Функции сравнения чисел*

Функция	Описание
=	Проверка равенства первого и остальных аргументов
<>	Проверка неравенства первого и остальных аргументов
>	Проверка того, что первый аргумент больше остальных
>=	Проверка того, что первый аргумент больше или равен остальным
<	Проверка того, что первый аргумент меньше остальных
<=	Проверка того, что первый аргумент меньше или равен остальным

Описанные выше функции сравнения чисел обладают одной важной особенностью. Благодаря тому, что точность чисел с плавающей точкой изменяется при переходе от одного компьютера к другому, функции сравнения могут получать различные результаты на различных компьютерах. Даже если код программы не переносится на другой ПК, погрешность округления может вызвать различные ошибки. В примере 15.2 функция = ошибочно возвратит значение TRUE, потому что оба числа округлены до 0.6666666666666666667 (рис. 15.2).

Пример 15.2. Ошибки округления

```
(= 0.66666666666666666 0.66666666666666666667)
```

Кроме описанных выше функций CLIPS предоставляет также три функции стандартной булевой логики: and, or и not.

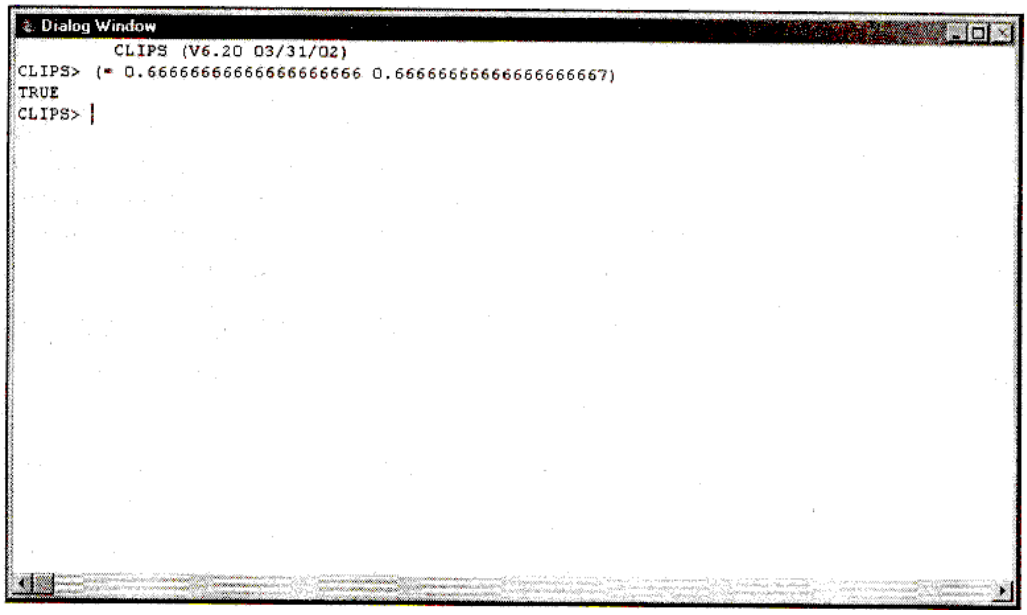

Рис. 15.2. Ошибки округления

Определение 15.4. Функции and, or и not

(and <выражение>+)
(or <выражение>+)
(not <выражение>+)

Функция and возвращает значение TRUE, если значение каждого из ее аргументов равняется TRUE. В противном случае она возвращает значение FALSE. Каждый аргумент функции проверяется слева направо. Если встречается аргумент со значением FALSE, проверка значений аргументов прекращается, и функция возвращает значение FALSE.

Функция or возвращает значение TRUE, если значение хотя бы одного из ее аргументов равняется TRUE. Иначе она возвращает значение FALSE. Как и в случае функции and,

аргументы функции проверяются слева направо. При встрече первого аргумента со значением TRUE проверка значений аргументов прекращается, и все выражение получает значение TRUE.

Функция not возвращает значение TRUE, если ее аргумент имеет значение, равное FALSE. В противном случае она возвращает значение FALSE.

15.2. Математические функции

Богатый набор математических функций, предоставляемый CLIPS, позволяет использовать эту среду не только для решения логических задач, но и для серьезных математических вычислений. Математические функции CLIPS разделены на два пакета: набор стандартных и набор расширенных математических функций. Стандартные математические функции перечислены в табл. 15.3.

Таблица 15.3. *Стандартные математические функции*

Функция	Описание
+	Сложение
-	Вычитание
*	Умножение
/	Деление
div	Целочисленное деление
max	Максимальное числовое значение
min	Минимальное числовое значение
abs	Абсолютное значение
float	Преобразование в тип float
integer	Преобразование в тип integer

Перечисленные функции могут быть использованы только с числовыми аргументами. Если в математическую функцию будет передан аргумент строкового или другого неподходящего типа, функция вернет сообщение об ошибке. Ниже приведен синтаксис и описаны некоторые особенности стандартных математических функций.

Функция + возвращает сумму своих аргументов. Если все аргументы функции принадлежат типу float, возвращаемое функцией значение также будет вещественного типа. Аналогично возвращаемый функцией результат принадлежит типу integer, если все аргументы целые. Сложение невыполнимо, если типы переданных аргументов различаются (integer и float). В этом случае функция возвращает промежуточное значение, полученное при сложении аргументов одинакового типа.

Определение 15.5. Функция +

(+ <выражение!> <выражение2>+)

Функция - возвращает значение ее первого аргумента минус все последующие.

Определение 15.6. Функция –

(- <выражение!> <выражение2>+)

Функция * возвращает произведение своих аргументов. Все замечания по поводу типов аргументов, сделанные для функции +, в полной мере применимы к аргументам функций - и *.

Определение 15.7. Функция *

(* <выражение1> <выражение2>+)

Функция / возвращает частное от деления первого аргумента на каждый из последующих. Как и в случаях функций +, - и *, функция / не допускает смешанного набора аргументов. Однако по умолчанию делимое автоматически преобразуется в число с плавающей точкой. Таким образом, все последующие аргументы и результат выполнения функции должны представлять собой числа с плавающей точкой. Функция set-auto-float-dividend позволяет менять подобное поведение системы. Если, например, режим автоматического преобразования чисел при делении отключен, выражение (/ 4 3 4.0) возвратит значение 0.25, в то время как при конфигурации по умолчанию результатом подобного действия является 0.333333333.

Определение 15.8. Функция /

(/ <выражение1> <выражение2>+)

Функция div возвращает частное от деления первого аргумента на каждый из последующих. Все аргументы данной функции автоматически преобразуются в целые для выполнения целочисленного деления. Эта функция возвращает значение типа integer.

Определение 15.9. Функция div

(div <выражение1> <выражение2>+)

Функции max и min предназначены для нахождения наибольшего и наименьшего аргумента соответственно. При необходимости, аргументы типа integer временно преобразуются в тип float для выполнения сравнения. Значение, возвращаемое этими функциями, может быть как целого, так и вещественного типа (в зависимости от типа наибольшего или наименьшего аргумента).

Определение 15.10. Функции max и min

(max <выражение>+) (min <выражение>+)

Функция abs возвращает абсолютное значение аргумента. Значение, возвращаемое этой функцией, может быть как целого, так и вещественного типа (в зависимости от типа аргумента).

Определение 15.11. Функция abs

(abs <выражение>)

Функции float и integer предназначены для преобразования аргумента в тип вещественный и целый тип соответственно.

Определение 15.12. Функции float и integer

(float <выражение>+)
(integer <выражение>+)

В дополнение к стандартным математическим функциям, CLIPS также содержит большое число научных и тригонометрических функций для более сложных вычислений. Пакет этих функций, включенный в основную версию CLIPS, может быть отключен с целью экономии памяти, если экспертная система не нуждается в этих возможностях.
Довольно большую подгруппу расширенных математических функций составляют различные тригонометрические функции. Такие функции принимают один числовой аргумент и возвращают число с плавающей точкой. Для проведения вычислений аргументы тригонометрических функций необходимо переводить в радианы. Полный список тригонометрических функций, предоставляемых CLIPS, приведен в табл. 15.4.

Таблица 15.4. *Тригонометрические функции*

Функция	Описание
acos	Арккосинус
acosh	Гиперболический арккосинус
acot	Арккотангенс
acoth	Гиперболический арккотангенс
acsc	Арккосеканс
acsch	Гиперболический арккосеканс
asec	Арксеканс
asech	Гиперболический арксеканс
asin	Арксинус
asinh	Гиперболический арксинус
atan	Арктангенс
atanh	Гиперболический арктангенс
cos	Косинус
cosh	Гиперболический косинус
cot	Котангенс
coth	Гиперболический котангенс
csc	Косеканс
csch	Гиперболический косеканс
sec	Секанс
sech	Гиперболический секанс
sin	Синус
sinh	Гиперболический синус
tan	Тангенс
tanh	Гиперболический тангенс

Остальные расширенные математические функции, не вошедшие в подгруппу тригонометрических функций, представлены в табл. 15.5.

Таблица 15.5. *Расширенные математические функции*

Функция	Описание
deg-grad	Преобразование из градусов в секторы
deg-rad	Преобразование из градусов в радианы
grad-deg	Преобразование из секторов в градусы
rad-deg	Преобразование из радиан в градусы
pi	Получение значение числа π
sqrt	Вычисление квадратного корня
**	Вычисление степени числа
exp	Вычисление экспоненты
log	Вычисление логарифма
log10	Вычисление десятичного логарифма
round	Округление числа
mod	Вычисление остатка от деления

CLIPS предоставляет 4 функции, предназначенные для преобразования своего аргумента из градусов в секторы и радианы и обратно (360° равняется сектору, размером 400, и 2 радианам). Значения, возвращаемые этими функциями, принадлежат вещественному типу. Синтаксис функций приведен ниже.

Определение 15.13. Функции deg-grad, deg-rad, grad-deg и rad-deg

(deg-grad <выражение>)

(deg-rad <выражение>)
(grad-deg <выражение>)
(rad-deg <выражение>)

Функция pi предназначена для получения числа «пи» с точностью до 14-го знака и не имеет параметров.

Определение 15.14. Функция pi

(pi)

Функция sqrt возвращает значение квадратного корня, извлеченного из ее аргумента, в виде числа с плавающей точкой.

Определение 15.15. Функция sqrt

(sqrt <выражение>)

Для возведения числа в заданную степень предназначена функция **. Она возводит первый аргумент в степень (возможно не целую), заданную вторым аргументом, и возвращает результат вещественного типа.

Определение 15.16. Функция **

(** <выражение1> <выражение2>)

Функция exp возводит число e (основу натурального логарифма, имеющую значение, приблизительно равное 2.718281828459045) в степень, равную полученному аргументу, и возвращает полученное значение в виде числа с плавающей точкой.

Определение 15.17. Функция exp

(exp <выражение>)

CLIPS предоставляет пару функций, предназначенных для вычисления натурального и десятичного логарифма.

Определение 15.18. Функции log и logl0

(log <выражение>)
(logl0 <выражение>)

Аргументы и возвращаемый результат обеих функций являются вещественными значениями. Функция log (натуральный логарифм) возвращает такое число x, что следующее уравнение является верным *n* = e*, где *n* — аргумент функции. В свою очередь функция log10 (десятичный логарифм) возвращает число x, удовлетворяющее уравнению $n = 10^x$, где *n* — аргумент функции. В примере 15.3 и на рис. 15.3 приведены варианты использования логарифмических функций.

Пример 15.3. Использование логарифмических функций

(log 2.71828182845904)

(log (exp 1))

(logl0 100)

Функция round округляет свой аргумент до ближайшего целого числа. Если аргумент находится точно между двумя целыми числами, то он округляется к меньшему числу. Тип возвращаемого результата — integer.

Определение 15.19. Функция round

(round <выражение>)

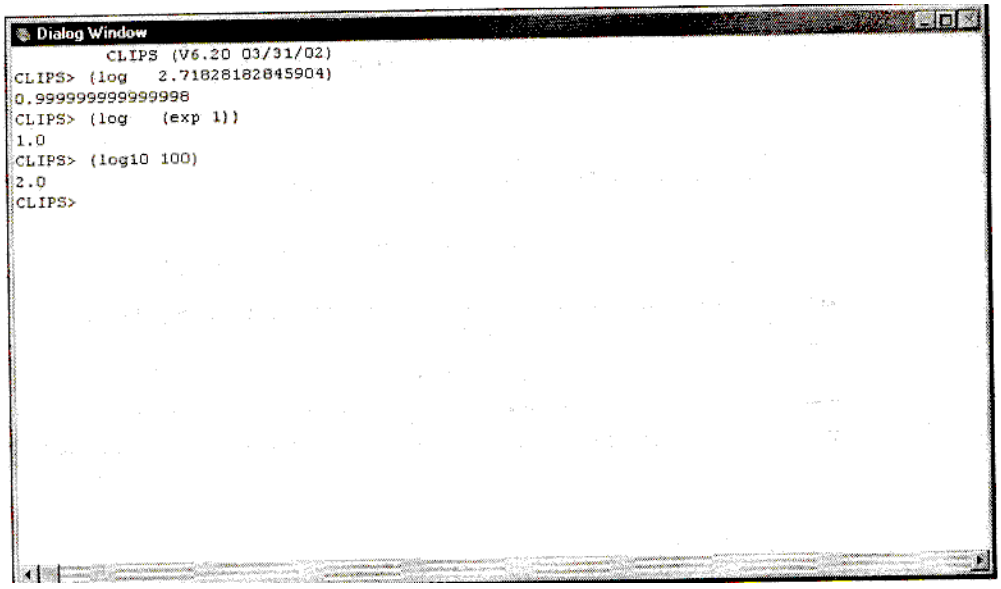

Рис. 15.3. Использование логарифмических функций

Функция mod возвращает остаток от деления первого аргумента на второй, предполагая, что результат деления должен быть целочисленным. Функция возвращает значение типа integer, если оба аргумента целочисленные. В противном случае функция возвращает вещественный результат.

Определение 15.20. Функция mod

(mod <выражение!> <выражение2>)

15.3. Функции работы со строками

CLIPS предоставляет 12 функций работы со строками. Данный набор обеспечивает программиста всеми необходимыми операциями для работы с тестом. Список этих функций и их краткое описание приведены в табл. 15.6.

Таблица 15.6. *Функции работы со строками*

Функция	**Описание**
str-cat	Объединение строк
sym-cat	Объединение строк в значение типа symbol
sub-string	Выделение подстроки
str-index	Поиск подстроки
eval	Выполнение строки в качестве команды CLIPS
build	Выполнение строки в качестве конструктора CLIPS
upcase	Преобразование символов строки в символы верхнего регистра
lowcase	Преобразование символов строки в символы нижнего регистра
str-compare	Сравнение двух строк
str-length	Определение длины строки
check-syntax	Проверка синтаксиса строкового выражения
string-to-field	Преобразование строки в поле одного из примитивных типов данных CLIPS

Функция str-cat объединяет все свои аргументы в строку и возвращает ее в качестве результата. Аргументы этой функции должны принадлежать одному из следующих типов: symbol, string, float, integer или instance-name.

Определение 15.21. Функция str-cat

(str-cat <выражение>*)

Функция sym-cat объединяет свои аргументы и возвращает в качестве результата значение типа symbol. Тип возвращаемого результата — единственное отличие sym-cat от функции str-cat.

Определение 15.22. Функция sym-cat

(sym-cat <выражение>*)

Варианты использования функций str-cat и sym-cat приведены в примере 15.4 и на рис. 15.4.

Пример 15.4. Использование функций str-cat и sym-cat

```
(str-cat   "foo"  bar)
(sym-cat  "foo" bar)
```

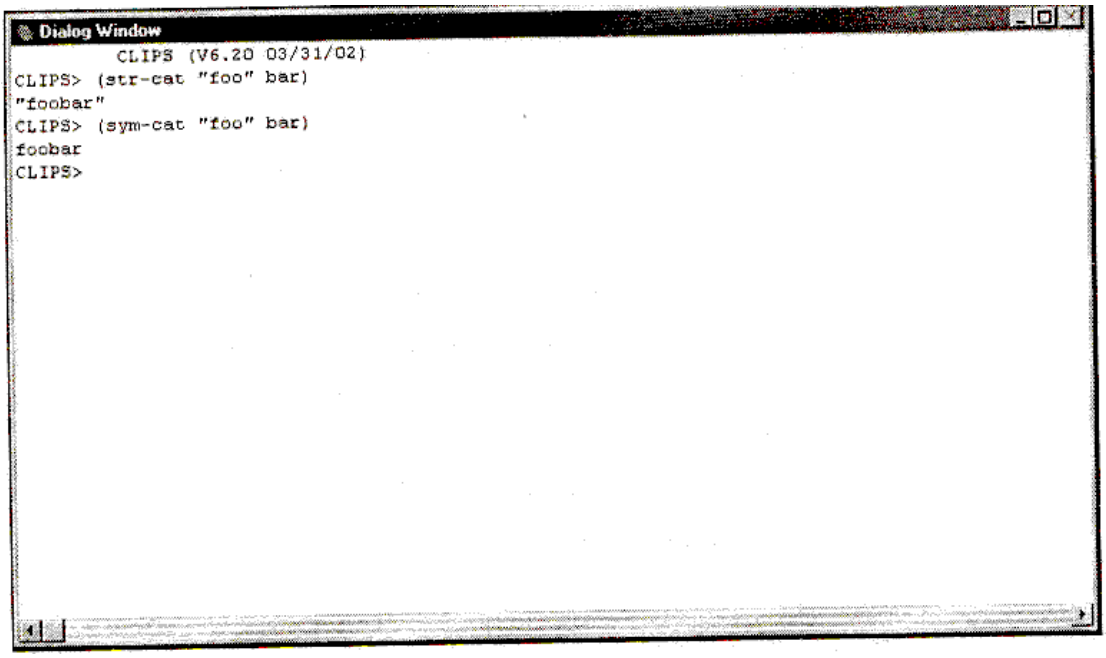

Рис. 15.4. Использование функций str-cat и sym-cat

Для выделения подстроки служит функция sub-string, которая возвращает фрагмент исходной строки как отдельную строку.

Определение 15.23. Функция sub-string

(sub-string <целочисленное-выражение1> <целочисленное-выражение2> <строка>)

Первый аргумент функции задает индекс первого символа выделяемой подстроки, а второй аргумент — последнего символа. Сама строка определяется последним аргументом. Если первый аргумент больше второго, функция возвращает 0.

Функция str-index возвращает позицию заданной подстроки внутри строки. Результат выполнения функции относится к целому типу и равен индексу первого символа подстроки. В случае если искомая подстрока не была найдена, функция str-index возвращает значение FALSE.

Определение 15.24. Функция str-index

(str-index <подстрока> <строка>)

Варианты использования функций sub-string и str-index приведены в примере 15.5 и на рис. 15.5.

Рис. 15.5. Использование функций sub-string и str-index

Пример 15.5. Использование функций stub-string и str-index

```
(sub-string 3 8 "abcdefghijkl")
(str-index "def" "abcdefghi")
(str-index "qwerty" "qwertypoiuyt")
(str-index "qwerty" "poiuytqwer")
```

Функция eval выполняет указанное выражение в строке, как будто это команда, введенная извне в среду CLIPS. Единственный аргумент функции — команда, которая будет выполнена, заданная значением типа string или symbol. Функция eval не позволяет использовать локальные переменные кроме случаев, когда локальная переменная определяется внутри исполняемой строки. К тому же данная функция не допускает выполнение конструкторов CLIPS. Значение, возвращаемое функцией eval, является результатом выполненной команды или равно FALSE в случае ошибки.

Определение 15.25. Функция eval

(eval <строка>)

Для выполнения строкового выражения в качестве конструктора среды CLIPS служит функция build.

Определение 15.26. Функция build

(build <строка>)

Единственный аргумент функции build — значение типа string или symbol. Аргумент представляют собой конструктор, который будет выполнен в среде CLIPS. Функция build возвращает значение TRUE, если выполнение прошло успешно, и значение FALSE, если при выполнении произошли ошибки. Варианты использования функций eval и build приведены в примере 15.6 и на рис. 15.6.

Пример 15.6. Использование функций eval и build

```
(eval  "(+ 3  4)")
(eval "(create$ a b c)")
(build "(defrule foo (a) => (assert (b)|
(rules)
```

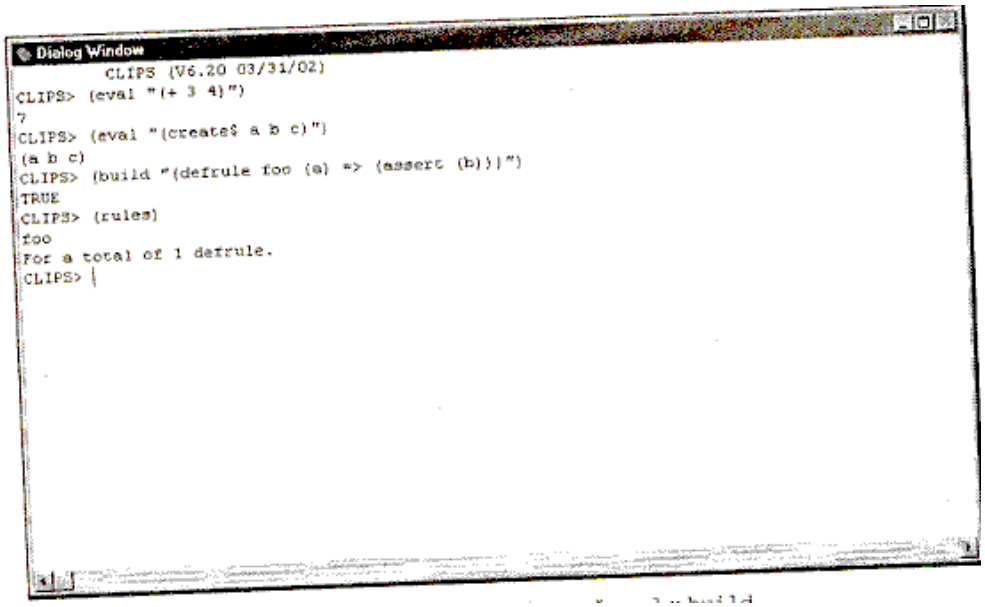

Рис. 15.6. Использование функций eval и build

Функции upcase и lowcase служат для преобразования всех символов заданной строки в верхний и нижний регистр соответственно. Аргумент этих функций должен принадлежать одному из типов: symbol или string. Результат, возвращаемый функциями, соответствует типу получаемого аргумента.

Определение 15.27. Функции upcase и lowcase

(upcase <строка>) (lowcase <строка>)

Варианты использования функций upcase и iowcase приведены в примере 15.7 и на рис. 15.7.

Пример 15.7. Использование функций upcase и lowcase

```
(upcase  "This is a test of upcase")
(lowcase        "This is a test of lowcase")
(upcase  A_Word_Test_for_Upcase)
(lowcase        A Word Test for Lowcase)
```

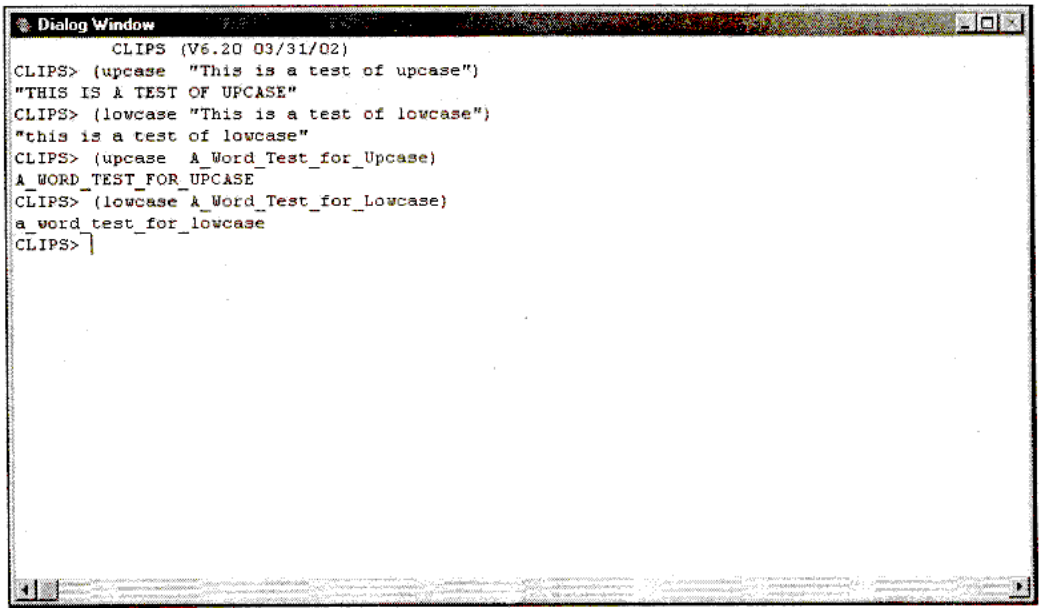

Рис. 15.7. Использование функций upcase и lowcase

Функция str-compare сравнивает две строки и определяет их логические отношения (т. е. "равно", "больше чем", "меньше чем"). Сравнение выполняется посимвольно до конца строк (если строки равны), либо пока не встретятся два неравных символа. Функция возвращает целое число, представляющее результат сравнения. Если сравниваемые строки равны, результатом является 0. Если первая строка меньше второй, результат — целое число меньше 0, и, наконец, если первая строка больше второй, возвращаемый результат – целое число больше 0. Аргументы функции должны принадлежать типу string или symbol.

Определение 15.28. Функция str-compare

(str-compare <строка1> <строка2>)

Для определения длины строки, заданной типом string или symbol, CLIPS предоставляет функцию str-length. Результат работы этой функции возвращается в виде целого числа.

Определение 15.29. Функция str- length

(str-length <строка>)

Варианты использования функций str-compare и str-length приведены в примере 15.8 и на рисунке 15.8.

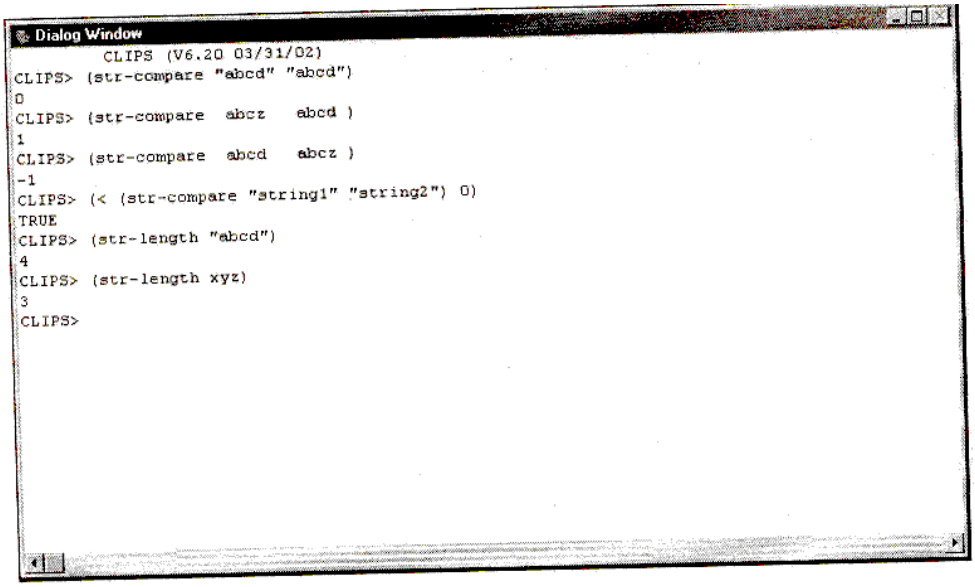

Рис. 15.8. Использование функций str-compare и str-length

Пример 15.8. Использование функций str-compare и str-length

```
(str-compare "abcd" "abcd")
(str-compare abcz  abcd)
(str-compare abcd  abcz)
 (< (str-compare "stringl" "string2") 0)
 (str-length "abed")
 (str-length xyz)
```

Функция check-syntax позволяет проверить текст, заданный строкой, на наличие синтаксических и семантических ошибок языка CLIPS. Функция возвращает значение FALSE, если в тексте не было найдено ошибок и неверных речевых конструкций. Значение MISSING-LEFT-PARENTHESIS возвращается, если первый значащий символ строки не является открывающей круглой скобкой. Значение EXTRANEOUS-INPUT-AFTER-LAST- PARENTHESIS означает, что в строке после заключительной скобки выражения или вызова функции присутствуют дополнительные символы. В случае обнаружения другой ошибки функция возвращает составную величину с двумя полями. Первое поле содержит строку с текстом сообщения об ошибке (или значение FALSE, если ошибок нет). Второе поле — строка, содержащая текст предупреждения (или значение FALSE, если предупреждения отсутствуют).

Определение 15.30. Функция check-syntax

(check-syntax <строка>)

Варианты использования функции check-syntax приведены в примере 15.9 и на рис. 15.9.

Пример 15.19 . Использование функции check-syntax.

```
(check-syntax "(defrule Examplel =>)")
(check-syntax "(defrule Example2 =>) foo")
(check-syntax " defrule Examples =>)")
(check-syntax "(defrule Example4 (3) =>)")
```

Функция string-to-field предназначена для разбора аргумента, имеющего тип string или symbol, на отдельные поля и преобразования их к одному из примитивных типов данных CLIPS.

Определение 15.31. Функция string-to-field

(string-to-field <строка>)

Данная функция преобразует и возвращает в качестве результата только первое поле указанной строки. Использование этой функции эквивалентно вызову функции read и вводу с клавиатуры или из файла соответствующей строки.

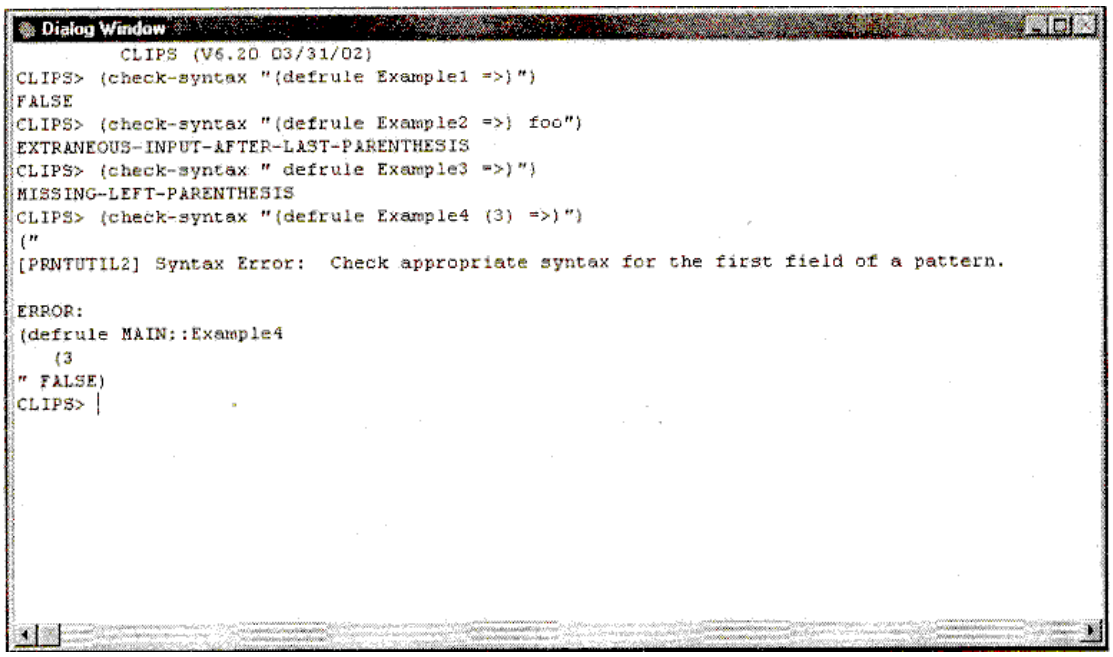

Рис. 15.9. Использование функции check-syntax

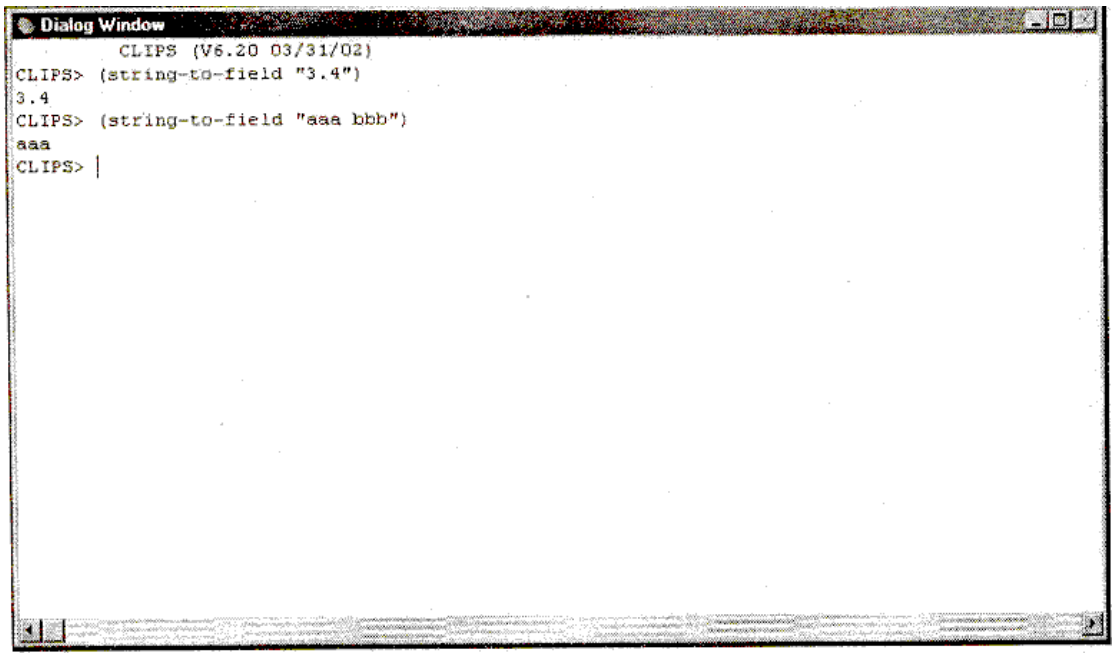

Рис. 15.10. Использование функции string-to-field

Варианты использования функции string-to-field приведены в примере 15.10 и на рис. 15.10.

Пример 15.10. Использование функции string-to-field

```
(string-to-field "3.4")
(string-to-field  "aaa bbb")
```

15.4. Функции работы с составными величинами

Составные величины или составные поля являются одной из отличительных особенностей среды CLIPS. Для работы с такими величинами предназначена специальная группа функций, краткое описание которых приведено в табл. 15.7.

Таблица 15.7. *Функции работы составными величинами*

Функция	Описание
create$	Создание составной величины
nth$	Получение конкретного элемента составной величины
member$	Поиск конкретного элемента составной величины
subsetp	Определение, не является ли одна составная величина подмножеством другой составной величины
delete$	Удаление конкретного элемента составной величины
explode$	Создание составной величины из строки
implode$	Создание строки из составной величины
subseq$	Извлечение подпоследовательности из составной величины
replace$	Замена элемента составной величины
insert$	Добавление новых элементов в составную величину
first$	Получение первого элемента составной величины
rest$	Получение остатка составной величины
length$	Определение числа элементов составной величины
delete-member$	Удаление заданных элементов составной величины
replace-member$	Замена заданных элементов составной величины

Функция create$ объединяет заданное количество выражений для создания составной величины. Независимо от количества полей, получившихся в результате, возвращаемое функцией значение всегда является составной величиной. Вызов функции без аргументов возвращает составную величину с нулевой длиной.

Определение 15.32. Функция create$

(create$ <выражение>*)

Для получения конкретного поля составной величины предназначена функция nth$.

Определение 15.33. Функция nth$

(nth$ <целое> <составная-величина>)

Первый аргумент данной функции должен быть целым числом, большим или равным 1, который определяет индекс поля в составной величине, заданной вторым аргументом. Если заданное число больше количества элементов в составной величине, функция вернет значение nil. Значение, возвращаемое функцией, в любом случае является значением типа symbol.

Функция member$ возвращает индекс поля, если оно содержится в составной величине. Если первый аргумент этой функции — простая величина, которая является каким-либо полем второго аргумента, то функция member$ вернет целое число — индекс соответствующего поля. Если первый аргумент — составная величина, и она представляет собой часть второго аргумента, тогда функция возвращает два индекса — начала и конца первой составной величины во второй величине. В противном случае функция возвращает значение FALSE.

Определение 15.34. Функция member$

(memberS <выражение> <составная-величина>)

Функция subsetp проверяет, не является ли одна составная величина подмножеством другой, т. е. содержатся ли все поля первой составной величины и во второй составной величине. Если первая составная величина является подмножеством второй, функция возвращает значение TRUE, в противном случае — FALSE. Порядок полей не оказывает влияния на работу функции. В случае если первый аргумент имеет нулевую длину, функция subsetp всегда возвращает значение TRUE.

Определение 15.35. Функция subsetp

(subsetp <составная-величина1> <составная-величина2>)

В примере 15.11 и на рис. 15.11 приведены варианты использования описанных функций для работы с составными величинами.

Пример 15.11. Работа с составными величинами

```
(create$ hammer drill saw screw pliers wrench)
(create$ (+ 34) (* 2 3) (/84))
(nth$ 3 (create$ a b c d e f g))
(nth$ 10 (create$ a b c d e f g))
(member$ blue (create$ red 3 "text" 8.7 blue))
(member$ 4 (create$ red 3 "text" 8.7 blue))
(member$ (create$ b c) (create$ abed))
(subsetp
        (create$ hammer saw drill)
        (create$ hammer drill wrench pliers saw))
(subsetp
        (create$ wrench crowbar)
        (create$ hammer drill wrench pliers saw))
```

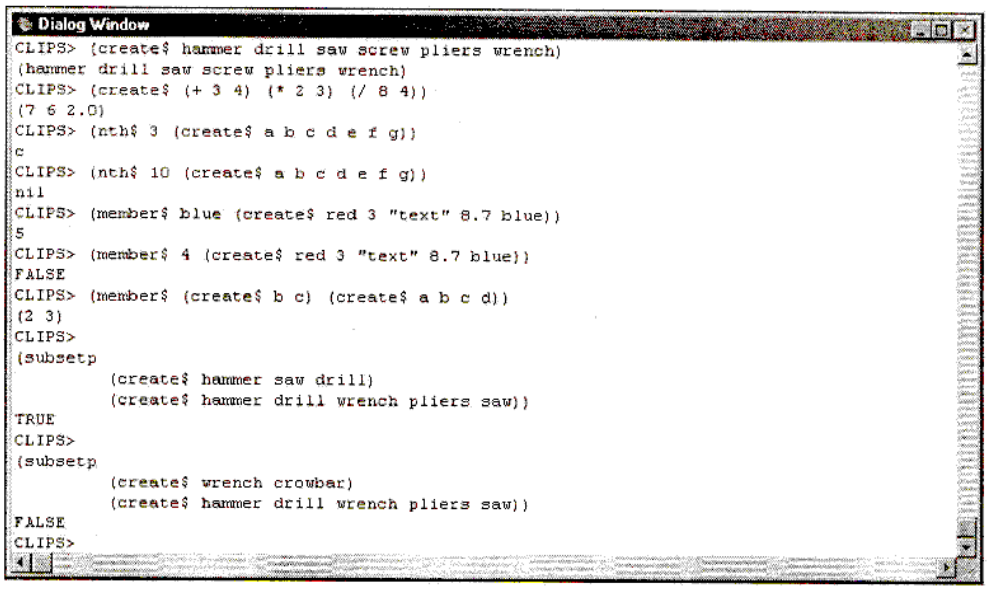

Рис. 15.11. Работа с составными величинами

Функции explode$ и implode$ предназначены для преобразования строки в составную величину и составной величины в строку соответственно. Пустая строка создает составную величину нулевой длины. Элементы строки типов, отличных от symbol, string, integer, float или instance-name (например, переменные), преобразуются в тип string. Синтаксис и примеры использования этих функций приведены ниже (см. также рис. 15.12).

Определение 15.36. Функции explode$ и implode$

(explode$ <строковое выражение>)
(implode$ <составная-величина>)

Пример 15.12. Использование функций explode$ и implode$

```
(explode$     "hammer drill  saw  screw")
(explode$     "1  2  abc  3  4   \"abc\"   \"def\"")
(explode$     "?x ~)")
(implode$     (create$  hammer drill  screwdriver))
```

```
(implode$      (create$  1  "abc" def  "ghi"  2))
(implode$      (create$  "abc def ghi"))
```

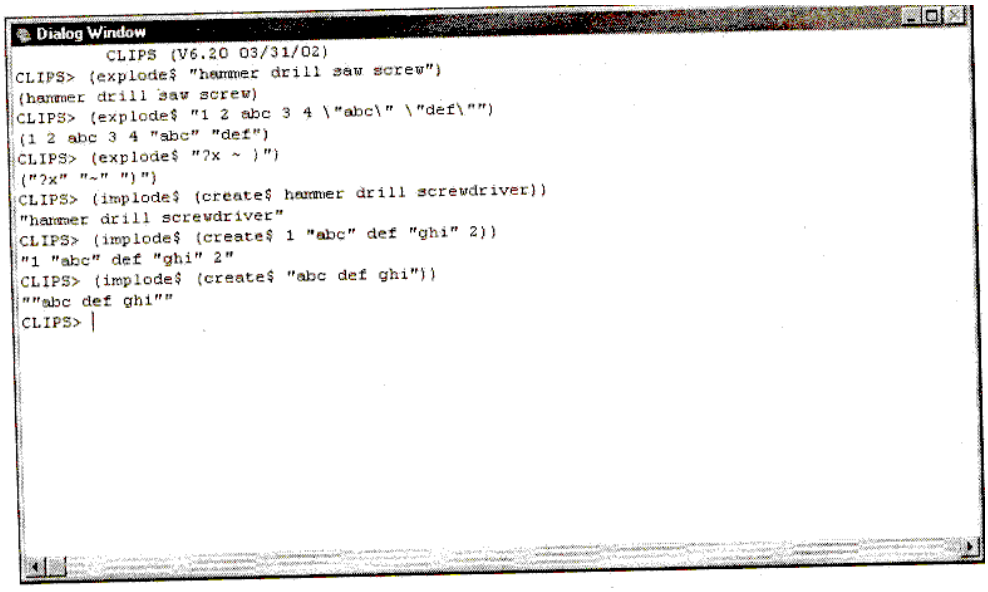

Рис. 15.12. Использование функций explode$ и implode$

Функция delete$ удаляет выбранные поля из составной величины и возвращает модифицированную составную величину, из которой удален отрезок, заданный индексами своего первого и последнего элемента. Если требуется удалить одно поле, то индекс начала отрезка должен совпадать с индексом конца.

Определение 15.37. Функция delete$

(delete$ <составная-величина> <индекс-начала> <индеке-конца>)

Для извлечения подпоследовательности из составной величины служит функция subseq$. Определение границ извлекаемой подпоследовательности происходит так же, как и у функции delete$.

Определение 15.38. Функция subseq$

(subseq$ <составная-величина> <индекс-начала> <индекс-конца>)

Функция replace$ предназначена для замены выбранного диапазона элементов составной величины на заданную простую или составную величину. Функция возвращает новую составную величину, содержащую измененный фрагмент.

Определение 15.39. Функция replace$

(replace$ <изменяемая-составная-величина> <индекс-начала>
<индекс-конца> <простая-или-составная-величина>+)

Для добавления нескольких простых или составных величин в заданное место некоторой составной величины предназначена функция insert$.

Определение 15.40. Функция insert$

(insert$ <изменяемая-составная-величина> <индекс-начала> <простая-или-составная-величина>+)

В качестве второго параметра эта функция принимает целое число, являющееся индексом, начиная с которого в заданную составную величину будут добавлены новые поля. Примеры использования этой и остальных функций, изменяющих содержание составной величины, приведены ниже (см. также рис. 15.13).

Пример 15.13. Использование функций, изменяющих содержание составной величины

```
(delete$  (create$ hammer drill saw pliers wrench) 3 4)
(delete$  (create$ computer printer hard-disk) 1 1)
(subseq$ (create$ hammer drill wrench pliers) 3 4)
(subseq$ (create$ 1 "abc" def "ghi" 2) 1 1)
(replace$  (create$ drill wrench pliers)  3 3 machete)
(replace$  (create$ abed)  2 3 x y  (create$ q r s))
(insert$  (create$ abed)  1 x)
(insert$  (create$ abed)  4 y z)
(insert$  (create$ abed)  5  (create$ q r))
```

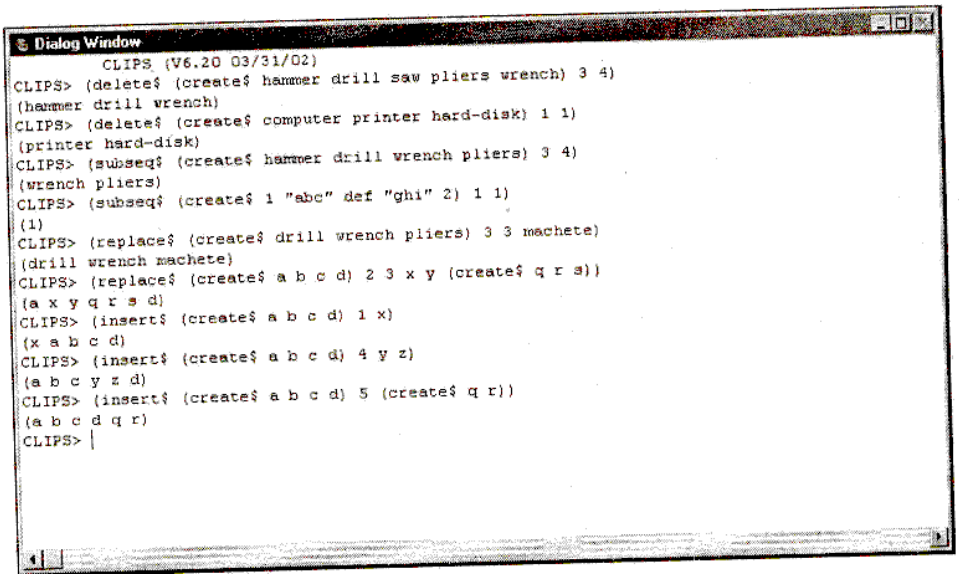

Рис. 15.13. Использование функций, изменяющих содержание составной величины

Для работы с составными переменными в рамках парадигмы списков, которая часто используется в различных логических языках программирования, например Пролог, CLIPS предоставляет функции first$ и rest$. Синтаксис этих функций и примеры приведены ниже (см. также рис. 15.14).

Определение 15.41. Функции first$ и rest$

```
(first$  <составная-величина>)
(rest$     <составная-величина>)
```

Первая функция возвращает в качестве составного значения первое поле заданной составной величины, а вторая — заданную составную величину без первого поля.

Пример 15.14. Использование функций first$ и rest$

```
(first$ (create$ abc))
(first$ (create$))
 (rest$ (create$ a b c))
 (rest$ (create$))
```

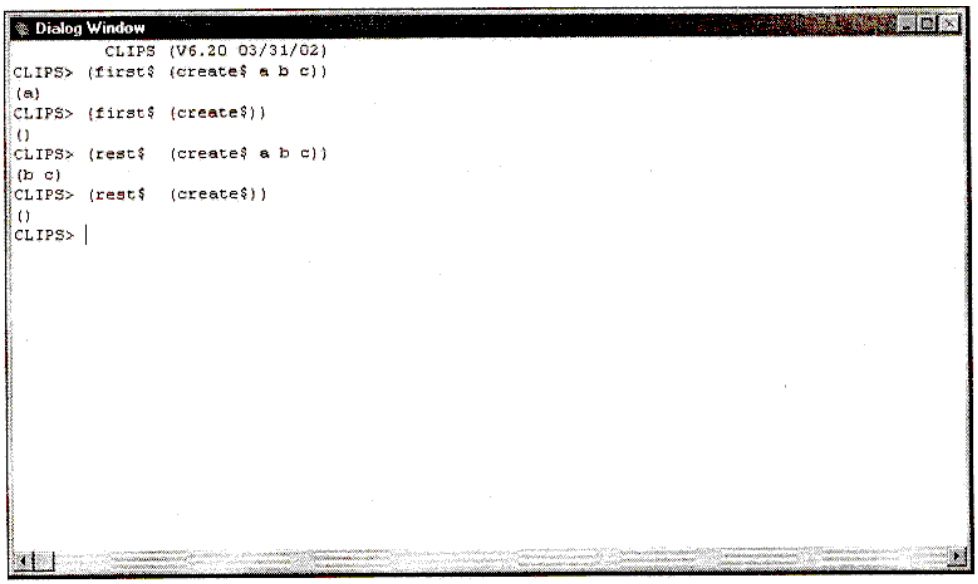

Рис. 15.14. Использование функций first$и rest$

Функция length$ возвращает число полей (целый тип), содержащихся в составной величине. Если параметр length$ не соответствует необходимому типу, функция возвращает — 1.

Определение 15.42. Функция length$

(length$ <составная-величина>)

Функция delete-member$ удаляет все вхождения заданных пользователем элементов или составных величин из начальной составной величины.

Определение 15.43. Функция delete-member$

(delete-member$ <составная-величина> <выражение>+)

Для замены всех вхождений элементов или составных величин из начальной составной величины на некоторое выражение служит функция replace-member$. Первым аргументом этой функции является исходная составная величина. Второй аргумент задает простое или составное значение, на которое будут заменены все найденные вхождения. Третий аргумент определяет произвольное количество простых или составных элементов, которые необходимо найти и заменить в исходной составной величине.

Определение 15.44. Функция replace-member$

(replace-member$ <составная-величина> <выражение> <>)

Ниже представлены примеры использования функций delete-member$ и replace-member$ (см. также рис. 15.15).

Пример 15.15. Использование функций delete-member$ и replace-member$

```
(delete-member$ (create$ a b a c) b a)
(delete-member$ (create$ a b c c b a) (create$ b a))
(replace-member$ (create$ a b a b) (create$ a b a) a b)
(replace-member$ (create$ a b a b) (create$ a b a) (create$ a b))
```

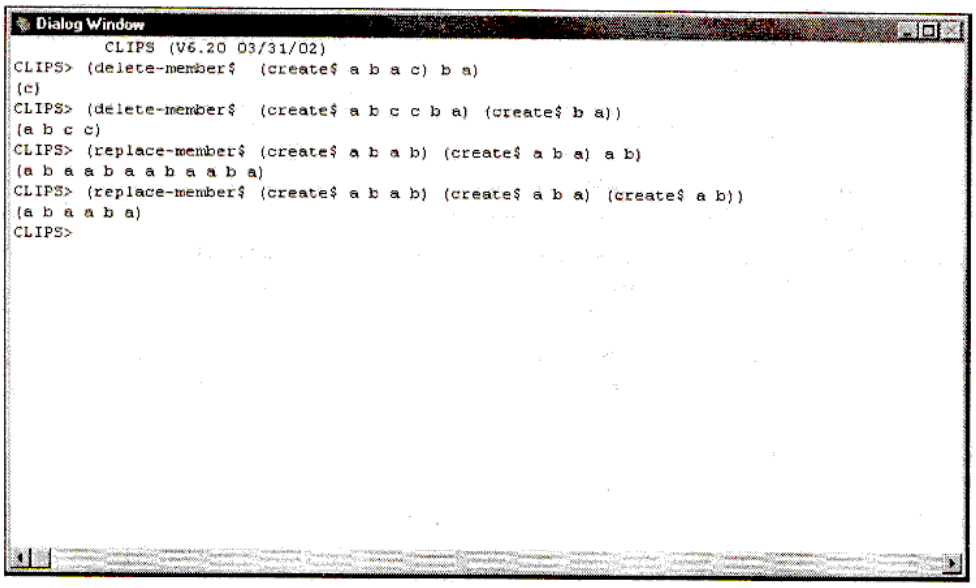

Рис. 15.15. Использование функций delete-member$ и replace-member$

15.5. Функции ввода/вывода

Система ввода/вывода, используемая CLIPS, называется *маршрутизацией ввода/вывода* (I/O routers). Одна из ключевых особенностей системы маршрутизации ввода/вывода — использование *логических имен*. Логическиеимена позволяют обращаться к устройствам ввода/вывода унифицированным способом, без необходимости учета особенностей конкретного устройства. Многие функции CLIPS используют логические имена в качестве параметров. Логическое имя, созданное пользователем, должно быть строкой, числом или принадлежать типу symbol. CLIPS предоставляет несколько предопределенных логических имен, список которых приведен в табл. 15.8.

Таблица 15.8. *Предопределенные логические имена CLIPS*

Имя	**Описание**
stdin	Логическое имя, определяющее устройство по умолчанию, предназначенное для ввода информации от пользователя. Если такое имя определено, его используют функции read и readln
stdout	Логическое имя, определяющее устройство по умолчанию, предназначенное для вывода информации для пользователя. Если такое имя определено, его используют функции printout и format
wclips	Логическое имя, определяющее устройство, которое использует справочная система CLIPS
wdialog	Устройство, ассоциированное с данным логическим именем, служит для отправки пользователю информационных сообщений
wdisplay	Отображение такой информации, как правила, факты и т. д., производится на устройство, ассоциированное с этим логическим именем
werror	Данное логическое имя определяет устройство, на которое будут выводиться все сообщения об ошибках
wwarning	Устройство, ассоциированное с данным логическим именем, используется для отображения предупреждений
wtrase	Вся отладочная информация посылается на устройство, ассоциированное с данным логическим именем

Подробную информацию о логических именах и системе маршрутизации ввода/вывода можно найти в книге *"CLIPS Reference Manual, Volume II, Advanced Programming Guide "*.

Описание функций ввода/вывода, предоставляемых системой CLIPS, приведены в табл. 15.9.

Таблица 15.9. *Функции ввода/вывода*

Функция	**Описание**
open	Открытие файла
close	Закрытие файла
printout	Вывод информации на заданное устройство
read	Ввод данных с заданного устройства
readline	Ввод строки с заданного устройства
format	Форматированный вывод информации на заданное устройство
rename	Переименование файла
remove	Удаление файла

Функция open предоставляет пользователю возможность открыть файл в функции или из правой части правила и присвоить файлу определенное логическое имя. Эта функция принимает три аргумента: имя открываемого файла, логическое имя, которое будет использоваться системой ввода/вывода CLIPS для связи с этим файлом и, наконец, параметр, определяющий способ открытия файла. Возможные варианты значений аргумента, определяющего способ открытия файла, приведены в табл. 15.10.

Таблица 15.10. *Способы открытия файла*

Значение	**Описание**
"r"	Доступ только для чтения
"w"	Доступ только для записи
"r+"	Доступ для чтения и записи
"a"	Добавление только в конец файла
"wb"	Возможность записи двоичных файлов

Определение 15.45. Функция open

(open <имя-файла> <логическое-имя> [<параметр-открытия>]

Параметр <имя-файла> может принадлежать типу string или symbol и включать в себя как полный, так и относительный путь к файлу. Если в качестве данного параметра используется строка, то обратная косая черта (\) и некоторые другие специальные символы должны быть записаны при помощи дополнительного знака обратной косой черты. Логическое имя не должно использоваться до его определения *с* помощью функции open. Если параметр, определяющий способ открытия файла, не определен, то по умолчанию файл будет открыт только для чтения. Функция open возвращает значение TRUE, если открытие файла прошло успешно. В противном случае возвращается значение FALSE. Ниже приведено насколько примеров использования функции open.

Пример 15.16. Использование функции open

```
(open "myfile.clp" writeFile "w")
(open "MS-DOS\\directory\\file.clp" readFile)
```

Функция close закрывает файл, открытый ранее функцией open. Конкретный файл определяется при помощи логического имени, присвоенного ему при открытии.

Определение 15.46. Функция close

(close [<логическое-имя>])

Если функция close вызвана без аргументов, CLIPS закроет все открытые файлы. Если файл, открытый пользователем, не был закрыт, возможна потеря последних изменений, сделанных в файле. Поэтому CLIPS пытается закрыть все открытые файлы при выполнении exit. Функция close возвращает значение TRUE, если файл был благополучно закрыт, иначе она возвращает значение FALSE. Функция printout позволяет выводить информацию на устройство, связанное с указанным логическим именем. Логическое имя должно быть определено на момент использования функции, а устройство должно быть предварительно приготовлено для работы. Для отправки информации на устройство, связанное с логическим именем stdout, обычно используется его синоним — символ t. Если указано логическое имя nil, функция printout не выполняет никаких действий.

Определение 15.47. Функция printout

(printout <логическое-имя> <выражение>+)

Функция printout принимает и выводит на устройство, ассоциированное с заданным логическим именем, любое число параметров. Каждое выражение вычисляется и отправляется на соответствующее устройство без пропусков между ними. Символ crlf служит для перевода каретки на следующую строку. Символы tab, vtab, и ff позволяют осуществлять табуляцию,вертикальную табуляцию и переход на новую страницу. Поведение функции printout при указании этих специальных символов может меняться в зависимости от используемой операционной системы. Функция read позволяет считывать очередную порцию данных с устройства, ассоциированного с заданным логическим именем.

Определение 15.48. Функция read

(read [<логическое-имя>])

Параметр <логическое-имя> является необязательным. Если он определен, функция попытается считывать информацию из присоединенного к логическому имени файла. Если параметр <логическое-имя> равняется t или не определен, функция будет считывать данные из устройства, связанного с stdin. Для отделения друг от друга элементов считываемых данных служат разделители, описанные в *гл. 4*. Функция read всегда возвращает значение одного из примитивных типов данных. Пробелы, символы возврата каретки и табуляция воспринимаются только как разделители и не содержатся в полученном результате (если они не заключены в двойные кавычки как часть строки). Если в процессе чтения был достигнут конец файла, функция вернет значение EOF. Если при чтении произошли ошибки, будет возвращено значение "*** READ ERROR ***".

Функция readline подобна функции read. Однако, в отличие от нее, функция readline позволяет при каждом использовании получать строку целиком. Другими словами, для функции readline разделителями являются только символ возврата каретки, точка с запятой или символ конца файла (EOF). Пробелы, табуляция и другие разделители воспринимаются функцией как часть строки. Результатом работы функции readline является строка.

Определение 15.49. Функция readline

readline [<логическое-имя>])

Для осуществления форматированного вывода на устройство, связанное с заданным логическим именем, служит функция format. Она может быть использована вместо функции printout, если необходимо специальное форматирование выводимой информации. Несколько более сложная функция format предоставляет программисту гораздо больший набор возможностей для форматирования данных. Функция format всегда возвращает строку, содержащую форматированный вывод. При необходимости получения форматированной строки без записи ее на устройство вывода можно использовать логическое имя nil.

Определение 15.50. Функция format

(format <логическое-имя> <строковое-выражение> <выражение>+)

Первый аргумент функции определяет имя логического устройства, на которое осуществляется вывод. Второй аргумент — строку, которая должна быть выведена и отформатирована. Последний аргумент представляет собой список параметров для форматирования, который управляет процессом вывода строки. Выводимая строка состоит из текста и флагов форматирования. Каждому такому флагу должен предшествовать знак процента. Общий вид флага форматирования можно представить следующим образом: %-M.Nx.

Здесь x является одним из флагов, перечисленных в табл. 15.11, знак -представляет собой произвольный флаг выравнивания, а M и N — необязательные параметры, которые определяют ширину поля и количество разрядов десятичного числа. Если параметр M задан, данные будут выведены в поле указанной длины. В случае если M начинается с нуля (например, 07), символ 0 используется в качестве заполнителя пустого пространства. Если перед M стоит минус, то значение будет выровнено по левому краю, иначе используется правое выравнивание. Если параметр N не определен, по умолчанию выводится шесть знаков после запятой.

Таблица 15 11. *Способы открытия файла*

Флаг форматирования	**Описание**
c	Отображает параметр как простой символ
d	Отображает параметр как число типа long integer (параметр N не учитывается)
f	Отображает параметр как число с плавающей точкой
e	Отображает параметр как число с плавающей точкой в экспоненциальной форме
g	Отображает параметр в экспоненциальной форме
o	Отображает параметр как восьмеричное число без знака (параметр N не учитывается)
x	Отображает параметр как шестнадцатеричное число без знака (параметр N не учитывается)
s	Отображает параметр как строку, взятую в кавычки (параметр N определяет максимальное число символов, которые могут быть напечатаны)
n	Помещает в вывод новую строку
r	Помещает в вывод возврат каретки
%	Помещает в вывод символ процента

Ниже приведено несколько примеров использования функции format (см. также рис. 15.16).

Пример 15.17. Использование функции format

```
(format nil "Integer: |%1d|" 12)
(format nil "Integer: |%41d|" 12)
(format nil "Integer: |%-041d|" 12)
(format nil Float: |%f|" 12.01)
(format nil Float: |%"7.2f │'' 12.01)
```

```
(format nil Test: |%e|"12.01)
(format nil Test: |%7.2e|"12.01)
(format nil "General: |%g|" 1234567890)
(format nil "Hexadecimal: |%x│" 12)
(format nil "Octal: │%o|" 12)
(format nil "Symbols: │%s| │ %s │ " value-a1 capacity)
```

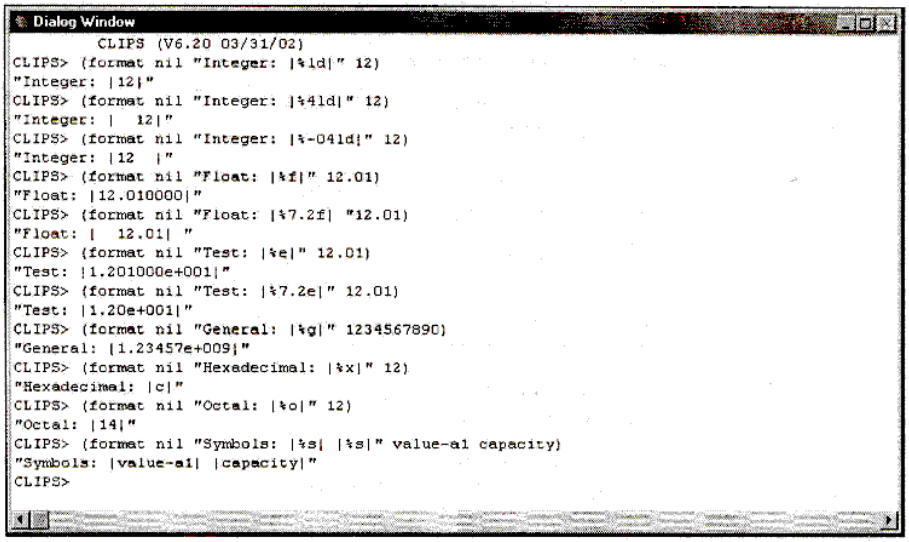

Рис.15.16. Использование функции format

Функция rename используется для изменения имени файла. Оба параметра этой функции (<старое-имя> и <новое имя>) должны быть значениями типа string или symbol и могут содержать полный путь к файлу. Если для задания параметров используется строка, то символ обратной косой черты (\) и другие специальные символы в параметрах <старое-имя> и <новое имя> должны быть записаны при помощи дополнительного знака обратной косой черты. Функция rename возвращает значение true, если операция изменения имени прошла успешно, и значение FALSE — в противном случае.

Определение 15.51. Функция rename

(rename <старое-имя> <новое-имя>)

Для удаления файла используется функция remove.

Определение 15.52. Функция remove

(remove <имя-файла>)

Здесь параметр <имя-файла> должен быть значением типа string или symbol и может содержать полный путь к файлу. Функция remove возвращает значение TRUE в случае успеха и значение FALSE — в случае неудачи.

15.6. Процедурные функции

CLIPS предоставляет 9 функций, которые реализуют возможности процедурного программирования, присущие таким стандартным языкам программирования, как Basic, Pascal, C, Ada. Использование этих функций позволят создавать отрезки процедурного кода в правилах и функциях, созданных с помощью конструктора deffunction. Краткое описание этих функций приведено в табл. 15.12.

Таблица 15.12. *Процедурные функции*

Функция	**Описание**
bind	Создание и связывание переменных
if	Оператор ветвления
while	Цикл с предусловием
loop-for-count	Итеративный цикл
progn	Объединение нескольких действий в рамках одной логической команды
prong$	Выполнение заданного набора действий над каждым элементом составного поля
return	Прерывание функции, цикла, правила, обработчика сообщения и т. д.
break	Прерывание текущей работы циклов, функций progn и progn$ и некоторых функций, выполняющих действия над набором объектов без возвращения параметров
switch	Оператор множественного ветвления

Если необходимо создание переменной или изменение значения уже существующей переменной, например в правой части правила, используется функция bind.

Определение 15.53. Функция bind

(bind <имя-переменной> <выражение>*)

Первый аргумент функции bind — <имя-переменной> — является именем глобальной или локальной переменной, созданной в правиле или функции. Переменная, определенная первым аргументом, будет связана со значением переданного функции выражения. Если заданная переменная еще не определена, она будет создана и связана с соответствующим значением. Помимо пользовательских функций и правил, функцию bind можно использовать в обработчиках сообщений для установки новых значений слотов объекта.

Если параметр <выражение> не определен, то выполнение функции bind не оказывает на локальные переменные никакого влияния, а глобальные переменные получают при этом значения по умолчанию. В случае если пользователь задал выражение, то его значение вычисляется и присваивается соответствующей переменной. Если задано несколько выражений, из их значений формируется составное поле, которое потом и будет связано с указанной переменной.

Функция bind возвращает значение FALSE при неудачном исходе операции. Во всех остальных случаях функция возвращает присвоенное переменной значение. Ниже приведено несколько примеров использования функции bind (см. также рис. 15.17 и 15.18).

Пример 15.18. Использование функции bind и глобальные переменные

```
(defglobal ?*x* =3.4)
?*x*
(bind ?*x*  ( + 8 9 ) }
?*x*
(bind ?*x*  (create$  a b c d )
?*x*
(bind  ?*x*  d e  f)
?*x*
(bind  ?*x*)
? *x*
```

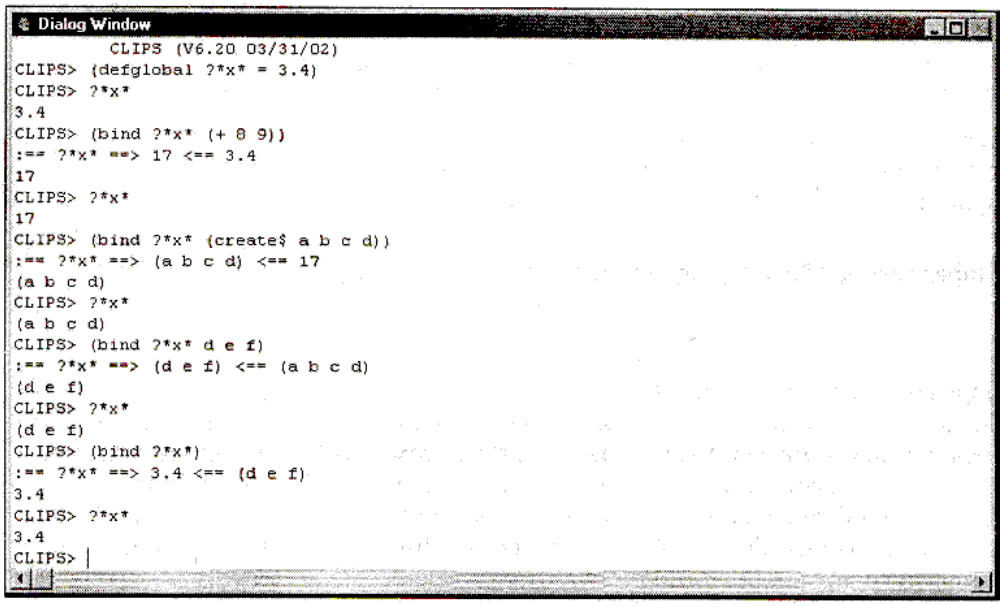

Рис. 15.17. Использование функции bind и глобальные переменные

Пример 15.19. Использование функции bind в обработчиках сообщений

```
(defclass A (is-a USER)
            (role concrete)
            (slot x)
            (slot y))
(defmessage- handler A init after ()
                        (bind ?self : x 3)
                        (bind ?self : y 4))
(make-instance a of A)
(send [a] print)
```

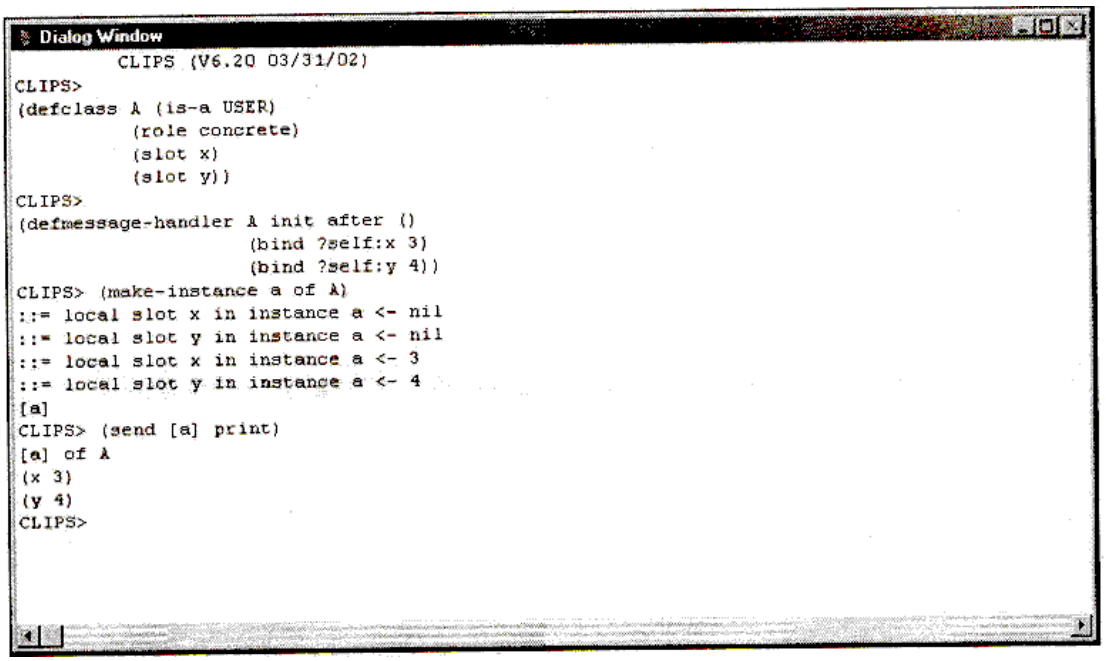

Рис. 15.18. Использование функции bind в обработчиках сообщений

Замечание

Для наглядности при демонстрации работы данных примеров были включены режимы просмотра изменений глобальных переменных и слотов объектов.

Функция if реализует стандартный оператор ветвления *"если...то...иначе",* применяемый практически во всех языках программирования. Эта функция позволяет задавать некоторое условие, в зависимости от выполнения (или невыполнения) которого будут выбраны те или иные действия.

Определение 15.54. Функция if

```
(if <выражение>
    then
        <действие>*
    [else
        <действие>*])
```

Если условие, заданное с помощью выражения, выполняется (т. е. не является ложным), выполняются действия, определенные в блоке then. В противном случае производятся действия из блока else. В каждом из таких блоков может быть задано любое количество действий. Любой блок может содержать вложенную конструкцию if.. .then.. .else. Блок else является необязательным. Значение, возвращаемое функцией if, равно значению последнего вычисленного выражения или выполненного действия. Ниже приведен пример использования функции if.

Пример 15.20. Использование функции if

```
(defrule closed-valves (temp high) (valve ?v closed)

        (if ( = ?v 6) then
            (printout t "The special valve " ?v " is closed!" crlf) (assert (perform special operation))
             else
                (printout t "Valve " ?v " is normally closed" crlf)))
```

Замечание

Обычно в подобных случаях предпочтительней использование двух правил, как показано в примере 15.21.

Пример 15.21. Альтернатива использованию функции if *в* правилах

```
(defrule  closed-valves-number-6
            (temp high)
            (valve  6 closed)
             =>
            (printout t  "The special valve  6 is closed!"  crlf)
            (assert   (perform special  operation)))
(defrule  closed-valves-other-than-6
            (temp high)
            (valve  ?v&~6  closed)
            =>
            (printout  t  "Valve  "   ?v  "  is normally closed"  crlf))
```

Функция while позволяет выполнять простой цикл с предусловием.

Определение 15.55. Функция while

```
(while <выражение> [do]
      <действие>*)
```

Для задания условия в выражении цикла while могут быть использованы любые предикатные функции. В теле цикла может содержаться произвольное количество действий, включая вложенные циклы или функцию if. Проверка условия выполняется перед выполнением тела цикла. Цикл выполняется до тех пор, пока условие остается истинным. Определение цикла while может содержать необязательный символ do между условием и первым действием тела цикла. Для прерывания работы цикла могут быть использованы функции break и return, описанные

ниже. Если для прерывания работы цикла не использовалась функция return, функция while возвращает значение FALSE.

Помимо функции while CLIPS предоставляет также функцию loop-for-count, которая реализует концепцию простого итеративного цикла, выполняющего определенные действия заданное число раз.

Определение 15.56. Функция loop-for-count

(loop-for-count <диапазон> [do] <действие>*)
<диапазон>::= <максимальное-значение-индекса> |(<переменная-цикла> [<минимальное-значение-индекса><максимальное-значение-индекса>]) | (<переменная-цикла> [<максимальное-значение-индекса>])
<минимальное-значение-индекса> ::= <целочисленное-выражение>
<максимальное-значение-индекса> ::= <целочисленное-выражение>

Цикл loop-for-count производит указанные действия некоторое количество раз согласно заданному параметру <диапазон>. Если параметр <минимальное-значение-индекса> не задан, ему автоматически присваивается 1. CLIPS предусматривает наличие параметра <переменная-цикла>, определяющего имя локальной переменной, которая может использоваться в теле цикла для определения текущего числа итераций. В теле цикла могут быть указаны любые внешние по отношению к циклу переменные (как глобальные, так и локальные). Однако при совпадении имени переменной цикла с какой-нибудь внешней переменной внешняя переменная скрывается. Использование переменной цикла после завершения работы функции loop-for-count вне тела цикла запрещено. Определение цикла может содержать необязательный символ do между условием и первым действием тела цикла. Если параметр <минимальное-значение-индекса> изначально больше параметра <максимальное-значение-индекса>, тело цикла не будет выполнено ни разу. Функции break и return могут быть использованы для экстренного прерывания работы цикла. В теле цикла может содержаться произвольное количество действий, включая вложенные циклы или функцию if. Функция loop-for-count возвращает значение FALSE, если для экстренного прерывания ее работы не использовалась функция return. Пример функции loop-for-count приведен ниже (см. также рис. 15.19).

Пример 15.22. Использование функции loop-for-count

```
(loop-for-count 2
                (printout t "Hello world" crlf))
 (loop-for-count (?cntl 2 4) do
           (loop-for-count (?cnt2 1 3) do
                (printout t ?cntl " " ?cnt2 crlf)))
```

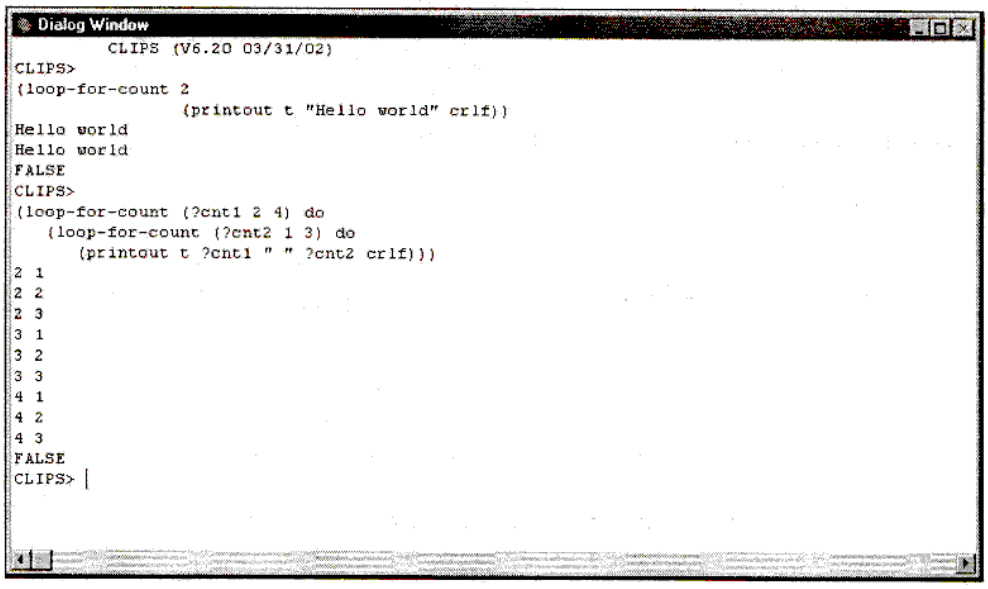

Рис. 15.19. Использование функции loop-for-count

Функция progn предназначена для выполнения нескольких вычислений (действий) в рамках одной команды, т. е. эта функция позволяет группировать и объединять набор действий в одну логическую команду.

Определение 15.57. Функция progn

(progn <действие>*)

Функция progn выполняет все действия, заданные в качестве аргументов, и возвращает результат выполнения последнего.
В отличие от progn функция progn$ предназначена для выполнения заданного набора действий над каждым элементом составного поля.

Определение 15.58. Функция progn$

(progn$ <определение-списка> <действие>*)

<определение-списка> ::=<составное-поле>| (<переменная-списка> <составное-поле>)

Текущее обрабатываемое поле может быть определено с помощью переменной списка, если она задана, а индекс текущего поля — с помощью переменной <переменная-списка>-index. Значение, возвращаемое функцией progn$, является результатом последнего действия над последним полем составной величины (см. пример 15.23 и рис. 15.20).

Пример 15.23. Использование функции progn$

```
(progn$  (?field  (create$  abc  def ghi))
         (printout  t "—>  "  ?field "  "  ?field-index  " <—"  crlf))
```

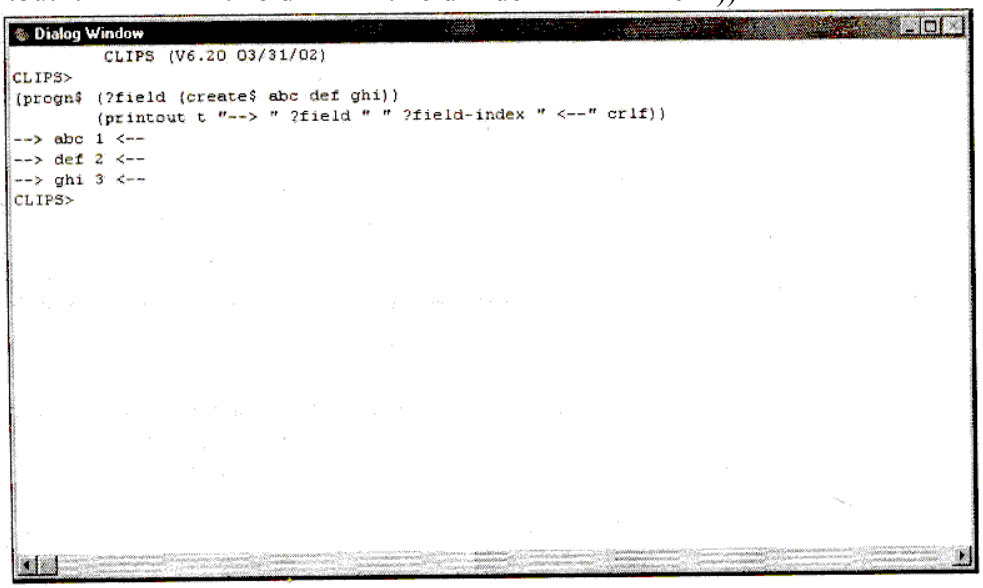

Рис. 15.20. Использование функции progn$

Функция return прерывает выполняющуюся функцию, цикл, правило, обработчик сообщения и т. д. Если функция return вызывается без аргументов, функция не возвращает никакого значения. Однако если аргумент присутствует, return возвращает результат вычислений, который присваивается значению прерванной функции, цикла или обработчика сообщения.

Определение 15.59. Функция return

(return [<выражение>])

При использовании функции return в правой части правила текущий фокус удаляется из стека фокусов. Эта функция не может быть указана в качестве аргументов другой функции. Допустимо применение return в функциях, выполняющих действия над набором объектов (do-for-instance, do-for-all-instances и delayed-do-for-all-instances). Однако в этом случае она выполняется, только если подобные действия допустимы окружением соответствующей функции.

Функция break прерывает текущую итерацию циклов while и loop-for-count, работу функций progn и progn$ и некоторых функций, выполняющих действия над набором объектов (do-for-instance, do-for-all-instances и delayed-do-for-all-instances). Функция break не должна использоваться в рамках функции progn, если это является недопустимым, исходя из внешнего контекста progn. Кроме того, функция break не должна указываться в качестве параметра обращения к другой функции.

Определение 15.60. Функция break

(break)

Функция switch реализует оператор множественного ветвления и позволяет связать определенную группу действий (среди нескольких подобных групп) с некоторой заданной величиной. При выборе этой величины выполняет связанные с ней действия.

Определение 15.61. Функция switch

```
(switch <выражение>)
        <условие-ветвления>*
        [<условие>])
<условие-ветвления>          ::= (case <выражение> then <действие>*)
<действия-по-умолчанию>  ::= (default действие>*)
```

Для эффективного применения функции switch необходимо наличие, по крайней мере, трех альтернативных групп действий, зависящих от заданного условного выражения.

Функция switch в первую очередь проводит вычисление аргумента <выражение>, а затем сравнивает его со всеми условиями ветвления по очереди. Если значение заданного выражения совпадает с одним из условий ветвления, выполняются соответствующие действия, и работа функции завершается. Если совпадений обнаружить не удалось, функция возвращает результат последнего сеанса выбора (если таковой имеется) или значение FALSE. Ниже приведен пример использования функции switch (см. также рис. 15.21).

Пример 15.24. Использование функции switch

```
(defglobal ?*x* = 0)
(defglobal ?*y* = 1)
(def function foo (?val)
                (switch ?val
                        (case ?*x* then *x*)
                        (case ?*y* then *y*)
                        (default none) ) )
(foo 0)
(foo 1)
(foo 2)
```

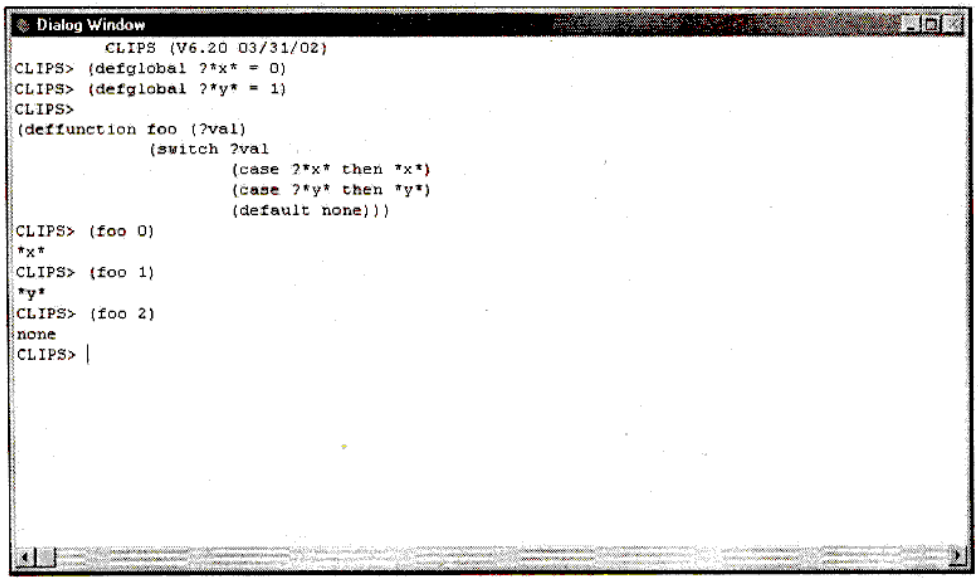

Рис. 15.21. Использование функции switch

15.7. Работа с родовыми функциями

Список функций, предоставляемых CLIPS для работы с методами родовых функций, а также краткое описание их назначения приведен в табл. 15.13.

Таблица 15.13. *Работа с родовыми функциями*

Функция	**Описание**
get-defgeneric-list	Получение списка существующих родовых функций
get-defmethod-list	Поиск модуля, в котором определена родовая функция
defgeneric-module	Получение списка существующих методов родовых функций
type	Определение типа получаемого параметра
next-methodp	Проверка существования скрытых методов родовой функции
call-next-method	Вызов скрытого метода родовой функции
override-next-method	Вызов скрытого метода родовой функции с измененными аргументами
call-specific-method	Вызов определенного метода родовой функции
get-method-restrictions	Получение ограничений для заданного метода родовой функции

Функция get-defgeneric-list возвращает составную величину, содержащую имена всех определенных в системе родовых функций, видимых в заданном модуле. Если параметр <имя-модуля> не задан, функция выведет список родовых функций, определенных в текущем модуле. Вместо имени модуля допустимо использовать символ *. В этом случае возвращается список всех родовых функций.

Определение 15.62. Функция get-defgeneric-list

(get-defgeneric-list [<имя-модуля>])

Функция defgeneric-module служит для определения модуля, в котором указана заданная родовая функция.

Определение 15.63. Функция defgeneric-module

(defgeneric-module <имя-родовой-функции>)

Для определения списка методов и индексов, определенных в системе родовых функций, видимых в заданном модуле, служит функция get-defmethod-list. Если необязательный параметр <имя-родовой-функции> не указан, будет получен список методов всех родовых функций.

Определение 15.64. Функция get-defmethod-list

(get-defmethod-list [<имя-родовой-функции>])

Функция type служит для определения типа или класса заданного аргумента (см. также пример 15.25 и рис. 15.22).

Определение 15.65. Функция type

(type <выражение>)

Пример 15.25. Использование функции type

```
(type (+22))
(defclass CAR (is-a USER) (role concrete))
(make-instance Rolls-Royce of CAR)
(type Rolls-Royce)
(type [Rolls-Royce])
```

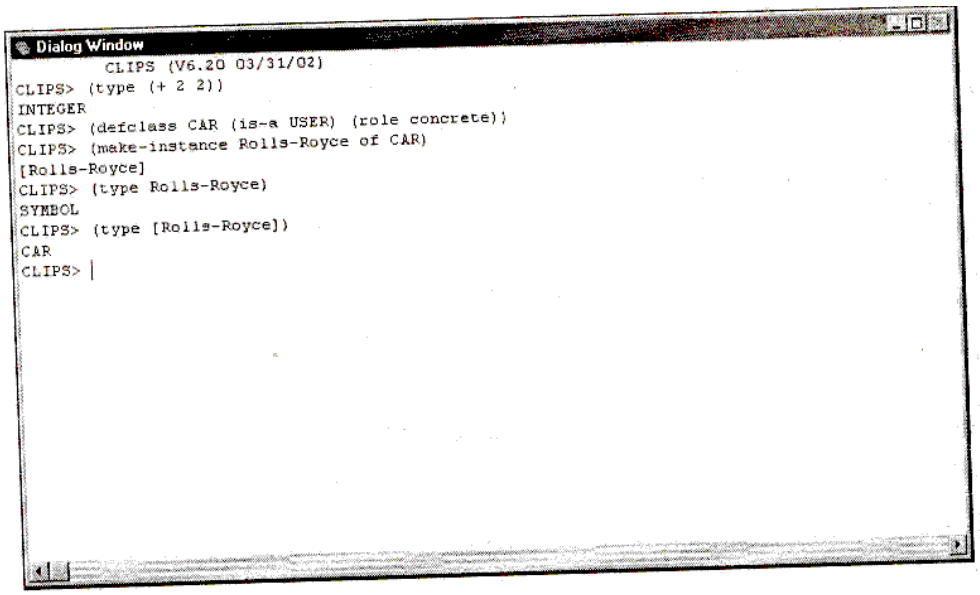

Рис. 15.22. Использование функции type

Функция next-methodp осуществляет проверку наличия скрытых методов родовой функции и возвращает значение true, если такой метод существует. В противном случае функция возвращает значение false. Вызов этой функции возможен только из метода родовой функции.

Определение 15.66, Функция next-methodp

(next-methodp)

Если функция next-methodp дала положительный результат, то обнаруженный скрытый метод родовой функции можно вызвать при помощи функции call-next-method. Скрытому методу передается тот же набор аргументов, что и вызывающему его методу родовой функции. Метод может продолжать свою работу и после вызова функции call-next-method. Кроме того, возможен множественный вызов скрытых методов родовой функции. Для этого нужно использовать функцию call-next-method необходимое число раз. Результатом работы данной функции служит результат вызванного скрытого метода или значение FALSE в случае ошибки (см. рис. 15.23).

Пример 15.26. Использование функции call-next-method

```
(defmethod describe   ((?a  INTEGER))
              (if   (next-methodp)   then
                   (bind  ?extension   (str-cat   "   "   (call-next-method)))
              else
                   (bind  ?extension  ""))
                   (str-cat   "INTEGER"   ?extension))
(describe  3)
(defmethod describe   ((?a NUMBER))    "NUMBER")
(describe 3)
(describe  3.0)
```

Функция call-next-handler подобна функции call-next-method и предназначена для вызова скрытых обработчиков сообщений объектов, определенных пользователем классов. Определение функции call-next-handler приведено ниже.

Определение 15.67. Функция call-next-handler

(call-next-handler)

Для изменения набора аргументов при вызове скрытого метода родовой функции служит функция override-next-method. В остальном она подобна предыдущей.

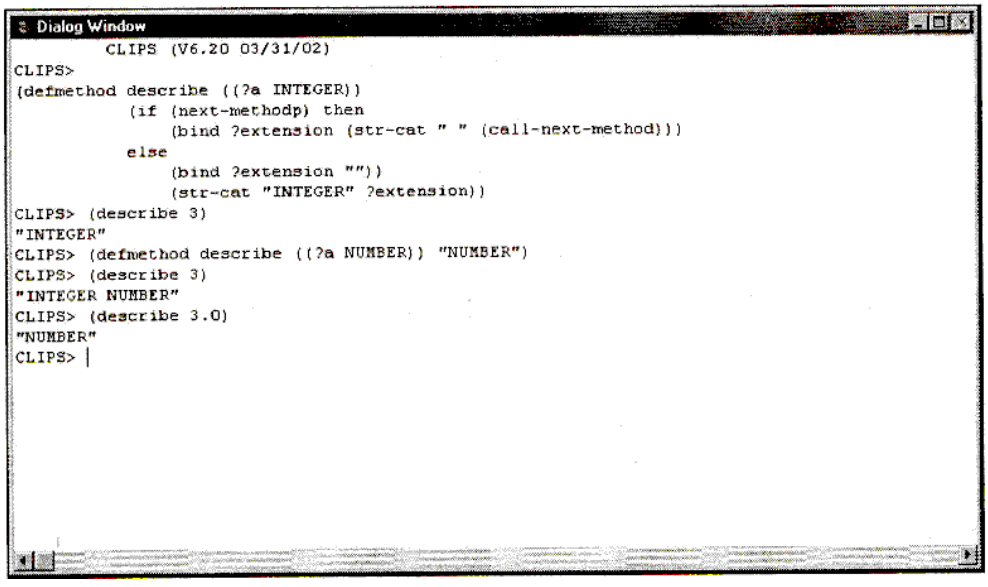

Рис. 15.23. Использование функции call-next-method

Определение 15.68. Функция override-next-method

(override-next-method <выражение>*)

Функция call-specific-method позволяет вызывать конкретный метод заданной родовой функции, без использования процесса родового связывания (рис. 15.24). Параметры, передаваемые вызываемому методу, должны быть применимы к ограничениям на его набор аргументов.

Определение 15.69. Функция call-specific-method

(call-specific-method <имя-родовой функции> <индекс-метода>
<выражение>*)

Пример 15.27. Использование функции call-specific-method

```
(clear)
(defmethod + ( (?a INTEGER) (?b INTEGER))
             (* (- ?a ?b) (- ?b ?a)))
(list-defmethods +)
(preview-generic +12)
(watch methods)
(+  1  2)
(call-specific-method +112)
(unwatch methods)
```

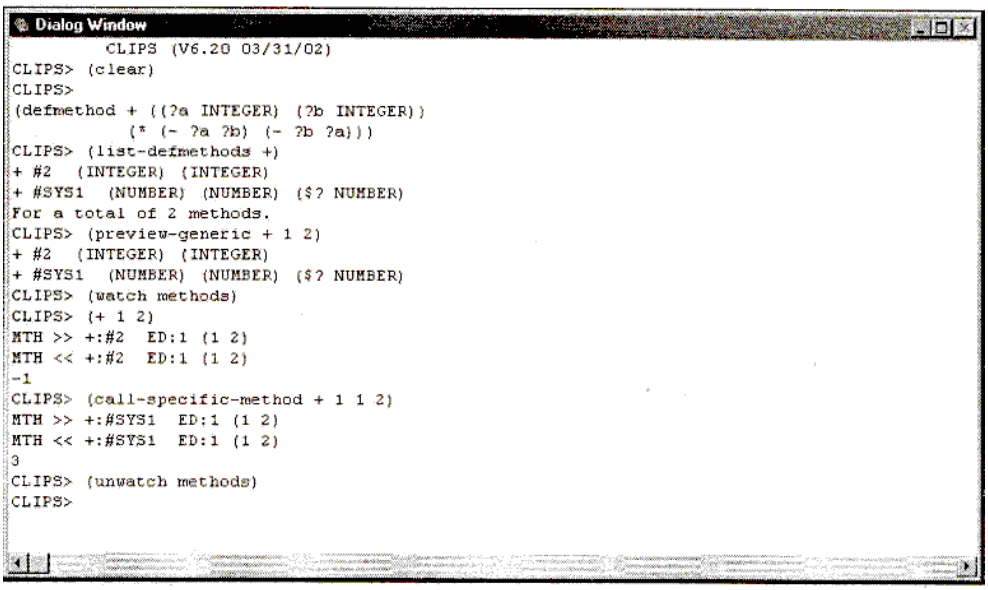

Рис. 15.24. Использование функции call-specific-method

Функция get-method-restrictions (рис. 15.25) возвращает составную величину, содержащую информацию об ограничениях для данного метода, используя следующий формат:

- минимальное число аргументов;
- максимальное число аргументов (может равняться —1 для групповых символов);
- количество ограничений;
- индекс составной величины, содержащей информацию о первом ограничении;
- индексы составных величин, содержащих информацию о втором, третьем ограничениях и т. д.;
- индекс составной величины, содержащей информацию о последнем ограничении;
- первое ограничение является ограничением запроса (TRUE или FALSE);
- число ограничений классов;
- первое, второе ограничения класса и т. д.;
- последнее ограничение класса;

- последнее ограничение является ограничением запроса (TRUE или FALSE);
- число ограничений классов;
- первое, второе ограничения класса и т. д.;
- последнее ограничение класса.

Определение 15.70. Функция get-method-restrictions

(get-method-restrictions <имя-родовой функции> <индекс-метода>
<выражение>*)

Пример 15.28. Использование функции get-method-restrictions

```
(clear)
(defmethod foo 50
          ((?a INTEGER SYMBOL)
          (?b (=11)) $?c))
(get-method-restrictions foo 50)
```

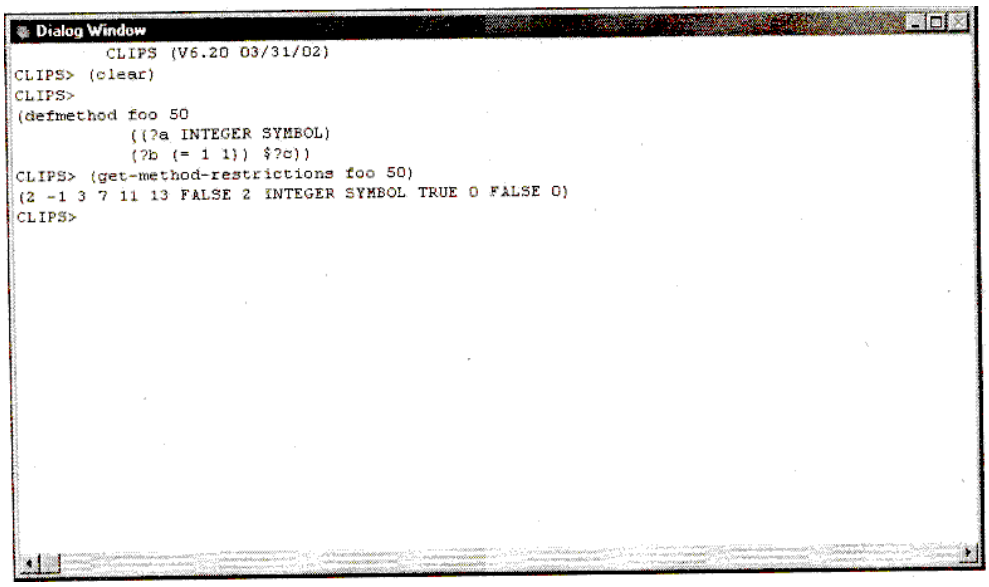

Рис. 15.25. Использование функции get-method-restrictions

15.8. Объектно-ориентированные функции

CLIPS предоставляет богатый набор функций, предназначенных для работы с классами, объектами, слотами, обработчиками сообщений. Описание этих функций приведено в табл. 15.14.

Таблица 15.14. *Объектно-ориентированные функции*

Функция	**Описание**
get-class-list	Получение списка существующих классов
defclass-module	Поиск модуля, в котором определен класс
class-existp	Определяет, существует ли заданный класс
superclassp	Определяет, является ли первый класс суперклассом второго
subclassp	Определяет, является ли первый класс подклассом второго

slot-existp	Проверяет существование слота
slot-writablep	Проверяет, доступен ли слот записи
slot-initablep	Проверяет, доступен ли слот инициализации
slot-publicp	Проверяет, является ли слот видимым
slot-direct-accessp	Проверяет, возможно ли прямое обращение к слоту
message-handler-existp	Проверяет существование обработчика сообщения
class-abstractp	Проверяет, является ли класс абстрактным
class-reactivep	Проверяет, является ли класс активным
class-superclasses	Получение списка суперклассов
class-subclasses	Получение списка подклассов
class-slots	Получение списка слотов класса
get-defmessage-handler-list	Получение списка обработчиков сообщений класса
slot-facets	Получение списка значений граней слота
slot-sources	Получение списка источников определения слота
slot-types	Получение типа слота
slot-cardinality	Получение мощности составного слота
slot-allowed-values	Получение списка допустимых значений слота
slot-range	Получение допустимого диапазона значений слота
slot-default-value	Получение значения по умолчанию для слота
next-handlerp	Проверка существования скрытых обработчиков
call-next-handler	Вызов скрытого обработчика
override-next-handier	Вызов скрытого обработчика с измененными аргументами
get-definstance-list	Получение списка классов, созданных с помощью конструктора definstance

definstances-module	Получения модуля, в котором определен конструктор definstance
init-slots	Инициализация слотов
unmake-instance	Удаление объекта
delete-instance	Удаление объекта из обработчика сообщения
class	Определение класса заданного объекта
instance-name	Определение имени объекта
instance-address	Определение адреса объекта
symbol-to-instance-name	Преобразование значение типа symbol в instance-name
instance-name-to-symbol	Преобразование значение типа instance-name в symbol
instancep	Проверка существования объекта с заданным адресом или именем
instance-addressp	Проверка существования объекта с заданным адресом
instance-namep	Проверка существования объекта с заданным именем
instance-existp	Проверка существования объекта
dynamic-put	Чтение значения слота
dynamic-get	Запись значения слота
slot-replase$	Замена полей составных слотов
slot-insert$	Добавления полей составных слотов
slot-delete$	Удаление полей составных слотов

Функция get-defclass-list возвращает составную величину, содержащую имена всех определенных в системе классов, видимых в заданном модуле. Если параметр <имя-модуля> не задан, функция выведет список классов, определенных в текущем модуле. Вместо имени модуля допустимо использовать символ *. В этом случае возвращается список всех классов

Определение 15.71. Функция get-defclass-list

(get-defclass-list [<имя-модуля>])

Для определения модуля, в котором указан заданный класс, служит функция defclass-module.

Определение 15.72. Функция defclass-module

(defclass-module <имя-класса>)

CLIPS предоставляет несколько предикатных функций, предназначенных для анализа свойств заданного класса. Синтаксис этих классов приведен ниже.

Определение 15.73. Предикатные функции для анализа свойств класса

(class-existp <имя-класса>)
(class-abstractp <имя-класса>)
(class-reactivep <имя-класса>)
(superclassp <имя-класса1> <имя-класса2>)
(subclassp <имя-класса1> <имя-класса2>)

Функция class-existp возвращает значение TRUE, если заданный класс определен в системе, и значение FALSE, если это не так. Функции class-abstractp и class-reactivep предназначены для проверки, является ли заданный класс абстрактным и участвует ли он в процессе сопоставления образцов. В случае положительного результата проверки функции возвращают значение TRUE, иначе — FALSE. Предикатная функция superclassp проверяет, является ли класс, заданный первым параметром, суперклассом класса, заданного вторым параметром, т. е. первый класс унаследован от второго. Аналогично функция subclass проверяет, является ли класс, заданный первым параметром, подклассом класса, заданного вторым параметром, т. е. первый класс является предком второго.

Для проверки существования заданного слота у некоторого класса предназначена функция slot-existp. Она возвращает значение TRUE в случае положительного результата проверки и значение FALSE, если это не так. Если при вызове функции указан необязательный параметр inherit, слот может быть унаследован от какого-нибудь суперкласса. В противном случае он должен быть определен непосредственно в проверяемом классе.

Определение 15.74. Функция slot-existp

(slot-existp <имя-класса> <имя-слота> [inherit])

Для проверки некоторых свойств отдельных слотов CLISP предоставляет несколько функций, синтаксис которых приведен ниже.

Определение 15.75. Функции для анализа свойств слотов

(slot-writablep <имя-класса> <имя-слота>)
(slot-initablep <имя-класса> <имя-слота>)
(slot-publicp <имя-класса> <имя-слота>)
(slot-direct-accessp <имя-класса> <имя-слота>)

В случае положительного результата проверки функции возвращают значение TRUE иначе — FALSE. В случае если указанный класс или слот не существует, перечисленные функции генерируют сообщение об ошибке. Функция slot-writablep проверяет возможность записи в слот. Функция slot-initablep предназначена для проверки возможности инициализации слота. Для выяснения, передается ли слот по наследованию (грань visibility имеет значение public), предназначена функция siot-publicp. Функция slot-direct-accessp служит для проверки, возможен ли прямой доступ к указанному слоту.

Для определения, существует ли указанный обработчик у заданного класса, предназначена функция message-handler-existp.

Определение 15.76. Функция message-handler-existp

```
(message-handler-existp <имя-класса>
                        <имя-обработчика-сообщения>
                        [<тип-обработчика-сообщения>])
<тип-обработчика-сообщения>  ::= around | before | primary | after
```

Эта функция возвращает значение TRUE, если указанный обработчик сообщений существует (обработчик должен быть определен непосредственно у заданного класса, а не получен при помощи наследования). Иначе возвращает значение FALSE. В качестве необязательного третьего параметра может быть задан тип обработчика. Если этого не сделано, по умолчанию анализируются только обработчики сообщений primary.

Пара функций class-superclasses и class-subclasses предназначена для получения списка всех суперклассов и подклассов указанного класса.

Определение 15.77. Функции class-superclasses и class-subclasses

```
(class-superclasses <имя-класса> [inherit])
(class-subclasses  <имя-класса> [inherit])
```

При использовании необязательного флага inherit функция class-superclasses генерирует список всех суперклассов. В противном случае возвращается список только прямых суперклассов (классов, от которых заданный класс унаследован непосредственно). Если в функции class-subclasses использован параметр inherit, то функция отображает всех наследников указанного класса, иначе только прямых наследников.

Варианты использования функций class-superclasses и class-subclasses представлены в примере 15.29 и на рис. 15.26.

Пример 15.29. Использование функций class-superclasses и class-subclasses

```
(class-superclasses   INTEGER)
(class-superclasses   INTEGER  inherit)
(class-subclasses  PRIMITIVE)
(class-subclasses   PRIMITIVE  inherit)
```

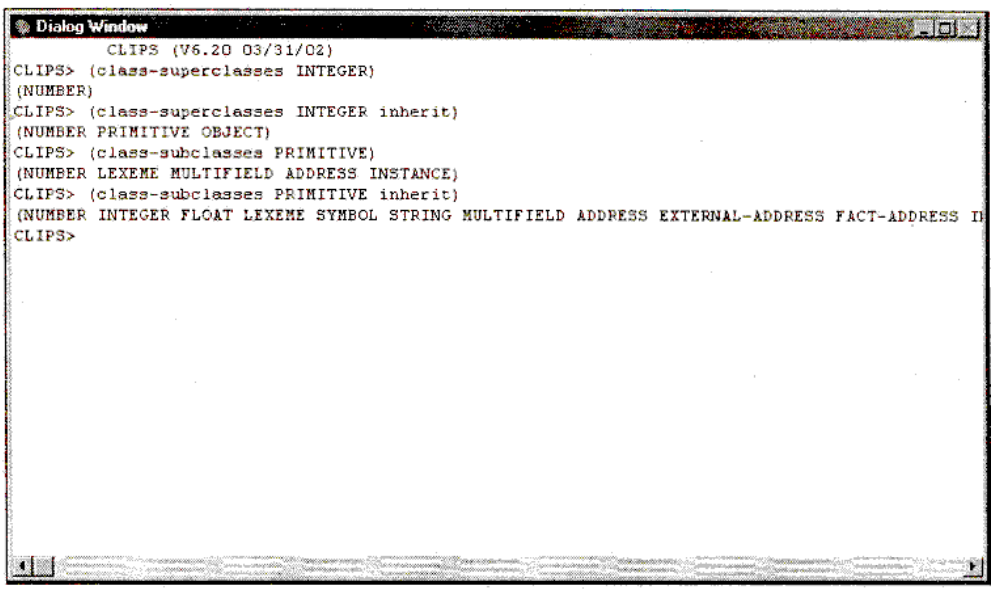

Рис. 15.26. Использование функций class-superclasses и class-subclasses

Функция class-slots возвращает составную величину, содержащую список слотов заданного класса. Если при вызове функции использовался необязательный параметр inherit, в список также помещаются слоты, унаследованные от суперклассов. В противном случае только слоты, напрямую определенные в заданном слоте. При возникновении ошибки функция возвращает составную величину нулевой длины.

Определение 15.78. Функция class-slots

(class-slots <имя-класса> [inherit])

Для получения списка обработчиков сообщений класса служит функция get-defmessage-handler-list. Синтаксис и назначение параметров этой функции во многом идентичны функции class-slots.

Определение 15.79. Функция get-defmessage-handler-list

(get-defmessage-handler-list <имя-класса> [inherit])

Ниже приведен пример использования функций class-slots и get-defmessage-handler-list (см. также рис. 15.27).

Пример 15.30. Использование функций class-slots и get-defmessage-handler-list

```
(defclass A (is-a USER) (slot x))
(defmessage-handler A foo ( ) )
(defclass B (is-a A) (slot y))
(class-slots B) (class-slots B inherit)
(get-defmessage-handler-list A)
(get-defmessage-handler-list A inherit)
```

Функция slot-facets возвращает список значений граней заданного слота (слот может быть как наследуемым, так и явно определенным) — рис. 15.28. Составная величина с нулевой длиной возвращается, если произошла ошибка. В табл. 15.15 приведено описание возвращаемых полей.

Таблица 15.15. *Поля, возвращаемые функцией* slot-facets

Поле	**Значение**	**Описание**
Тип поля	SGL/MLT	Простое или составное
Значение по умолчанию	STC/DYN/NIL	Статическое, динамическое или отсутствует
Распространение при наследовании	INH/NIL	Наследуется или нет
Доступ	RW/R/INT	Чтение-запись, только чтение, инициализация
Тип хранения	LCL/SHR	В экземпляре объекта или в классе
Активность при сопоставлении	RCT/NIL	Активный или нет
Источник свойств унаследованного слота	CMP/EXC	Составной или обычный
Видимость слота	PUB/PRV	Открытый или закрытый
Автоматическое создание акцессоров	R/W/RW/NIL	Только чтение, только запись, Чтение- запись или отсутствуют
Перегруженный обработчик	<имя сообщения>	Имя сообщения, посылаемого для переопределения слота функцией make-instance и др.

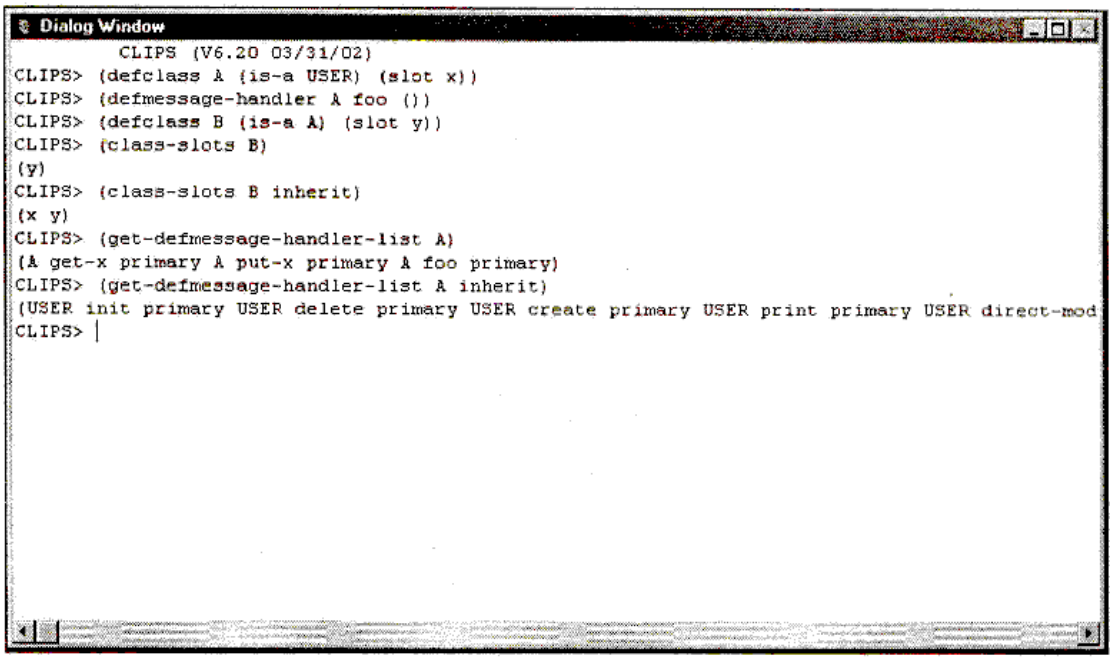

Рис. 15.27. Использование функций class-slots и get-defmessage-handler-list

Определение 15.80. Функция slot-facets

(slot-facets <имя-класса> <имя-слота>)

Пример 15.31. Использование функции slot-facets

```
(clear)
(defclass A (is-a USER)
            (slot x (access read-only)))
(defclass B (is-a A)
            (multislot y))
  (slot-facets B x)
  (slot-facets B y)
```

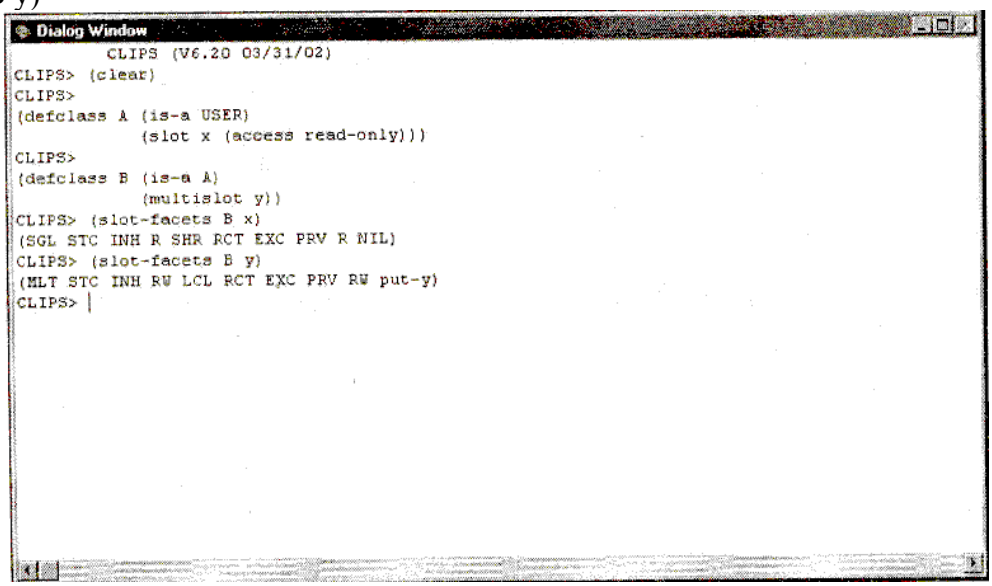

Рис. 15.28. Использование функции slot-facets

Следующая группа функций предназначена для определения значений отдельных свойств и граней заданных слотов.

Функция slot-sources возвращает список названий классов, которые использовались при наследовании граней заданного слота (пример 15.32 и рис. 15.29). Если грани слота эксклюзивно определял один класс, то список будет состоять из имени одного класса. Составная величина нулевой длины возвращается в случае возникновения ошибки.

Определение 15.81, Функция slot-sources

(slot-sources <имя-класса> <имя-слота>)

Пример 15.32. Использование функции slot-sources

```
(clear)
(defclass A (is-a USER)
        (slot x (access read-only))) (defclass B (is-a A)
        (slot x (source composite)
                (default 100))) (defclass C (is-a B))
                 (slot-sources A x) (slot-sources B x)
                 (slot-sources C x)
```

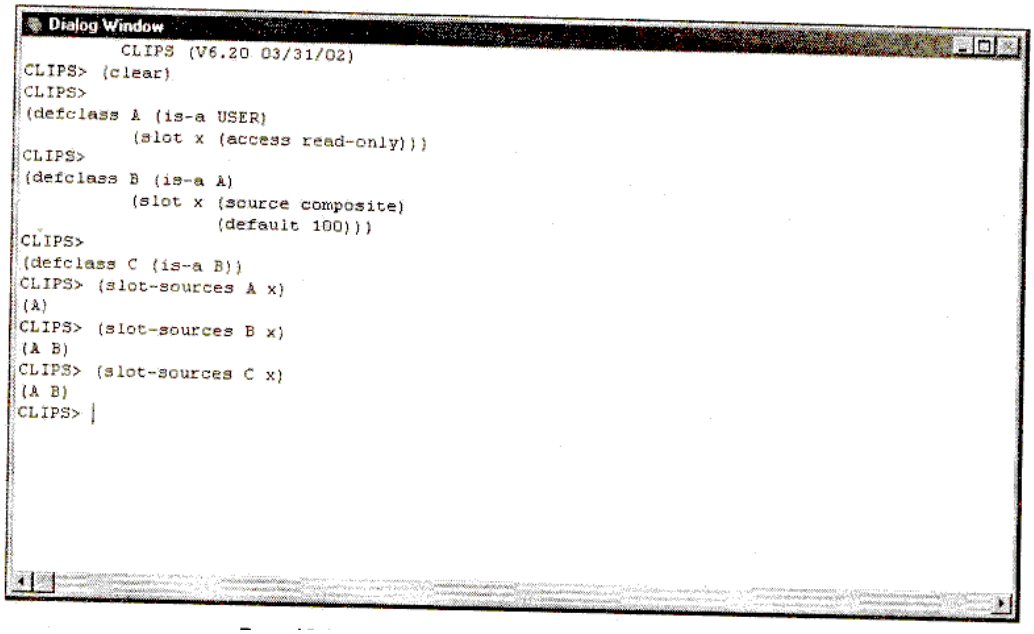

Рис. 15.29. Использование функции slot-sources

Функция slot-types возвращает список названий примитивных типов слота.

Определение 15.82. Функция slot-types

(slot-types <имя-класса> <имя-слота>)

Функция slot-cardinality возвращает составное поле, содержащее минимальное и максимальное число элементов, допустимое для заданного составного слота. Максимальная емкость слота — положительная бесконечность, обозначается символом $+\infty$. Ниже приведены примеры использования функций slot-types и slot-cardinality (см. также рис. 15.30).

Определение 15.83. Функция slot-cardinality

(slot-cardinality <имя-кпасса> <имя-слота>)

Пример 15.33. Использование функций slot-types и slot-cardinality

```
(clear)
(defclass A (is-a USER)
        (slot x (type INTEGER LEXEME))
        (multislot y (cardinality ?VARIABLE 5))
        (multislot z (cardinality 3 ?VARIABLE)))
(slot-types A x)
(slot-cardinality A x)
(slot-cardinality A y)
(slot-cardinality A z)
```

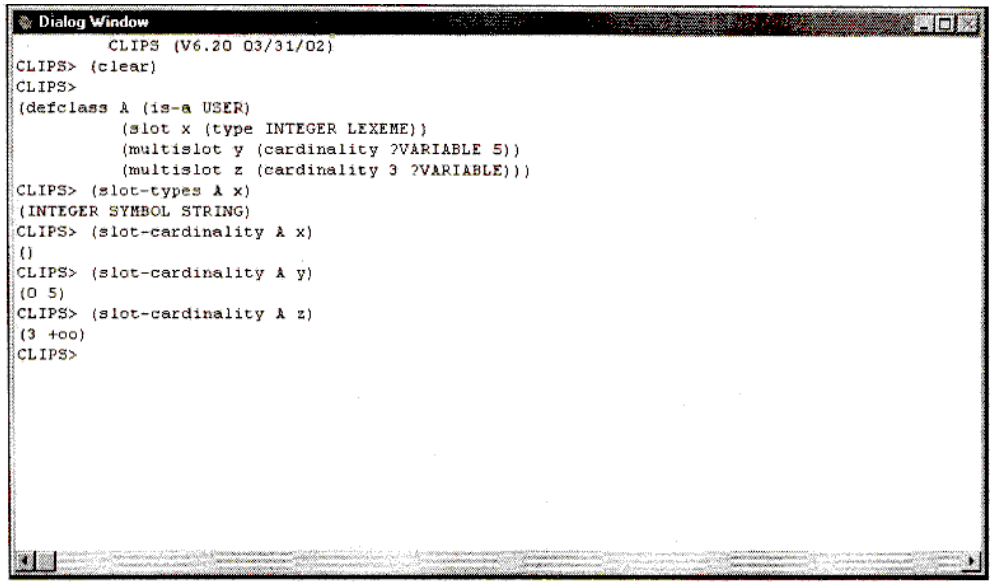

Рис. 15.30. Использование функций slot-types и slot-cardinality

Функция slot-allowed-values возвращает допустимые значения для слота, заданные с помощью соответствующей грани. Если ограничения на значения слота отсутствуют, то функция возвращает значение FALSE. Составная величина с нулевой длиной возвращается в том случае, если произошла ошибка.

Определение 15.84. Функция slot-allowed-values

(slot-allowed-values <имя-класса> <имя-слота>)

Для определения допустимого диапазона значений слота предназначена функция slot-range. Минимальное значение обозначается символом - ∞, максимальное значение — символом + ∞. Значение FALSE возвращается для слота, который не поддерживает числовые значения.

Определение 15.85. Функция slot-range

(slot-range <имя-класса> <имя-слота>)

Ниже приведены примеры использования функций slot-allowed-values и slot-range (см. также рис. 15.31).

Пример 15.34. Использование функций slot-allowed-values и slot-range

```
(clear)
(defclass A   (is-a USER)
              (slot x)
              (slot  y   (allowed-integers  2  3)
                         (allowed-symbols  foo)))
(defclass  B   (is-a  USER)
              (slot  x)
              (slot y (type SYMBOL))
              (slot z (range 3 10)))
 (slot-allowed-values A x)
 (slot-allowed-values A y)
 (slot-range B x)
 (slot-range B y)
 (slot-range B z)
```

Функция slot-default-value возвращает значение, по умолчанию связанное с заданным слотом. Если слот имеет динамическое значение по умолчанию, результат вычисляется в момент вызова функции. Значение FALSE возвращается в том случае, если произошла ошибка.

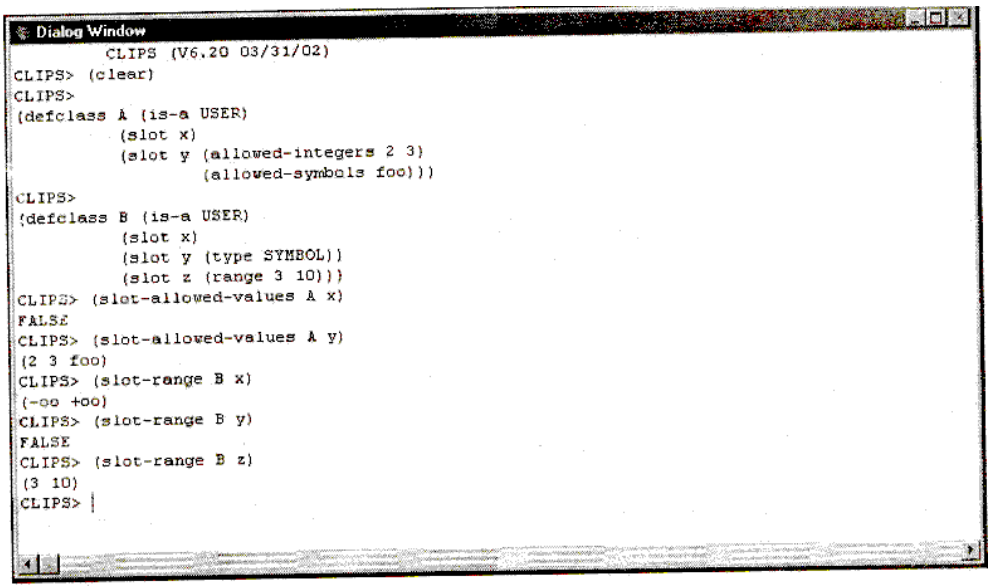

Рис. 15.31. Использование функций slot-allowed-values и slot-range

Определение 15.R6. Функция slot-default-value

(slot-default-value <имя-класса> <имя-слота>)

Функция next-handlerp осуществляет проверку наличия скрытых обработчиков сообщений и возвращает значение TRUE, если такой обработчик существует, в противном случае функция возвращает значение FALSE.

Определение 15.87. Функция next-handlerp

(next-handlerp)

Если функция next-handlerp дала положительный результат, то обнаруженный скрытый обработчик можно вызвать при помощи функции call-next-handler. Скрытому обработчику передаются те же аргументы, что и вызывающему обработчику. Обработчик может продолжать свою работу после вызова функции call-next-handler. Кроме того, возможен множественный вызов функции call-next-handler, в этом случае скрытый обработчик будет вызываться нужное количество раз. Результатом работы данной функции служит результат вызванного скрытого обработчика или значение FALSE в случае ошибки.

Определение 15.88. Функция call-next-handler

(call-next-handler)

Для изменения набора аргументов при вызове скрытого обработчика служит функция override-next-handler. В остальном эта функция подобна предыдущей.

Определение 15.89. Функция override-next-handler

(override-next-handler <выражение>*)

Ниже приведены примеры использования функций call-next-handler и override-next-handler (см. также рис. 15.32).

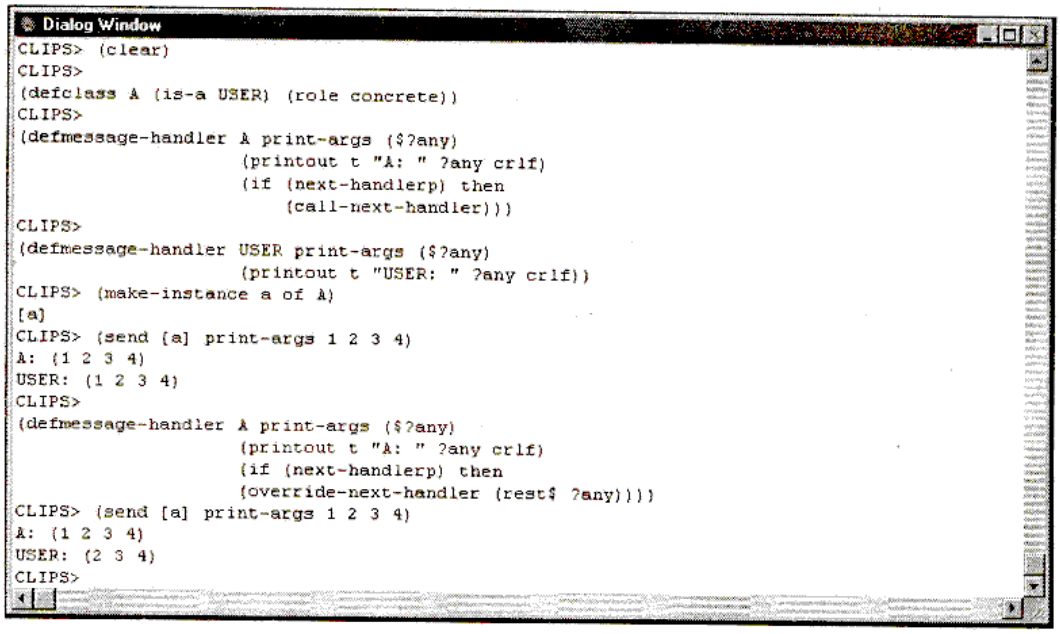

Рис. 15.32. Использование функций call-next-handler и override-next-handler

Пример 15.35. Использование функций call-next-handler и override-next-handler

```
(clear)
(defclass A (is-a USER) (role concrete))
(defmessage-handler A print-args ($?any)
                        (printout t "A: " ?any crlf)
                (if (next-handlerp) then
                    (call-next-handler)))
(defmessage-handler USER print-args ($?any)
                        (printout t "USER: " ?any crlf))
(make-instance a of A)
(send [a] print-args 1234)
(defmessage-handler A print-args ($?any)
                        (printout t "A: " ?any crlf)
                        (if (next-handlerp) then
                        (override-next-handier (rest$ ?any))))
(send [a] print-args 1234)
```

Для получения всех конструкторов definstances, определенных в заданном модуле, предназначена функция get-definstances-list. Функция def instances-module служит для определения модуля, в котором используется заданный конструктор definstances. Синтаксис этих функций приведен ниже.

Определение 15.90. Функции get-def instances-list и def instances-module

(get-definstances-list [<имя-модуля>])

(definstances-module <имя-конструктора>)

Функция init-slots предназначена для выполнения процедуры инициализации слотов. Она доступна только в обработчиках сообщений класса. Обычно функция автоматически вызывается командами make-instance и initzializate-instance.

Определение 15.91. Функция init-slots

(init-slots)

Функции unmake-instanceи delete-instance удаляют некоторый объект, а функция delete-instance предназначена для использования внутри обработчиков сообщений и удаляет активный объект,

unmake-instance может использоваться вне определения обработчиков сообщений, поэтому требует указания объекта, который необходимо удалить.

Определение 15.92. Функции unmake-instance и delete-instance

(unmake-instance <имя-или-адрес-объекта>+)
(delete-instance)

Для определения класса, имени и адреса объекта служат функции class, instance-name и instance-address соответственно.

Определение 15.93. Функции class, instance-name и instance-address

(class <объект>)
(instance-name <объект>)
(instance-address [<имя-модуля> \ *] <объект>)

Пара функций symbol-to-instance-name и instance-name-to-symbol применяется для преобразования значения типа symbol в instance-name и обратно. При возникновении ошибки обе функции возвращают FALSE, в случае удачи — TRUE.

Определение 15.94. Функции symbol-to-instance-name и instance-name-to-symbol

(symbol-to-instance-name <значение-типа-symbol>)
(instance-name-to-symbol <значение-типа-instance-name>)

Предикатные функции instancep, instance-namep и instance-addressp служат для определения, является ли их аргумент объектом, именем объекта и адресом объекта соответственно.

Определение 15.95. Функции instancep, instance-namep и instance-addressp

(instancep <выражение>)
(instance-namep <выражение>)
(instance-addressp <выражение>)

Функция instance-existp определяет, присутствует ли в настоящий момент в системе объект, заданный ее аргументом.

Определение 15.96. Функция instance-existp

(instance-existp <имя-или-адрес-объекта>)

Функции dynamic-get и dynamic-put предназначены для динамического получения и изменения содержимого слотов активного объекта (т. е. эти функции доступны только из обработчиков сообщений). Отличие этих функций от стандартного способа получения и изменения слотов объекта заключается в том, что связь с конкретным слотом осуществляется при каждом вызове функции, а не при создании соответствующего обработчика сообщения.

Определение 15.97. Функции dynamic-get и dynamic-put

(dynamic-get <имя-слота>)

(dynamic-put <имя-слота> <выражение>)

Для работы со значениями составных слотов CLIPS предоставляет функции slot- replace$, slot-insert$ и slot-delete$, действия которых аналогичны действиям соответствующих функций для работы с составными значениями. В отличие от стандартных, функции для работы со значениями составных слотов имеют два варианта синтаксиса. Первый предназначен для применения вне объекта, а второй для использования внутри обработчиков сообщений.

Определение 15.98. Внешнее использование функций slot-replace$, slot-insert$ и slot-delete$

```
(slot-replace$   <имя-объекта> <имя-слота>
                 <начало-блока> <конец-блока> <выражение>+)
(slot-insert$    <имя-объекта> <имя-слота>
                 <начало-блока> <выражение>+)
(slot-delete$    <имя-объекта> <имя-слота>
                 <начало-блока> <конец-блока>)
```

Определение 15.99. Внутреннее использование функций slot-replace$, slot-insert$ и slot-delete$

```
(slot-replace$   <имя-слота>
                 <начало-блока> <конец-блока> <выражение>+)
(slot-insert$    <имя-слота>
                 <начало-блока> <выражение>+)
(slot-delete$    <имя-слота>
                 <начало-блока> <конец-блока>)
```

15.9. Вспомогательные функции

В заключение данной главы рассмотрим несколько вспомогательных функций, список которых приведен в табл. 15.16.

Таблица 15.16. *Вспомогательные функции*

Функция	**Описание**
gensym	Генератор идентификатора
gensym*	Расширенный генератор идентификатора
setgen	Установка начального значения для генератора идентификаторов
random	Генератор случайного числа
seed	Сброс генератора случайных чисел

Функция gensym предназначена для генерации уникального идентификатора. Эта функция используется прежде всего для маркировки отдельных конструкций CLIPS, если пользователю не требуется осмысленное наименование идентификатора. Многократные запросы к функции gensym гарантированно возвращают различные идентификаторы в виде genx, где x — некоторое положительное число. Первый вызов gensym возвращает значение geni, все последующие вызовы увеличивают номер на единицу. Функция gensym не перезагружается после команды clear.

Определение 15.100. Функция gensym

```
(gensym)
```

Функция setgen предоставляет пользователю возможность устанавливать начальный номер, используемый функциями gensym и gensym*. Аргумент функции setgen должен быть положительным целым числом.

Определение 15.101. Функция setgen

```
(setgen <целочисленное-выражение>}
```

Отличие функции gensym* от gensym заключается в том, что она создает уникальный идентификатор, не использующийся в данный момент в системе.

Определение 15.102. Функция gensym*

(gensym*)

Ниже приведены примеры использования функций gensym, gensym* и setgen (см. также рис. 15.33).

Пример 15.36. Использование функций gensym, gensym* и setgen

```
(setgen 1)
(assert (genl gen2 gen3) )
(gensym)
(gensym*)
```

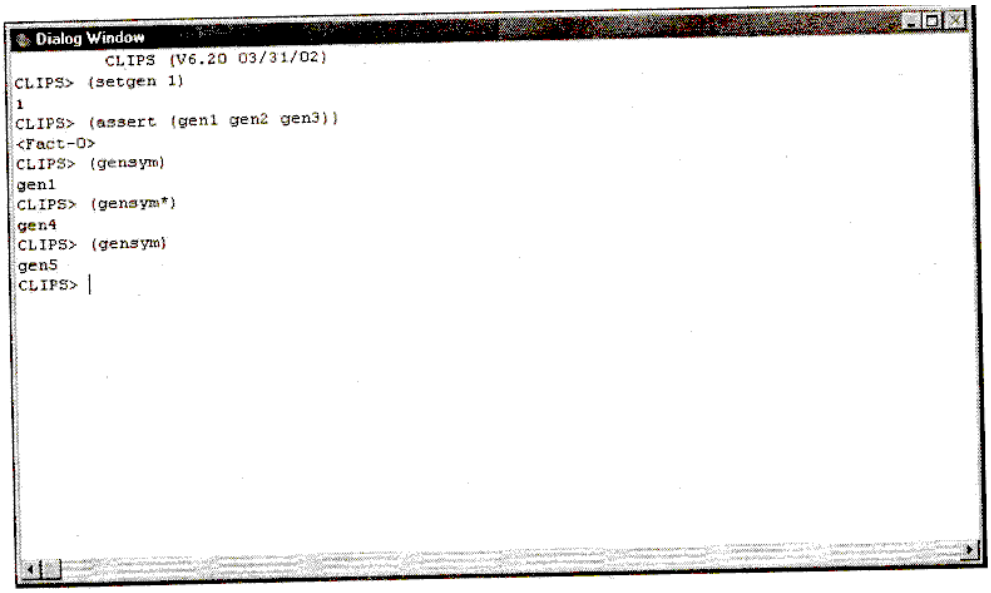

Рис. 15.33. Использование функций gensym, gensym* и setgen

Функция random возвращает случайное целое число. Если при вызове функции задан диапазон, полученное в результате случайное число будет лежать между указанными целочисленными границами.

Определение 15.103. Функция random

(random [<начало-диапазона> <конец-диапазона>])

Для сброса генератора случайных чисел служит функция seed.

ГЛАВА 16. Основные команды CLIPS

Как уже упоминалось ранее, командами в CLIPS называются функции, которые не возвращают пользователю численного или символьного результата, а выполняют те или иные действия. Однако это правило не всегда соблюдается. Некоторые из перечисленных в данной главе команд выводят в диалоговое окно системы результаты своей работы. Более точно термин команда можно определить следующим образом: *команда* — это функция, которая предназначена для использования в интерактивном режиме работы пользователя в диалоговом окне CLIPS. Это не означает, что команды нельзя использовать внутри определения правил или, например, функций, но, как правило, такое применение команд не вызывает никаких действий системы и лишено смысла.

CLIPS предоставляет весьма богатый набор команд, охватывающий все потребности пользователя системы, среди которых, например: команды управления интерактивной средой, команды, помогающие при отладке программ и профилировании, команды для работы с конструкторами deftemplate, deffacts, defrule, defglobal, deffunction, defgeneric, defmethod, defmodule, команды управления памятью, текущим планом решения задачи и многое другое. Данная глава целиком посвящена описанию команд среды CLIPS.

16.1. Управление интерактивной средой

Последняя версия CLIPS содержит 19 команд, предназначенных для управления интерактивной средой. Краткое описание этих команд приведено в табл. 16.1.

Таблица 16.1. *Команды работы со средой*

Команда	**Описание**
load	Загрузка конструкторов из текстового файла
load*	Загрузка конструкторов из текстового файла без отображения процесса загрузки в диалоговом окне CLIPS
save	Сохранение созданных конструкторов, в текстовый файл
bload	Загрузка конструкторов из двоичного файла
bsave	Сохранение созданных конструкторов, в двоичный файл
clear	Очистка рабочей памяти системы
exit	Выход из CLIPS
reset	Сброс рабочей памяти системы
batch	Запуск командного файла
batch*	Запуск командного файла без отображения процесса выполнения в диалоговом окне CLIPS
options	Вывод в диалоговое окно информации обо всех текущих установках системы
system	Выполнение команды операционной системы
set-auto-float-dividend	Установка режима автоматического преобразования типа чисел при делении
get-auto-float-dividend	Проверка текущего состояния режима автоматического преобразования типов чисел при делении
set-dynamic-constraint-checking	Установка режима динамической проверки ограничений
get-dynamic-constraint-checking	Проверка текущего состояния режима динамической проверки ограничений
set-static-constraint-checking	Установка режима статической проверки ограничений
get-static-constraint-checking	Проверка текущего состояния режима статической проверки ограничений
apropos	Поиск любых элементов языка, введенных в рабочую память и содержащих заданную подстроку

Команды load и load* предназначены для загрузки конструкторов из текстового файла. Загружаемый текстовый файл можно создать в любом текстовом редакторе, самом CLIPS или с помощью команды save. Обе команды имеют схожий формат, приведенный ниже.

Определение 16.1. Команды load и load*

(load <имя-файла>) (load* <имя-файла>)

Команда load, в отличие от load*, выводит в диалоговое окно CLIPS информацию о процессе загрузки файла. Если включен режим отображения изменения наборов того или иного элемента, CLIPS выводит в диалоговое окно соответствующее сообщение при выполнении каждого конструктора. Если в процессе загрузки в текстовом файле встретится ошибочное определение конструктора, CLIPS выведет соответствующее сообщение об ошибке и продолжит чтение файла. При успешном выполнении обе команды возвращают значение TRUE, в противном случае — FALSE.

Команда save создает текстовый файл и записывает в него все определенные в системе на данный момент конструкторы.

(save <имя-файла>)

Важной особенностью команды save является то, что при сохранении конструкторов deffunction и defmethod в текстовом файле автоматически генерируются предварительные определения функций и методов родовых функций. Это необходимо для избежания циклических ссылок, возможных при рекурсивных вызовах. Для демонстрации такой ситуации выполните действия, представленные в примере 16.1.

Пример 16.1. Использование команды save

```
(clear)
(deffunction example-1        ( )
                (printout  t "Function example 1 without parameters" crlf)
             )
(deffunction example-2        (?a)
                (printout  t "Function example 2 with 1 parameter" crlf)
                (printout  t "Parameter = "?a crlf)
             )

(deffunction example-3 (?a ?b)
                (printout t "Function example 3 with 2 parameters" crlf)
                (printout t "Parameter 1 = "?a crlf)
                (printout t "Parameter 2 = "?b crlf)
   )
(save "example-22.CLP")
```

После выполнения описанных выше действий откройте полученный файл example.CLP (например, с помощью программы Notepad (Блокнот) операционной системы Windows). Полученный файл имеет содержание, представленное в примере 16.2.

Пример 16.2. Результат применения команды save

```
(deffunction MAIN::example-l ( ) )
(deffunction MAIN::example-2 (?p0))
(deffunction MAIN::example-3 (?p0 ?pl))

(deffunction MAIN::example-1
                ( )
                (printout t "Function example 1 without parameters" crlf))
(deffunction MAIN::example-2
                (?a)
                (printout t "Function example 2 with 1 parameter" crlf)
```

```
        (printout t "Parameter = " ?a crlf))
(deffunction MAIN::example-3
        (?a ?b)
        (printout t "Function example 3 with 2 parameters" crlf)
        (printout t "Parameter 1 = " ?a crlf)
        (printout t "Parameter 2 = " ?b crlf))
```

Как видно из приведенного примера, в начале файла сохраняются предварительные определения всех функций — объявления их названий и списка необходимых параметров, без определения последовательности действий, выполняемых функциями. И только после предварительного определения всех функций, содержащихся в данный момент в системе, в текстовый файл помещают полные определения функции.

В случае успешного выполнения команда save возвращает значение TRUE, в противном случае — FALSE.

Кроме команд save и load, позволяющих сохранять и загружать конструкторы из текстовых файлов, CLIPS предоставляет аналогичные функции bsave и bload, использующие бинарные файлы. Формат этих функций приведен ниже.

Определение 16.3. Команды bsave и bload

(bsave <имя-файла>) (bload <имя-файла>)

Замечание

Бинарные файлы обладают определенными преимуществами. Они загружаются гораздо быстрее текстовых. Кроме того, формат бинарных файлов не зависит от платформы, на которой применяется CLIPS, и поэтому файлы могут использоваться, например, для переноса информации из UNIX-версии системы CLIPS в Windows-версию. Однако бинарные файлы обладают также и рядом недостатков. Во-первых, они имеют гораздо больший размер, чем текстовые файлы. Во-вторых, создание бинарных файлов возможно только с помощью выполнения команды bsave непосредственно в среде CLIPS. В-третьих, бинарные файлы не сохраняют текстовое определение конструкторов, поэтому для конструкторов, загруженных с помощью бинарных файлов, невозможно использование команд, предназначенных для вывода определения конструктора в диалоговое окно (например, ppdefrule, ppdeftemplate и т. д.). Кроме того, в бинарных файлах не сохраняется информация об ограничениях, ассоциированная с конструкторами, если включен режим динамической проверки ограничений.

Обе команды возвращают значение TRUE, если в процессе выполнения не произошло ошибок, и значение FALSE — в противном случае.

Команда clear предназначена для очистки рабочей памяти системы. Она удаляет все определенные в системе на текущий момент конструкторы и ассоциированные с ними данные. Команда clear не возвращает никакого значения и имеет следующий формат:

Определение 16.4. Команда clear

(clear)

С помощью команды exit можно завершить сеанс работы пользователя с системой и закрыть CLIPS. Команда имеет следующий формат:

Определение 16.5. Команда exit

(exit [<целочисленное-выражение>])

Необязательный параметр позволяет передавать операционной системе код завершения работы приложения для последующего анализа. Команда exit может использоваться как в процессе диалога пользователя с системой, так и в командных файлах.

Команда reset предназначена для перезагрузки рабочей памяти системы. Она очищает текущий план решения задачи, удаляет все факты из списка фактов и объекты из списка объектов, устанавливает модуль MAIN текущим. Кроме того, она добавляет в систему предопределенный факт initial-fact, предопределенный объект initial-object и все факты, объекты и глобальные

переменные, определенные пользователем с помощью конструкторов deffacts, definstances И defglobals. Формат этой команды приведен ниже.

Определение 16.6. Команда reset

(reset)

Команды batch и batch* позволяют выполнять командные файлы. Командные файлы представляют собой обычные текстовые файлы, содержащие любые команды, функции и конструкторы, которые могли бы использоваться в диалоговом окне CLIPS. (Однако применение в командном файле конструкторов не рекомендуется. Вместо этого желательно использовать команду load внутри командного файла.) В случае успешного выполнения командного файла обе функции возвращают значение TRUE, в противном случае — FALSE. Формат обеих функций приведен ниже.

Определение 16.7. Команды batch и batch*

(batch <имя-файла>) (batch* <имя-файла>)

Отличие команды batch* заключается в том, она не выводит в диалоговое окно системы информацию о процессе выполнения командного файла. Если в командном файле использовалась команда exit, то после выполнения командного файла среда CLIPS будет закрыта.

Команда options выводит в диалоговое окно системы информацию о конфигурации системы CLIPS. Эта команда не имеет параметров. Пример ее использования приведен на рис. 16.1.

Определение 16.8. Команда options

(options)

Благодаря использованию команды system, CLIPS позволяет выполнять команды операционной среды. Эта команда имеет следующий формат:

Определение 16.9. Команда system

(system <команда-операционной-системы>*)

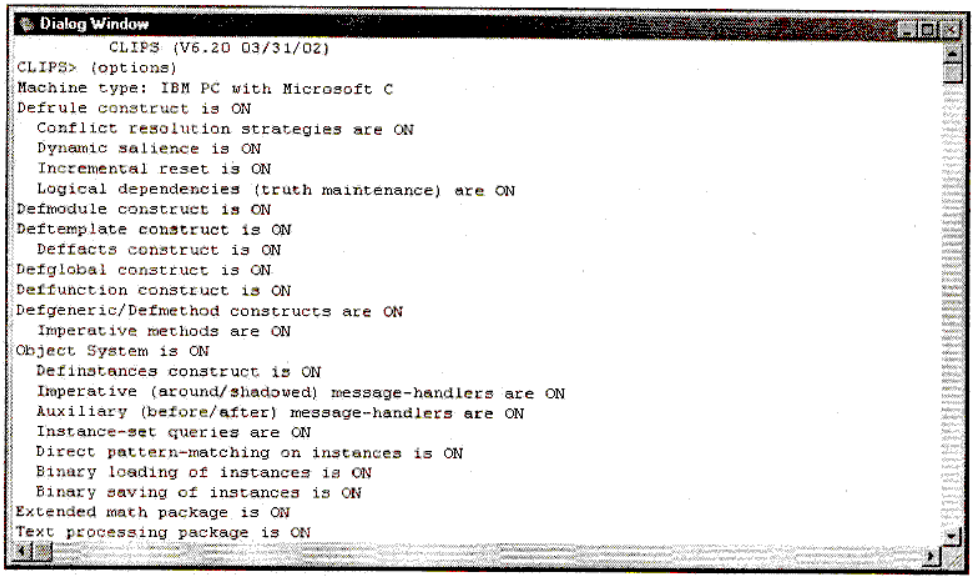

Рис. 16.1. Пример использования команды options

Команды set-auto-float-dividend и get-auto-float-dividend предназначены для установки и проверки режима *автоматического преобразования типа чисел при делении* (auto-float dividend behavior mode). Если данный режим включен, что является установкой по умолчанию, то CLIPS автоматически приводит тип делимого к вещественному типу. Команда set-auto-float-dividend

использует заданное логическое выражение для установки текущего значения режима автоматического преобразования типа чисел при делении и возвращает предыдущее установленное значение. Команда get-auto-float-dividend выводит в диалоговое окно системы текущее установленное значение режима автоматического преобразования типа чисел при делении (TRUE — включен, FALSE — выключен). Синтаксис этих команд приведен ниже.

Определение 16.10. Команды set-auto-f lost-dividend и get-auto-float-dividend

(set-auto-float-dividend <логическое-выражение>) (get-auto-float-dividend)

Команды set-dynamic-constraint-checking и get-dynamic-constraint-checking позволяют выполнить установку и проверку текущего состояния режима динамической проверки ограничений. По умолчанию данный режим отключен (значение false). Команда set-dynamic-constraint-checking устанавливает текущий режим динамической проверки ограничений, используя заданное логическое выражение, и возвращает предыдущее значение. Команда get-dynamic-constraint-checking отображает текущее значение режима динамической проверки ограничений.

Определение 16.11. Команды set-dynamic-constraint-checking и get-dynamic-constraint-checking

(set-dynamic-constraint-checking <логическое-выражение>)

(get-dynamic-constraint-checking)

Команды set-static-constraint-checking и get-static-constraint-checking предназначены для установки и проверки текущего состояния режима статической проверки ограничений. По умолчанию данный режим включен (значение TRUE). В остальном эти команды аналогичны командам set-dynamic-constraint-checking и get-dynamic-constraint-checking.

Определение 16.12. Команды set-static-constraint-checking и get-static-constraint-checking

(set-static-constraint-checking <логическое-выражение>)

(get-static-constraint-checking)

Команда apropos предназначена для поиска любых элементов языка, содержащих некоторую заданную подстроку.

Определение 16.13. Команда apropos

(apropos <строка>)

16.2. Работа с конструкторами *deftemplate*

Список команд, предназначенных для работы с конструкторами deftemplate, и их краткое описание приведены в табл. 16.2.

Таблица 16.2. *Команды работы с конструкторами deftemplate*

Команда	Описание
ppdeftemplate	Вывод определения конструктора в диалоговое окно CLIPS
list-deftemplates	Вывод в диалоговое окно системы списка всех определенных конструкторов deftemplate
undeftemplate	Удаление определенного конструктора deftemplate

С помощью команды ppdeftemplate пользователь может вывести определение конструктора deftemplate в диалоговое окно системы.

Определение 16.14. Команда ppdeftemplate

(ppdeftemplate <имя-конструктора>)

Команда list-deftemplates предназначена для отображения в диалоговом окне списка имен всех определенных в системе конструкторов deftemplate.

Определение 16.15. Команда list-deftemplates

(list-deftemplates [<имя-модуля>])

Если необязательный параметр <имя-модуля> не задан, то данная команда выводит список имен всех конструкторов deftemplate, определенных в текущем модуле. Если параметр содержит имя конкретного модуля, команда list-deftemplates выводит список конструкторов, определенных в заданном модуле. В качестве параметра допускается использование символа *. В этом случае команда выведет список имен всех конструкторов deftemplate, определенных во всех модулях системы.

Для удаления определенных пользователем конструкторов deftemplate предназначена команда undeftemplate.

Определение 16.16. Команда undeftemplate

(undeftemplate <имя-конструктора>)

В качестве параметра <имя-конструктора> возможно использование символа *. В этом случае команда попытается удалить все определенные пользователем конструкторы deftemplate (предопределенный системный конструктор deftemplate initial-fact удалить нельзя). Если выбранный конструктор используется, например, фактом или правилом, его удаление закончится неудачей.

Для иллюстрации использования перечисленных выше команд, предназначенных для работы с определенными пользователем конструкторами deftemplate, выполните следующие действия.

Пример 16.3. Работа с конструкторами deftemplate

```
(clear)
(deftemplate templ-1
             "Template - 1"
             (slot value))
(deftemplate templ-2
             "Template — 2"
             (slot value))
(deftemplate templ-3
             "Template — 3"
             (slot value))
(ppdeftemplate templ-2)
(list-deftemplates)
(undeftemplate templ-2)
(list-deftemplates)
(undeftemplate *)
(list-deftemplates)
```

Результат выполнения описанных выше команд приведен на рис. 16.2.

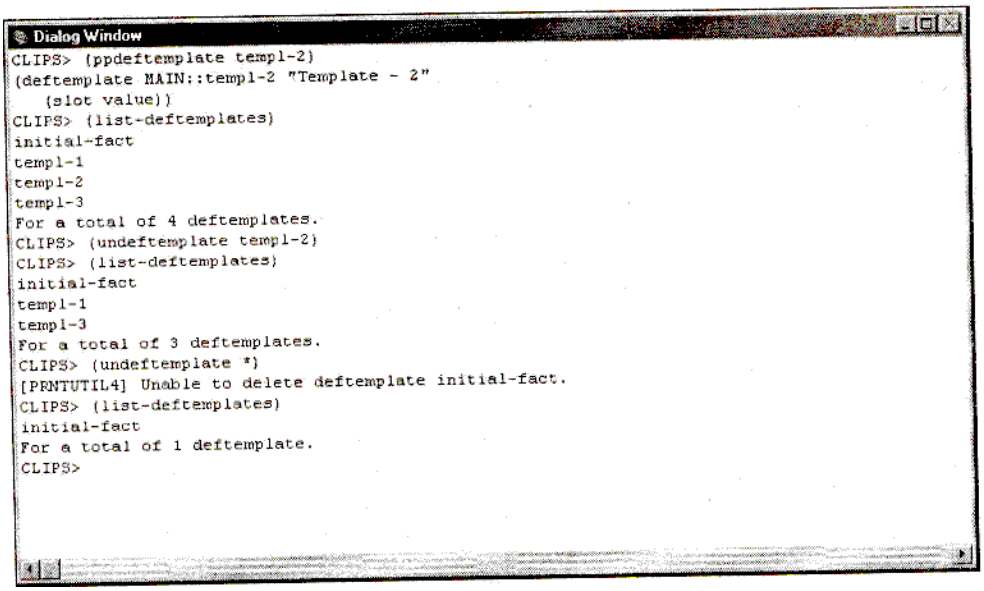

Рис. 16.2. Использование команд работы с конструкторами deftemplate

16.3. Работа с фактами

CLIPS предоставляет 5 команд, предназначенных для работы с фактами и списком фактов (табл. 16.3).

Таблица 16.3. *Команды работы с фактами*

Команда	**Описание**
facts	Загрузка конструкторов из текстового файла
load-facts	Загрузка списка фактов из текстового файла
save-facts	Сохранение текущего списка фактов в текстовый файл
set-fact-duplication	Установка режима дублирования фактов
get-fact-duplication	Проверка режима дублирования фактов

Команда facts предназначена для вывода в диалоговое окно системы текущего списка фактов.

Определение 16.17. Команда facts

```
(facts  [<имя-модуля>]
        [<минимальный-индекс-факта>
          [<максимальный-индекс-факта>
            [<максимальное-число-фактов>]]])
```

Если команда facts используется без параметров, то CLIPS выведет в диалоговое окно список фактов, видимых в текущем модуле. Если необязательный параметр <имя-модуля> задан, то данная команда выводит список имен всех фактов, видимых в указанном модуле. В качестве данного параметра допускается использование символа *. В этом случае команда выведет список имен всех фактов из всех модулей системы.

Параметры <минимальный-индекс-факта>, <максимальный-индекс-факта> и <максимальное-число-фактов> позволяют указать три целочисленных выражения, с помощью которых можно управлять объемом выводимой информации. Параметр <минимальный-индекс-факта> задает минимальный допустимый индекс факта. Факты с индексом, меньше заданного, не будут выводиться в диалоговое окно. Аналогично не будут выводиться факты с индексом, больше заданного, в параметре <максимальный-индекс-факта>. Параметр <максимальное-число-фактов> задает максимально допустимое количество фактов, которые могут быть отображены в диалоговом окне.

Команды save-facts и load-facts позволяют сохранять и загружать текущий список фактов из текстового файла.

Определение 16.18. Команды save-facts и load-facts

(save-facts <имя-файла>
[<границы> <имена-конструкторов-deftemplate>*])
<границы> : := visible | local

(load-facts <имя-файла>)

Необязательный параметр <границы> команды save-facts позволяет установить границы видимости сохраняемых фактов. Если этот параметр принимает значение local (значение по умолчанию), команда save-facts сохраняет только те определенные в текущем модуле факты, конструкторы deftemplate которых также определены в данном модуле. Если параметр принимает значение visible, то сохраняются все факты, видимые в текущем модуле.

Кроме границ видимости сохраняемых фактов пользователь имеет возможность задать список конструкторов deftemplate. В этом случае команда save-facts сохранит факты только заданных шаблонов.

Команда load-facts позволяет загружать созданный ранее текстовый файл со списком фактов. Обычно подобный файл получается с помощью команды save-facts, хотя он может быть создан и вручную посредством любого тестового редактора, поддерживающего формат ASCII. Каждый факт в текстовом файле должен располагаться на отдельной строке и заключаться в круглые скобки. В случае шаблонов имена и значения каждого слота также должны обрамляться скобками.

Замечание

При загрузке фактов, соответствующих определенным шаблонам, необходимо, чтобы в момент выполнения команды load-facts все используемые шаблоны уже были определены.

Команды set-fact-duplication и get-fact-duplication предназначены для установки и проверки режима дублирования фактов. Если данный режим выключен (значение false), что является установкой по умолчанию, то CLIPS запрещает добавлять в систему факты с одинаковыми именами. Команда set-fact-duplication использует заданное логическое выражение для установки текущего значения режима дублирования фактов и возвращает предыдущее установленное значение. Команда get-fact-duplication выводит в диалоговое окно системы текущее установленное значение этого режима. Синтаксис этих команд приведен ниже.

Определение 16.19. Команды set-fact-duplication и get-fact-duplication

(set-fact-duplication <логическое-выражение>)

(get-fact-duplication)

16.4. Работа с конструкторами *deffacts*

Название команд, предназначенных для работы с конструкторами deffacts, и их краткое описание приведены в табл. 16.4.

Таблица 16.4. *Команды работы с конструкторами deffacts*

Команда	Описание
ppdeffacts	Вывод определения конструктора в диалоговое окно CLIPS
list-deffacts	Вывод в диалоговое окно системы списка всех определенных конструкторов deffacts
undeffacts	Удаление определенного конструктора deffacts

Команда ppdeffacts выводит в диалоговое окно системы определение заданного Конструктора deffacts.

Определение 16.20. Команда ppdeffacts

(ppdeffacts <имя-конструктора>)

Команда list-deffacts предназначена для отображения в диалоговом окне списка имен всех определенных в системе конструкторов deffacts.

Определение 16.21. Команда list-deffacts

(list-deffacts [<имя-модуля>])

Если необязательный параметр <имя-модуля> не задан, то данная команда выводит список имен всех конструкторов deffacts, определенных в текущем модуле. Если параметр содержит имя конкретного модуля, команда list-deffacts выводит список конструкторов, определенных в заданном модуле. В качестве параметра допускается использование символа *. В этом случае команда выведет список имен всех конструкторов deffacts, определенных во всех модулях системы.

Для удаления определенных пользователем конструкторов deffacts предназначена команда undeffacts.

Определение 16.22. Команда undeffacts

(undeffacts <имя-конструктора>)

В качестве параметра <имя-конструктора> возможно использование символа*. В этом случае команда попытается удалить все определенные пользователем Конструкторы deffacts.

16.5. Работа с правилами

CLIPS предоставляет 12 команд, предназначенных для работы с правилами и конструкторами defrule (табл. 16.5).

Таблица 16.5. *Команды работы с правилами*

Команда	**Описание**
ppdefrule	Вывод определения конструктора defrule в диалоговое окно CLIPS
list-defrules	Вывод в диалоговое окно системы списка всех определенных конструкторов defrule
undefrule	Удаление определенного конструктора defrule
matches	Просмотр списка набора данных (фактов или объектов), способных активировать заданное правило
set-break	Задание точки останова
remove-break	Удаление точки останова
show-breaks	Просмотр всех точек останова
refresh	Помещение всех текущих активаций заданного правила в план решения задачи
set-incremental-reset	Установка режима обновления правил
get-incremental-reset	Проверка текущего состояния режима обновления правил
dependencies	Вывод списка зависимостей факта или объекта
dependents	Вывод списка всех зависимых фактов или объектов

Команда ppdefrule выводит в диалоговое окно системы определение заданного конструктора defrule.

Определение 16.23. Команда ppdefrule

(ppdefrule <имя-конструктора>)

Команда list-defrules предназначена для отображения в диалоговом окне списка имен всех определенных в системе конструкторов defrules.

Определение 16.24. Команда list-defrules

(list-defrules [<имя-модуля>])

Если необязательный параметр <имя-модуля> не указан, то данная команда выводит имена всех конструкторов defrule, определенных в текущем модуле. Если параметр содержит имя конкретного модуля, команда list-defrules выводит список конструкторов, определенных в заданном модуле. Допускается использование символа *, в этом случае команда выведет в диалоговое окно имена всех конструкторов defrules, определенных во всех модулях системы.

Команда undefrule предназначена для удаления определенных пользователем конструкторов defrules.

Определение 16.25. Команда undefrules

(undefrule <имя-конструктора>)

В качестве параметра <имя-конструктора> допускается использование символа *. В этом случае команда попытается удалить все определенные пользователем правила.

Для просмотра всех наборов данных (фактов и объектов), способных активировать заданное правило, предназначена команда matches, синтаксис которой приведен ниже.

Определение 16.26. Команда matches

(matches <имя-правила>)

Пример использования данной команды приведен *в разд. 6.6.5*.

Команды set-break и remove-break предназначены для задания и удаления точек останова на указанном правиле. Эта возможность чрезвычайно полезна при отладке и проверке правильности правил сложной экспертной системы. Цикл выполнения правил прерывается перед исполнением правила, для которого определена точка останова. Если такое правило является первым в плане решения задачи, приостановка цикла выполнения правил производиться не будет.

Определение 16.27. Команды set-break и remove-break

(set-break <имя-правила>)

(remove-break [<имя-правила>])

Если при выполнении команды remove-break не указан необязательный параметр <имя-правила>, будут сняты все определенные ранее точки останова.

Для просмотра списка всех правил, на которых установлена точка останова, предназначена команда show-breaks.

Определение 16.28. Команда show-breaks

(show-breaks [<имя-модуля>])

С помощью необязательного параметра <имя-модуля> можно указать имя конкретного модуля, и в этом случае команда будет выводить только правила, определенные в заданном модуле. Если в качестве данного параметра использовать *, то команда отобразит правила с определенными

точками останова во всех модулях системы. Если параметр <имя-модуля> не задан, в диалоговое окно системы будут выведены правила, определенные в текущем модуле.

Команда refresh предназначена для помещения всех текущих активаций заданного правила в план решения задачи.

Определение 16.29. Команда refresh

(refresh <имя-правила>)

Команды set-incremental-reset и get-incremental-reset служат для установки и проверки режима *обновления правил* (incremental reset behavior mode). Если данный режим включен, что является установкой по умолчанию, то только что добавленные правила будут обновляться согласно текущему состоянию списка фактов системы. Если данный режим отключен, только что добавленные правила будут обновляться только фактами, добавленными после определения правила. Команда set-incremental-reset использует заданное логическое выражение для установки текущего значения режима обновления правил и возвращает предыдущее установленное значение. Команда get-incremental-reset выводит в диалоговое окно системы текущее установленное значение режима обновления правил (TRUE — включен, FALSE — выключен). Синтаксис этих команд приведен ниже.

Определение 16.30. Команды set-incremental-reset и get-incremental-reset

(set-incremental-reset <логическое-выражение>)

(get-incremental-reset)

Изменение режима добавления правил допускается только, если в системе еще не определен ни один конструктор defrule.

Команда dependencies определяет набор данных, от которых заданный образец (факт или объект) получил логическую поддержку *(см. разд. 6.5.8)*.

Определение 16.31. Команда dependencies

(dependencies <спецификатор-факта-или-объекта>)

Под спецификатором факта или объекта в данном случае понимается не только индекс факта или имя объекта, но и переменная, связанная с некоторым фактом или объектом в левой части правила. Таким образом, dependencies является одной из немногих команд, которые разрешается использовать при определении правой части правил.

В отличие от dependencies, команда dependents определяет набор данных, которые получили логическую поддержку от заданного образца.

Определение 16.32. Команда dependents

(dependents <спецификатор-факта-или-объекта>)

Спецификатор факта или объекта в данной команде имеет такое же значение, как и в команде dependencies.

16.6. Работа с планом решения задачи

Помимо команд, предназначенных для работы *с* правилами и конструкторами defrule, CLIPS предоставляет также несколько команд для работы непосредственно с планом решения задачи (табл. 16.6).

Таблица 16.6. *Команды работы с планом решения задачи*

Команда	**Описание**
agenda	Вывод текущего плана решения задачи
run	Запуск цикла выполнения правил
focus	Помещение модуля в стек модулей
halt	Прекращение цикла выполнения правил
set-strategy	Установка текущей стратегии разрешения конфликтов
get-strategy	Проверка текущей стратегии разрешения конфликтов
list-focus-stack	Просмотр состояния текущего стека модулей
clear-focus-stack	Удаление всех модулей из стека модулей
set-salience-evaluation	Установка режима вычисления приоритета правил
get-salience-evaluation	Проверка текущего состояния режима вычисления приоритета правил
refresh-agenda	Обновления текущего плана решения задачи

Для просмотра содержимого текущего плана решения задачи предназначена команда agenda. Она отображает в диалоговом окне все активации, содержащиеся в плане решения задачи в соответствующем порядке, вместе с данными, активировавшими правила.

Определение 16.33. Команда agenda

(agenda [<имя-модуля>])

Если необязательный параметр <имя-модуля> не задан, то команда agenda выводит список активаций текущего модуля. Если параметр содержит имя конкретного модуля, то команда выводит активации заданного модуля. В качестве параметра допускается использование символа *. В этом случае команда выведет список активаций всех модулей системы.

Команда run предназначена для запуска процесса выполнения правил.

Определение 16.34. Команда run

(run [<целочисленное-выражение>])

Если необязательный параметр <целочисленное-выражение> является положительным, то выполнение правил прекращается после заданного числа запусков правил или в случае, если текущий план решения задачи не содержит ни одной активации. Если данный параметр не указан или отрицателен, то текущий план решения задачи выполняется полностью. В случае если в момент вызова команды run стек фокусов пуст, модуль MAIN автоматически помещается в стек. Если включен режим просмотра статистической информации, после выполнения команды run пользователь получает сведения о количестве запущенных правил, общее и среднее время выполнения правил.

Команда focus помещает один или более модулей в стек модулей. Модули добавляются в стек в порядке, обратном заданному при запуске команды.

Текущим устанавливается последний модуль, помещенный в стек. Команда focus возвращает значение TRUE в случае успешного выполнения операции и FALSE — в случае неудачи.

Определение 16.35. Команда focus

(focus <имя-модуля>+)

Для отображения всех модулей, содержащихся в стеке, предназначена команда list-focus-stack.

Определение 16.36. Команда list-focus-stack

(list-focus-stack)

Команда clear-focus-stack служит для очистки стека фокусов.

Определение 16.37. Команда clear-focus-stack

(clear-focus-stack)

Для прекращения цикла выполнения правил предназначена команда halt. Эта команда, как правило, используется в левой части правил и не имеет параметров. Команда halt не оказывает никакого влияния на план решения задачи, и после ее применения выполнение можно возобновить с помощью команды run.

Определение 16.38. Команда halt

(halt)

Команды set-strategy и get-strategy предназначены для изменения и проверки текущей стратегии разрешения конфликтных ситуаций. При изменении текущей стратегии с помощью команды set-strategy возвращается предыдущая стратегия разрешения конфликтов и переупорядочивается план решения задачи. По умолчанию текущей стратегией является depth.

Определение 16.39. Команды set-strategy и get-strategy

```
(set-strategy <стратегия>)
<стратегия>      :: = depth |
                      breadth  |
                      simplicity |
                      complexity |
            lex |
            mea |
            random
(get-strategy)
```

Для изменения и проверки режима вычисления приоритета правил предназначены команды set-salience-evaluation и get-salience-evaluation соответственно. По умолчанию приоритет правил вычисляется при определении правила (значение when-defined). Команда set-salience-evaluation возвращает предыдущее установленное значение. Значение приоритета может быть вычислено в одном из трех случаев: при добавлении нового правила (when-defined), при активации правила (when-activated) и на каждом шаге основного цикла выполнения правил (every-cycle).

Определение 16.40. Команды set-salience-evaluation и get-salience-evaluation

```
(set-salience-evaluation <способ-вычисления>)
<способ-вычисления>      ::= when-defined  |
                             when-activated  |
                             every-cycle
(get-salience-evaluation)
```

Команда refresh-agenda вызывает процесс вычисления приоритетов и переупорядочивания правил в плане решения задачи.

Определение 16.41. Команда refresh-agenda

(refresh-agenda [<имя-модуля>])

Если необязательный параметр <имя-модуля> не задан, то данная команда обновляет план решения задачи текущего модуля. Если параметр содержит имя конкретного модуля, команда refresh-agenda обновляет план решения задачи заданного модуля. В качестве параметра допускается использование символа *. В этом случае команда обновит план решения задачи всех модулей системы.

16.7. Работа с глобальными переменными

Название команд, предназначенных для работы с глобальными переменными, и их краткое описание приведены в табл. 16.7.

Таблица 16.7. *Команды работы с глобальными переменными*

Команда	**Описание**
ppdefglobal	Вывод определения конструктора defglobals в диалоговое окно CLIPS
list-defglobals	Вывод в диалоговое окно системы списка всех определенных конструкторов defglobals
undefglobal	Удаление определенного конструктора defglobals
show-defglobals	Вывод в диалоговое окно системы списка и значений всех глобальных переменных
set-reset-globals	Установка режима обновления глобальных переменных
get-reset-globals	Проверка текущего состояния режима обновления глобальных переменных

Команда ppdefglobal выводит в диалоговое окно системы определение заданной глобальной переменной.

Определение 16.42. Команда ppdefglobal

(ppdefglobal <имя-глобальной-переменной>)

Имя глобальной переменной должно быть задано без вопросительного знака и символов *, т. е. name для переменной ?*name*.

Команда list-defglobals предназначена для отображения в диалоговом окне списка имен всех определенных в системе глобальных переменных.

Определение 16.43. Команда list-defglobals

(list-defglobals [<кмя-модуля>])

Если необязательный параметр <имя-модуля> не указан, то данная команда выводит имена глобальных переменных, определенных в текущем модуле. Если параметр содержит имя конкретного модуля, команда list-defglobals выводит список переменных, определенных в заданном модуле. Допускается использование символа *. В этом случае команда выведет в диалоговое окно имена всех глобальных переменных, определенных во всех модулях системы.

Команда show-defglobals, в отличие от команды list-defglobals, выводит в диалоговое окно CLIPS не только имена глобальных переменных, но и их значения. В остальном эти две команды практически идентичны.

Определение 16.44. Команда show-defglobals

(show-defglobals [<имя-модуля>])

Команда undefglobal предназначена для удаления определенных пользователем глобальных переменных.

Определение 16.45. Команда undefglobal

(undefgloba1 <имя-глобальной-переменной>)

В качестве параметра <имя-глобальной-переменной> допускается использование символа *. В этом случае команда попытается удалить все определенные пользователем глобальные переменные. Если глобальная переменная указана, например, в определении функции, удаление этой переменной закончится неудачей.

Пара команд set-reset-globals и get-reset-globals предназначена для установки и проверки режима обновления глобальных переменных. В случае если данный режим включен (значение TRUE), что является установкой по умолчанию, то при выполнении команды reset CLIPS присваивает глобальным переменным начальные значения. Команда set-reset-globals использует заданное логическое выражение для установки текущего значения режима обновления глобальных переменных и возвращает предыдущее установленное значение. Команда get-reset-globals выводит в диалоговое окно системы текущее установленное значение этого режима. Синтаксис команд приведен ниже.

Определение 16.46. Команды set-reset-globals и get-reset-globals

(set-reset-globals <логическое-выражение>)
(get-reset-globals)

16.8. Работа с конструкторами *deffunction*

Название команд, предоставляемых CLIPS для работы с конструкторами deffunction, и их краткое описание приведены в табл. 16.8.

Таблица 16.8. *Команды работы с конструкторами deffunction*

Команда	Описание
ppdeffunction	Вывод определения конструктора deffunction в диалоговое окно CLIPS
list-deffunctions	Вывод в диалоговое окно системы списка всех определенных конструкторов deffunction
undeffunction	Удаление определенного конструктора def function

Команда ppdeffunction выводит определение заданной функции на экран.

Определение 16.47. Команда ppdeffunction

(ppdeffunction <имя-функции>)

Команда list-deffunctions предназначена для отображения в диалоговом окне списка имен всех определенных в системе функций.

Определение 16.48. Команда list-deffunctions

(list-def functions)

Для удаления функций определенных пользователем с помощью конструкторов deffunction предназначена команда undeffunction.

Определение 16.49. Команда undeffunction

(undeffunction <имя-функции>)

В качестве параметра <имя-функции> возможно использование символа *. В этом случае команда попытается удалить все определенные пользователем функции. Удаление функции закончится неудачей, если выбранная функция в данный момент используется или выполняется (например, правилом).

16.9. Работа с родовыми функциями

Помимо команд, предназначенных для работы с функциями, CLIPS также предоставляет достаточно богатый набор команд для работы с родовыми функциями (табл. 16.9).

Таблица 16.9. *Команды работы с родовыми функциями*

Команда	**Описание**
ppdefgeneric	Вывод определения конструктора defgeneric в диалоговое окно CLIPS
ppdefmethod	Вывод определения конструктора defmethod в диалоговое окно CLIPS
list-defgenerics	Вывод в диалоговое окно системы списка всех определенных конструкторов defmethod
list-defmethods	Вывод в диалоговое окно системы списка всех определенных конструкторов defgeneric
undefgeneric	Удаление определенного конструктора defgeneric
undefmethod	Удаление определенного конструктора defmethod
preview-generic	Вывод в диалоговое окно системы списка всех методов, применимых к заданному набору аргументов

Команда ppdefgeneric выводит в диалоговое окно заголовок выбранной родовой функции (явно созданной пользователем с помощью конструктора defgeneric или неявно созданной системой при определении метода).

Определение 16.50. Команда ppdefgeneric

(ppdefgeneric <имя-родовой-функции>)

Команда ppdefmethod предназначена для вывода определения конкретного метода выбранной родовой функции. Выбор метода осуществляется с помощью его индекса.

Определение 16.51. Команда ppdefmethod

(ppdefmethod <имя-родовой-функции> <индекс-метода>)

Пара команд list-defgenerics и list-defmethods предназначена для отображения в диалоговом окне списка имен родовых функций и их методов соответственно.

Определение 16.52. Команды list-deffgenerics и list-defmethods

(list-deffgenerics [<имя-модуля>])
(list-defmethods [<имя-родовой-функиии>])

С помощью необязательного параметра <имя-модуля> команды defgenerics можно указать имя конкретного модуля, и в этом случае команда будет выводить только родовые функции, определенные в заданном модуле. Если в качестве данного параметра использован символ *, то команда отобразит родовые функции, определенные во всех модулях системы. Если параметр <имя-модуля> не задан, в диалоговое окно системы будут выведены заголовки родовых функций текущего модуля.

Параметр <имя-родовой-функции> функции list-defmethods определяет конкретную функцию, методы которой будут выведены на экран. Если этот параметр не задан, команда выведет методы всех родовых функций, определенных в текущем модуле. Для каждой родовой функции методы отображаются в порядке приоритета вместе с их индексами.

Команды undefgeneric и undefmethod предназначены для удаления заголовков родовых функций и их методов.

Определение 16.53. Команды undefgeneric и undefraethod

(undefgeneric <имя-родовой-функции>)
(undefmethod <имя-родовой-функции> <индекс-метода>)

В качестве параметра <имя-родовой-функции> команды undefgeneric возможно указание символа *. В этом случае команда попытается удалить все определенные пользователем родовые функции и их методы. Удаление родовой функции закончится неудачей, если выбранная функция в данный момент используется или выполняется (например, правилом).

Параметр <имя-метода> команды undefmethod определяет имя конкретного метода для удаления. Если в качестве данного параметра задан символ *, то команда undefmethod удалит все методы выбранной родовой функции. Отличие данной команды от команды undefgeneric состоит в том, что даже при удалении всех методов некоторой функции команда оставляет ее заголовок. В случае если вместо имени конкретной родовой функции использовался символ * (значение параметра <имя-метода> в такой ситуации также должно равняться *), будут удалены все методы всех определенных пользователем родовых функций. Заголовки всех функций при этом останутся в системе.

Команда preview-defgeneric чрезвычайно полезна при отладке родовых функций. Она отображает список всех методов выбранной родовой функции, *применимых* к заданному набору аргументов, в отличие от команды list-defmethod, которая отображает список *всех* методов выбранной родовой функции. Выполнение данной команды фактически не выполняет методы родовой функции, однако производит проверку соответствия заданных аргументов, вычисление и проверку заданных ограничений, а также выполняет родовое связывание, что может вызывать различные побочные эффекты.

Определение 16.54. Команда preview-generic

(preview-generic <имя-родовой-функции> <выражение>*)

Параметр <выражение> в данном определении задает аргумент родовой функции. В качестве примера можно использовать следующие действия (см. также рис. 16.3).

Пример 16.4. Работа с командой defmethod

```
(clear)
(defmethod + ((?a NUMBER) (?b INTEGER)))
(defmethod + ((?a INTEGER) (?b INTEGER)))
(defmethod + ((?a INTEGER) (?b NUMBER)))
(defmethod + ((?a NUMBER) (?b NUMBER)
                ($?rest PRIMITIVE)))
(defmethod + ((?a NUMBER) (?b INTEGER (> ?b 2))))
(defmethod +((?a INTEGER (> ?a 2))
              (?b INTEGER (> ?b 3))))
(defmethod +((?a INTEGER (> ?a 2)) (?b NUMBER))) (preview-generic +45)
```

Результат выполнения приведенных выше команд представлен на рис. 16.3.

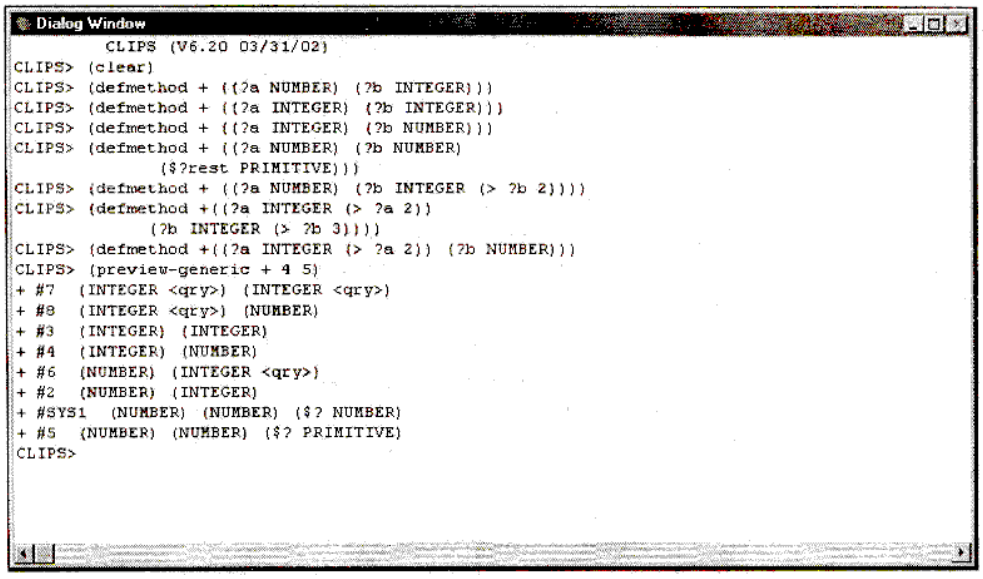

Рис. 16.3. Использование команды defmethod

16.10. Работа с классами и объектами

CLIPS предоставляет достаточно богатый набор возможностей, предназначенный для работы с классами и объектами, который включает в себе команды для работы с конструкторами defclass, defmessage-handler, definstances и команды для работы с экземплярами объектов (табл. 16.10).

Таблица 16.10. *Команды для работы с классами и объектами*

ppdefclass	Вывод определения конструктора defclass в диалоговое окно CLIPS
list-defclasses	Вывод в диалоговое окно системы списка всех определенных конструкторов defclass
undefclass	Удаление определенного конструктора defclass
describe-class	Вывод в диалоговое окно подробного описания класса
browse-classes	Вывод в диалоговое окно иерархии наследования класса
ppdefmessage-handler	Вывод определения конструктора defmessage-handler в диалоговое окно CLIPS
list-defmessage-handlers	Вывод в диалоговое окно системы списка всех определенных конструкторов defmessage-handler
undefmessage-handler	Удаление определенного конструктора defmessage-handler
preview-send	Вывод в диалоговое окно системы списка всех применимых сообщений заданного типа и класса
ppdefinstances	Вывод определения конструктора definstances в диалоговое окно CLIPS
1ist-definstances	Вывод в диалоговое окно системы списка всех определенных конструкторов definstances
undefinstances	Удаление определенного конструктора definstances
instances	Вывод в диалоговое окно системы текущего списка обьектов
ppinstance	Вывод содержимого слотов объекта
save-instances	Сохранение объектов в текстовый файл
bsave-instances	Сохранение объектов в бинарный файл
load-instances	Загрузка объектов из текстового файла
restore-instances	Загрузка объектов из текстового файла без использования сообщений
bload-instances	Загрузка объектов из бинарного файла

Команда ppdefdass выводит в диалоговое окно определение класса, созданного пользователем с помощью конструктора defclass.

Определение 16.55. Команда ppdefclass

(ppdefclass <имя-класса>)

Команда list-defdasses отображает в диалоговом окне список имен всех (как определенных пользователем, так и системных) классов. Если параметр <имя-модуля> не задан, команда выводит список имен классов, определенных в текущем модуле. Если в качестве данного аргумента использовать имя конкретного модуля в диалоговое окно будут выведены классы, определенные в указанном модуле. В случае если в качестве параметра <имя-модуля> использован символ *, будет выведен список классов, определенных во всех модулях системы.

Определение 16.56. Команда list-defclasses

(list-defclasses [<имя-модуля>])

Для удаления определения класса, созданного пользователем с помощью конструкторов defdass, предназначена команда undefdass.

Определение 16.57. Команда undefclass

(undefclass <имя-класса>)

Данная команда удаляет также все классы, являющиеся наследниками класса, указанного в параметре <имя-класса>. В качестве этого параметра возможно указание символа *. В этом случае команда попытается удалить все определенные пользователем классы. Удаление определения класса закончится неудачей, если в списке объектов системы присутствует хотя бы один объект указанного класса или данный класс используется в качестве ограничений параметров родовой функции.

Команда describe-class предназначена для получения подробного описания класса, которое включает в себя: описание роли класса (абстрактный или конкретный класс), список прямых суперклассов и наследников данного класса, список предшествования классов, список слотов с именами классов, от которых они унаследованы, и значениями всех граней, а также список всех присоединенных к данному классу обработчиков сообщений.

Определение 16.58. Команда describe-class

(describe-class <имя-класса>)

Для демонстрации работы команды describe-class используем следующий пример:

Пример 16.5. Работа с командой describe-class

```
(clear)
(defclass CHILD (is-a USER)
            (role abstract)
            (multislot parents (cardinality 2 2))
            (slot age (type INTEGER)
                        (range 0 18))
            (slot sex (access read-only)
            (type SYMBOL)
             (allowed-symbols male female)
             (storage shared))
 )
(defclass BOY (is-a CHILD)
            (slot sex (source composite)
               (default male))
              )
(defmessage-handler BOY play ()
                        (printout t "The boy is now playing..." crlf) )
(describe-class CHILD)
```

Результат выполнения приведенных выше команд представлен на рис. 16.4.

В табл. 16.11 содержится список полей и их возможные значения, использующиеся для описания свойств и граней слотов заданного класса.

Таблица 16.11. *Поля, использующиеся для описания свойств и граней слотов заданного класса*

Поле	**Значение**	**Описание**
FLD	SGL/MLT	Тип слота (простой или составной)
DEF	STC/DYN/NIL	Значение по умолчанию (статическое, динамическое или отсутствует)
PRP	INH/NIL	Распространение по наследованию (наследуется или нет)
ACC	RW/R/INT	Разрешенный доступ (чтение-запись, чтение, инициализация)
STO	LCL/SHR	Тип хранения (в экземпляре объекта или в классе)
MCH	RCT/NIL	Активность при сопоставлении образцов (активный или нет)
SRC		Источник свойств унаследованного слота (составной или обычный)
VIS	PUB/PRV	Видимость (открытый или закрытый)
CRT	R/W/RW/NIL	Автоматическое создание акцессоров (чтение, запись, чтение-запись или отсутствуют)
OVRD-MSG	<имя сообщения>	Имя сообщения, посылаемого для переопределения слота функцией make-instance и др.
SOURCE(S)	<класс>+	Класс-источник слота (если слот унаследован от нескольких классов — список классов)

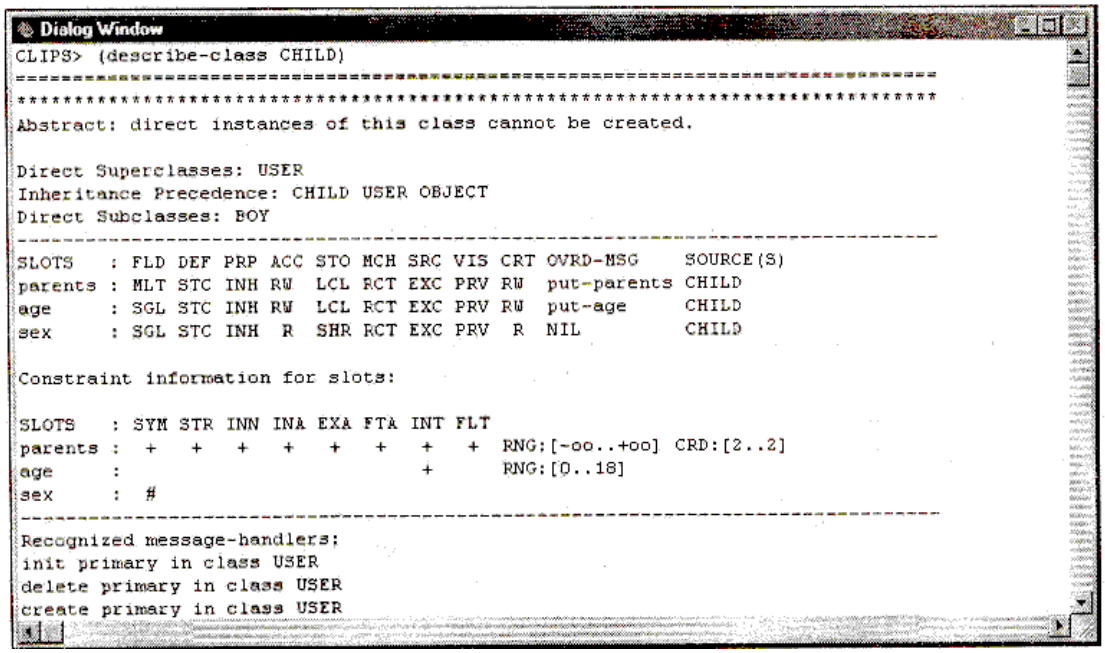

Рис. 16.4. Результат выполнения команд из примера 16.5

Помимо описания граней слотов команда describe-dass отображает также список всех ограничений типов, используемых для каждого слота. Список сокращений, применяемых для описания ограничений, приведен в табл. 16.12. В случае если определенный тип отмечен знаком +, то слот может содержать любые значения соответствующего типа. Если тип отмечен знаком #, это означает, что слот может принимать только некоторые заданные значения этого типа.

Таблица 16.12. *Используемые сокращения*

Аббревиатура	**Описание**
SYM	SYMBOL
STR	Строка (string)
INN	Имя объекта (Instance Name)
INA	Адрес объекта (Instance Address)
EXA	Внешний адрес (External Address)
FTA	Адрес факта(Fact Address)
INT	Целое (INTEGER)
FLT	Вещественное (float)
RNG	Диапазон
CRD	Мощность

Команда browse-classes предназначена для отображения в диалоговом окне системы иерархии наследования классов, определенных в данный момент в системе.

Определение 16.59. Команда browse-classes

(browse-classes [<имя-класса>])

В случае если необязательный аргумент <имя-класса> не задан, CLIPS отобразит дерево наследования классов, начиная с предопределенного класса OBJECT. Если указать имя конкретного класса, CLIPS выведет дерево наследования заданного класса. Некоторые классы могут появляться в дереве наследования несколько раз при использовании множественного наследования. Символом * отмечаются классы, являющиеся прямыми наследниками более чем одного класса.

Пример 16.6. Использование команды browse-classes

```
(defclass      a      (is-a USER))
(defclass      b      (is-a USER))
(defclass      c      (is-a a b))
(defclass      d      (is-a USER))
(defclass      e      (is-a c d))
(defclass      f      (is-a e))
(browse-classes USER)
```

Результат выполнения приведенных выше команд представлен на рис. 16.5.

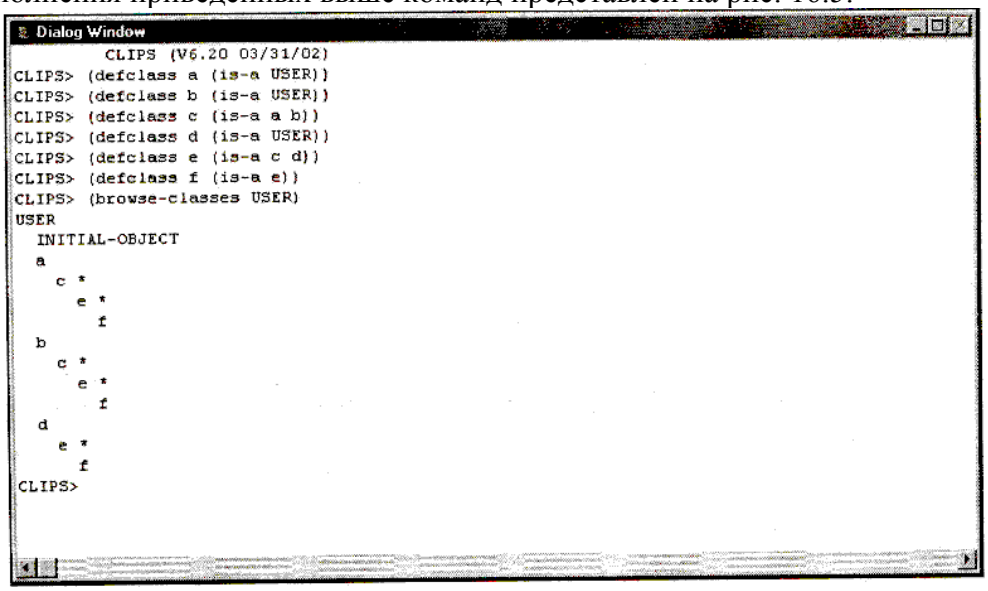

Рис. 16.5. Использование команды browse-classes

С помощью команды ppdefmessage-handler пользователь может вывести определение конструктора defmessage-handler в диалоговое окно системы.

Определение 16.60. Команда ppdefmessage-handler

(ppdefmessage-handler <имя-класса> <имя-обработчика> [<тип-обработчика>])

<тип-обработчика> ::= around | before | primary | after

Параметры <имя-класса> и <имя-обработчика> используются для указания конкретного обработчика, определение которого необходимо вывести. Необязательный параметр <тип-обработчика> определяет тип выводимого обработчика сообщения. По умолчанию этот параметр принимает значение primary.

Команда list-defmessage-handiers предназначена для отображения в диалоговом окне списка имен обработчиков сообщений. Без указания необязательного параметра <имя-класса> команда выведет на экран список обработчиков сообщений всех классов, присутствующих в системе. В случае если задан конкретный класс, будут выведены только его обработчики сообщений. Если при этом будет указан необязательный параметр inherit, выведутся также все унаследованные обработчики сообщений указанного класса.

Определение 16.61. Команда list-defmessage-handlers

(list-defmessage-handlers [<имя-класса> [inherit]])

Пример 16.7. Использование команды list-defmessage-handlers

(defclass A (is-a USER))

(defmessage-handler A foo ())

(list-defmessage-handlers A)

(list-defmessage-handlers A inherit)

Результат выполнения приведенных выше команд представлен на рис. 16.6.

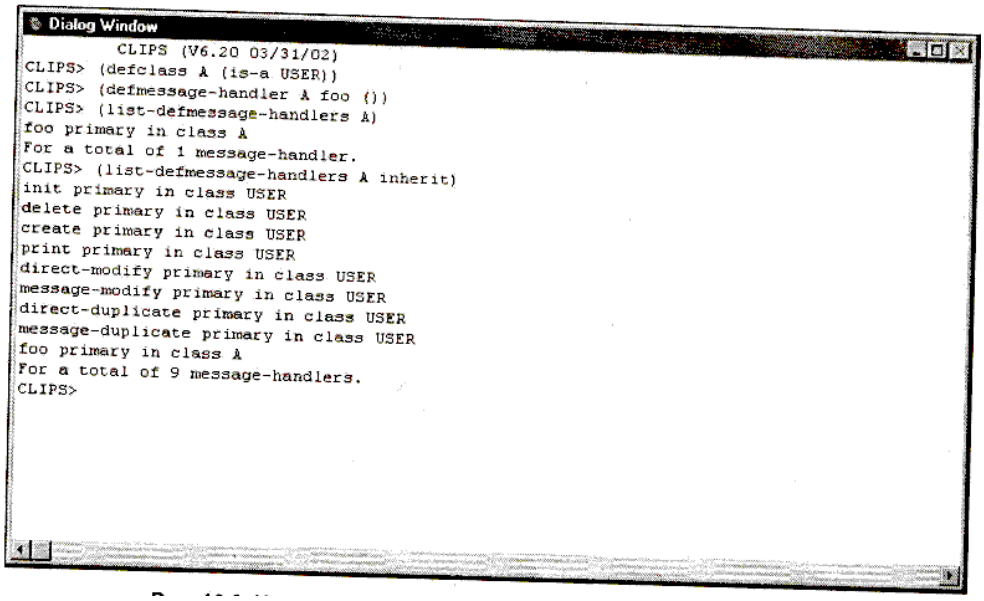

Рис. 16.6. Использование команды list-defmessage-handlers

Для удаления обработчиков сообщений предназначена команда undefmes-sage-handler.

Определение 16.62. Команда undefmessage-handler

(undefmessage-handler <имя-класса> <имя-обработчика> [<тип-обработчика>])
<тип-обработчика> ::= around | before | primary | after

Назначение аргументов данной команды аналогично назначению аргументов команды ppdefmessage-handler. В качестве любого аргумента может быть использован символ *.

Команда preview-send предназначена для отображения всех обработчиков, применимых к заданному сообщению.

Определение 16.63. Команда preview-send

(preview-send <имя-класса> <имя-сообщения>)

Пример 16.8. Использование команды preview-send

```
(defmessage-handler USER my-message  around   ()
                        (call-next-handler))
(defmessage-handler USER nay-message before   ())
(defmessage-handler USER my-message   ()
                        (call-next-handler))
(defmessage-handler USER my-message after ())
(defmessage-handler OBJECT my-message around ()
                        (call-next-handler))
(defmessage-handler OBJECT my-message before ())
(defmessage-handler OBJECT my-message ())
(defmessage-handler OBJECT my-message after ())
(preview-send USER my-message)
```

Результат выполнения приведенных выше команд представлен на рис. 16.7.

Команда ppdefinstances выводит в диалоговое окно системы определение заданного конструктора definstances.

Определение 16.64. Команда ppdefinstances

(ppdefinstances <имя-конструктора>)

Команда list-definstances предназначена для отображения в диалоговом окне списка имен всех определенных в системе конструкторов definstances.

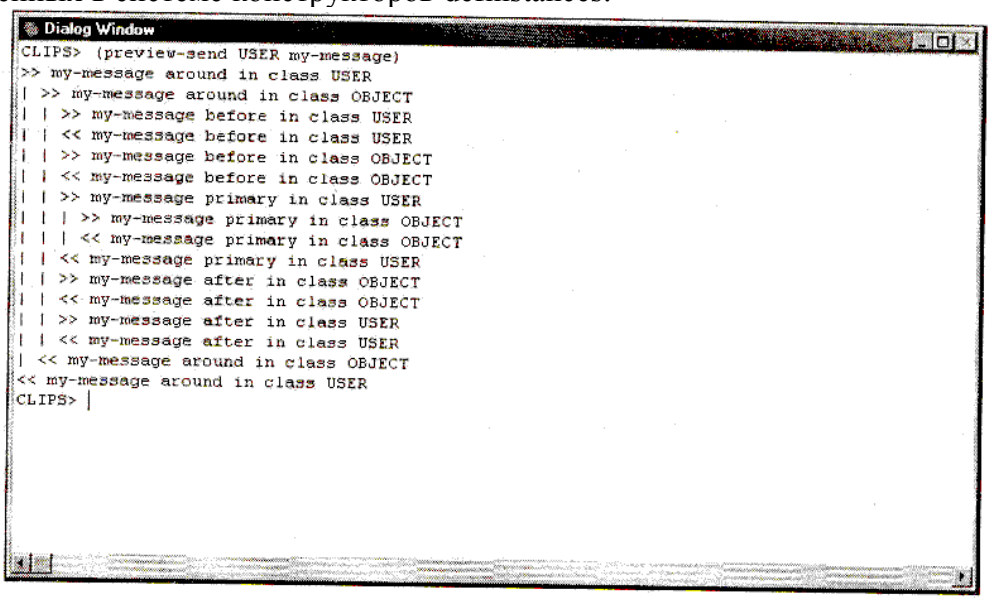

Рис. 16.7. Использование команды preview-send

Определение 16.65. Команда list-definstances

(list-definstances)

Для удаления определенных пользователем конструкторов definstances служит команда undefinstances.

Определение 16.66. Команда undefinstances

(undefinstances <имя-конструктора>)

В качестве параметра <имя-конструктора> возможно использование символа *. В этом случае команда попытается удалить все определенные пользователем Конструкторы definstances.

Команда instances предназначена для отображения списка объектов определенных пользователем классов.

Определение 16.67. Команда instances

(instances [<имя-модуля> [<имя-класса> [inherit]]])

Если имя модуля не указано, команда выведет список имен объектов всех классов, находящихся в области видимости данного класса. В противном случае будут выведены объекты классов, находящихся в области видимости конкретного модуля. Если в качестве аргумента <имя-модуля> использован символ *, команда отобразит список объектов всех модулей системы. При помощи аргумента <имя-класса> команда отобразит только имена объектов заданного класса. В случае указания необязательного ключа inherit на экран будут выведены также имена всех подклассов заданного класса.

Применение команды ppinstance возможно только внутри обработчика сообщения, созданного пользователем класса. Эта команда предназначена для вывода на экран содержимого слотов объекта, получившего сообщения, и использует в своей реализации сообщение print.

Определение 16.68. Команда ppinstance

(ppinstance)

CLIPS позволяет сохранять определенные в системе объекты в текстовый файл, формат которого приведен ниже.

Определение 16.69. Формат файла, содержащего объекты

(<имя-объекта> of <имя-класса> <определение-слота>*)

<определение-слота> ::= (<имя-слота> <текущее-значение>)

Файл включает определения всех слотов объекта независимо от того, содержат ли они значения по умолчанию или нет.

Для сохранения объектов в текстовый файл предназначена команда save-instances.

Определение 16.70. Команда save-instances

(save-instances <имя-файла>
[local | visible [[inherit] <имя-класса>+]])

По умолчанию (ключ local) команда сохраняет только объекты определенных в текущем модуле классов. При указании ключа visible команда сохранит объекты всех видимых в данном модуле классов. Команда save-instances позволяет указать список классов. В этом случае будут сохранены объекты только указанных в списке классов. При указании ключа inherit будут сохранены также объекты всех подклассов указанного класса. По окончании работы команда save-instances возвращает количество сохраненных объектов.

Для *загрузки* объектов из текстового файла служат команды load-instances и restore-instances. Отличие этих команд заключается в том, что команда restore-instances не использует сообщения для присвоения значений слотам загружаемых объектов.

Определение 16.71. Команды load-instances и restore-instances

(load-instances <имя-файла>)

(restore-instances <имя-файла>)

Использование обеих команд идентично применению нескольких последовательных вызовов функции make-instance. После завершения команды возвращают количество успешно загруженных объектов.

Замечание

При загрузке объектов необходимо наличие определения соответствующих классов. Кроме того, перед загрузкой требуется установить текущим необходимый модуль, т. к. текстовый файл не содержит информацию о модуле, в котором должен содержаться загружаемый объект.

Помимо команд, позволяющих загружать и сохранять объект в текстовый файл, CLIPS предоставляет пару команд с идентичным синтаксисом, предназначенных для работы с бинарными файлами, — bsave-instances и bload-instances.

Определение 16.72. Команды bsave-instances и bload-instances

(bsave-instances <имя-файла>
[local | visible [[inherit] <имя-класса>+]])
(bload-instances <имя-файла>)

Загрузка бинарных файлов происходит значительно быстрее. Однако, в отличие от текстовых, создание которых возможно в любом текстовом редакторе, поддерживающем формат ASCII, бинарные файлы можно создать только с помощью команды bsave-instances из среды CLIPS.

16.11. Работа с конструкторами *defmodule*

Перечень команд, предоставляемых CLIPS для работы с конструкторами defmodule, и их краткое описание приведены в табл. 16.13.

Таблица 16.13. *Команды работы с конструкторами defmodule*

Команда	Описание
ppdefmodule	Вывод определения конструктора defmodule в диалоговое окно CLIPS
list-defmodules	Вывод в диалоговое окно системы списка всех определенных конструкторов defmodule

С помощью команды ppdefmodule пользователь может вывести определение конструктора defmodule в диалоговое окно системы.

Определение 16.73. Команда ppdefmodule

(ppdefmodule <имя-модуля>)

Команда list-defmodules предназначена для отображения в диалоговом окне списка имен всех определенных в системе модулей.

Определение 16.74. Команда list-defmodules

(list-defmodules)

16.12. Профилирование и отладка

CLIPS предоставляет несколько специальных команд, предназначенных для профилирования и отладки ваших программ (табл. 16.14).

Таблица 16.14. *Команды профилирования и отладки*

Команда	**Описание**
set-profile-percent-threshold	Установка порогового времени профилирования
get-profile-percent-threshold	Проверка текущего значения порогового времени профилирования
profile-reset	Сброс всей собранной информации о профилировании правил и функций
profile-info	Отображение собранной профилирующей информации
profile	Включение/выключение режима профилирования
dribble-on	Создание трассировочного файла
dribble-off	Закрытие трассировочного файла
watch	Включение режима просмотра списка элементов среды CLIPS
unwatch	Выключение режима просмотра списка элементов среды CLIPS
list-watch-items	Просмотр текущего состояния режима просмотра списка элементов среды

Команды set-profile-percent-threshold и get-profile-percent-threshold предназначены для установки и проверки текущего состояния процента порогового времени профилирования. Это значение является минимальным пороговым временем выполнения конструктора или функции, которое будет отображено с помощью команды profile-info. По умолчанию значение равно 0. Таким образом, поведением системы по умолчанию является вывод всей профилирующей информации.

Определение 16.75. Команды set-profile-percent-threshold и get-profile-percent-threshold

(set-profile-percent-threshold <целочисленное-выражение>)

(get-profile-percent-threshold)

С помощью команды profile-reset можно очистить всю накопленную профилирующую информацию.

Определение 16.76. Команда profile-reset

(profile-reset)

Команда profile-info выводит в диалоговое окно системы всю собранную профилирующую информацию о работе конструкторов или функций, определенных пользователем. Собранная информация отображается в таблице из шести столбцов. Описание содержания всех столбцов приведено ниже.

- Имя профилируемого конструктора или определенной пользователем функции.
- Количество вызовов профилируемого конструктора или определенной пользователем функции.
- Продолжительность выполнения конструктора или определенной пользователем функции.
- Процент времени, потраченного на выполнение конструктора или определенной пользователем функции, относительно общего времени профилирования,
- Продолжительность выполнения конструктора или определенной пользователем функции и выполнения всех внутренних вызовов конструкторов или функций.
- Процент времени, потраченного на выполнение конструктора или определенной пользователем функции и выполнения всех внутренних вызовов конструкторов или функций, относительно общего времени профилирования.

Определение 16.77. Команда profile-info

(profile-info)

Команда profile предназначена для включения/выключения профилирования конструкторов или определенных пользователем функций.

Определение 16.78. Команда profile

(profile constructs | user-functions | off)

При профилировании конструкторов система замеряет время выполнения определенных пользователем и родовых функций, а также обработчиков сообщений, вызванных из правой части правил. При профилировании функций замеряется время выполнения системных и определенных пользователем функций. Эти режимы не совместимы и устанавливаются с помощью ключей constructs и user-function соответственно. Для выключения режима профилирования используется ключ off.

Для примера профилирования очистите систему CLIPS и добавьте следующие определения:

Пример 16.9. Необходимые определения

```
(clear)
(deffacts start (fact 1) )
(deffunction function-1 (?x)
                (bind ?y 1)
                (loop-for-count (* ?x 10)
                                (bind ?y (+ ?y ?x))))

(defrule rule-1
        ?f <- (fact ?x&: (< ?x 100))
        =>

        (function-1 ?x)
        (retract ?f)
        (assert (fact (+ ?x 1))))
```

Для профилирования конструктора правил используйте следующую последовательность команд:

Пример 16.10. Профилирование конструкторов

```
(reset) (progn
        (profile constructs)
        (run)
        (profile off))
(profile-info)
```

Результат выполнения приведенных выше команд приведен на рис. 16.8.

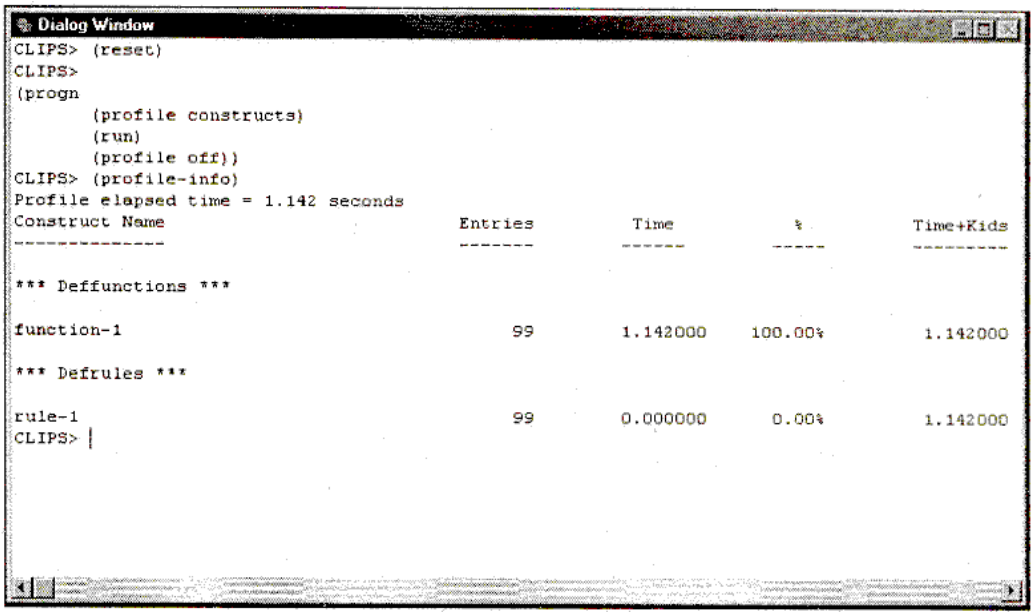

Рис. 16.8. Профилирование конструкторов

Профилирование определенных пользователем функций осуществляется следующим образом:

Пример 16.11. Профилирование функций

```
(profile-reset)
(reset)
(progn
        (profile user-functions)
        (run)
        (profile off)) (profile-info)
```

Результат выполнения приведенных выше команд приведен на рис. 16.9.

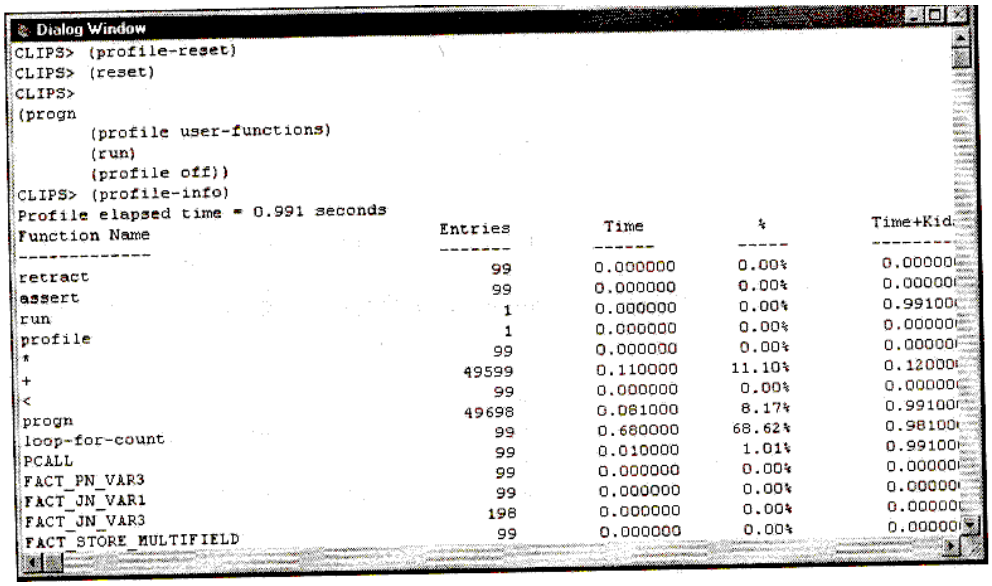

Рис. 16.9. Профилирование функций

Команды dribble-on и dribble-off предназначены для создания и закрытия трассировочного файла. Они возвращают значение TRUE в случае успешного выполнения и FALSE — в случае неудачи. После выполнения команды dribbie-on вся информация, выводимая в диалоговое окно системы, помещается также в файл, заданный параметром <имя-файла>.

Определение 16.79. Команды dribble-on и dribble-off

(dribble-on <имя-файла>)

(dribble-off)

С помощью команды watch можно установить тот или иной режим просмотра информации об изменении набора фактов, объектов, конструкторов и т. д. Обобщенный синтаксис команды watch приведен ниже.

Определение 16.80. Команда watch

```
(watch <элемент>) <элемент> : : = all |
                    compilations |
                    statistics |
                    focus |
                    messages |
                    deffunctions <имя-функции>* |
                    globals <имя-глобальной-переменной>* |
                    rules <имя-правила>* |
                    activations <имя-правила>* |
                    facts <имя-шаблона>* |
                    instances <имя-класса>* |
                    slots <имя-класса>* |
                    message-handlers <определение-обработчика-1>*
                                [<определение-обработчика-2>]) |
                    generic-functions <имя-родовой-функции>* |
                    methods <определение-метода-1>*
                                [<определение-метода-2>]
<определение-обработчика-1> ::= <имя-класса>
                                <имя-обработчика> <тип-обработчика> <определение-
обработчика-2> ::= <имя-класса>
                                <имя-обработчика> [<тип-обработчика>]
<определение-метода-1>    : : =  <имя-родовой-функции> <method-index>
<определение-метода-2>    : : =  <имя-родовой-функции> [<индекс метода>]
```

Ключ all включает все возможные режимы отображения. Ключ compilations отображает процесс добавления конструкторов в систему. Применение остальных ключей уже было описано выше в соответствующих главах.

Команда unwatch предназначена для отключения режимов просмотра информации об изменении, включенных с помощью команды watch.

Определение 16.81. Команда unwatch

(unwatch <элемент>)

С помощью команды list-watch-items можно увидеть текущее состояние просмотра того или иного элемента.

Определение 16.82. Команда list-watch-items

(list-watch-items [<элемент>])

Необязательный параметр <элемент> может указывать конкретный режим просмотра, состояние которого необходимо вывести на экран. В случае если данный параметр отсутствует, будет выведена информация обо всех элементах. Пример использования команды list-watch-items приведен на рис. 16.10.

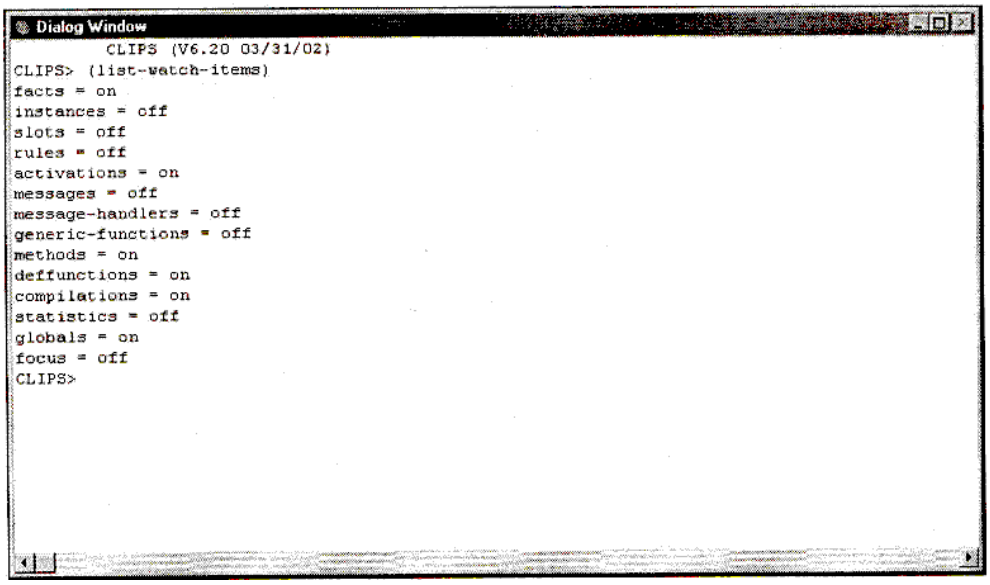

Рис. 16.10. Использование команды list-watch-items

16.13. Управление памятью

Для управления памятью CLIPS предоставляет несколько специальных команд (табл. 16.15).

Таблица 16.15. *Команды управления памятью*

Команда	**Описание**
mem-used	Определение объема памяти, используемого системой
mem-requests	Определяет количество сделанных запросов на получение дополнительной оперативной памяти
release-mem	Возвращает всю освободившуюся память оперативной системе
conserve-mem	Включение режима экономии памяти

Команда mem-used предназначена для определения объема памяти, используемой системой. Команда возвращает целое число, равное количеству байтов, используемых CLIPS в данный момент. Это число не включает в себя служебную память, необходимую операционной системе для управления процессом функционирования программы.

Определение 16.83. Команда mem-used

(mem-used)

С помощью команды mem-requests пользователь может определить число запросов, сделанных CLIPS к операционной системе для выделения дополнительной оперативной памяти.

Определение 16.84. Команда mem-requests

(mem-requests)

Использование команды release-mem позволяет системе возвращать освободившуюся память операционной системе. Это может помочь операционной системе более эффективно управлять свободной памятью. Обычно команда вызывается автоматически, в случае если для выполнения текущих задач CLIPS не нуждается в большом количестве памяти. Данная команда рекомендуется только опытным программистам, точно оценивающим потребности системы, т. к. чрезмерное освобождение памяти может отрицательно сказаться на скорости работы системы. После выполнения процедуры освобождения памяти команда release-mem возвращает объем освобожденной памяти в байтах.

Определение 16.85. Команда release-mem

(release-mem)

Команда conserve-mem предназначена для включения и выключения режима экономии памяти.

Определение 16.86. Команда conserve-mem

(conserve-mem on | off)

Экономия памяти осуществляется за счет неиспользования определений конструкторов, необходимых для вывода в диалоговое окно (например, с помощью команд ppdefrule, ppdeftemplate и т. д.). Это помогает сильно экономить память в достаточно больших системах. Включение данного режима следует выполнить до загрузки конструкторов в систему.

ЧАСТЬ VI.Приложения.

ПРИЛОЖЕНИЕ1. Основные БНФ-определения

В данном приложении собраны БНФ-определения наиболее важных определений и конструкторов среды CLIPS.

Определение П1.1. Представление целого числа

<целое> ::=[+ | -] <цифра>+
<цифра> ::=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Определение П1.2. Представление вещественного числа

<вещественное>: := <целое> <экспонента> |
<целое> . [экспонента] |
<беззнаковое-целое> [экспонента] |
<целое> . <беззнаковое-целое> [экспонента]
<беззнаковое-целое> : : = <цифра>+
<экспонента>: := e | E <целое>

Определение П1.3. Упорядоченный факт

(данное_типа_symbol [поле] *)

Определение П1.4. Синтаксис конструктора deftemplate

(deftemplate <имя-шаблона> [<необязательные-комментарии>]
[<определение-слота>*])

<определение-слота> ::= <определение-простого-слота> |
<определение-составного-слота

<определение-простого-слота> ::= (slot <имя-поля>
<атрибуты-шаблона>)

<определение-составного-слота> ::= (multislot <имя-поля>
<атрибуты-шаблона>)

<атрибуты-шаблона> ::= <атрибут-значение-по-умолчанию>|
<атрибут-ограничения>

<атрибут-значение-по-умолчанию> ::= (default ?DERIVE|?NONE |
<Выражение>)|
(default-dynamic <Выражение>)

Определение П1.5. Синтаксис конструктора deffacts

(deffacts <имя-списка-фактов> [<необязательные-комментарии>]
[<факт>*])

Определение П1.6. Предопределенные шаблоны и факты

(deftemplate initial-fact)
(deffacts initial-fact

(initial-fact))

Определение П1.7. Синтаксис конструктора defrule

```
(defrule
        <имя-правила>
        [<комментарии>]
        [<определение-свойства-правила>]
        <предпосьшки >                  ; левая часть правила
        =>
        <следствие>                     ; правая часть правила
                )
```

Определение П1.8. Синтаксис свойств правил

```
<определение-свойства-правила> ::= (declare <свойство-правила>)
<свойство-правила>             ::= (salience <целочисленное выражение>) |
                                   (auto-focus TRUE|FALSE)
```

Определение П1.9. Синтаксис условного элемента

```
<условный-элемент>  ::=  <pattern-CE>   |
                         <assigned-pattern-CE>  |
                         <not-CE>   |
                         <and-CE>   |
                         <or-CE>    |
                         <logical-CE>   |
                         <test-CE>  |
                         <exists-CE>    |
                         <forall-CE>
```

Определение П1.10. Синтаксис символьных ограничений для неупорядоченного факта

(<ограничение-1> ... <ограничение-n>)

Определение П1.11. Синтаксис символьных ограничений для шаблона

```
(<имя-шаблона >     (<имя-слота-1> <ограничение-1>)
                     ...
                    (<имя-слота-n>  <ограничение-n>))
```

Определение П1.12. Синтаксис связывающих ограничений

<элемент-1>& <элемент-2> ... & элемент -n>
<элемент-1> | <элемент-2> ... | <элемент -n>
~ <элемент>

Определение П1.13. Синтаксис предикатного ограничения

:<вызов-функции>

Определение П 1.14. Синтаксис ограничения, возвращающего значение

=<вызов-функции>

Определение П 1.15. Синтаксис понятия "элемент"

<элемент> ::= <константа> |
<простая-переменная> |
<составная-переменная>|
:<вызов-функции>
=<вызов-функции>

Определение П1.16. Синтаксис ограничений

<ограничение> ::= <символьное-ограничение > |
? |
$? |
<связанное-ограничение>

<связанное-ограничение> ::= <простое-ограничение> |
<простое-ограничение>&<связанное-ограничение> |
<простое-ограничение> | <связанное-ограничение> |

<простое-ограничение> ::= <элемент>|~ <элемент>

<элемент> ::= <константа>|
<простая-переменная> |
<составная-переменная>

Определение П1.17. Синтаксис образцов объектов

<образец объекта> ::= (object <атрибуты-ограничения>)

<атрибуты-ограничения> ::= (is-a <ограничение>) |
(name <ограничение>) |
(slot <ограничение>)

Определение П1.18. Синтаксис адреса образца

<адрес-образца> ::= ?<имя-переменной> <- <образец>

Определение П1.19. Синтаксис условного элемента test

<условный-элемент-test > :: = (test <вызов-функиии>)

Определение П1.20. Синтаксис условного элемента or

<условный-элемент-ог > :: = (or <условный-элемент>+)

Определение П1.21. Синтаксис условного элемента and

<условный –элемент- and> :: =(and <условный элемент>+)

Определение П 1.22. Синтаксис условного элемента not

<условный-элемент-not> ::= (not <условный-элемент>)

Определение П 1.23. Синтаксис условного элемента exists

<условный-элемент-exists> :: = (exists <условный-элемент>+)

Определение П 1.24. Синтаксис условного элемента forall

<условный-элемент-forall> ::= (forall <условный-элемент>
<условный-элемент>+)

Определение П1.25. Синтаксис условного элемента logical

<условный-элемент-1ogica1> ::= (logical <условный-элемент>+)

Определение П 1.26. Синтаксис предопределенного факта и объекта

(initial-fact)
(object (is-a INITIAL-OBJECT) (name [initial-object]))

Определение П 1.27. Синтаксис конструктора defglobal

(defglobal [<имя-модуля>] <определение-переменной>*)
<определение-переменной> ::= <имя-переменной> = <выражение>
<имя-переменной> : : = ?*<значение-типа-зутЬо1>*

Определение П 1.28. Синтаксис конструктора def function

(deffunction <имя-функции>
[<комментарии>]
<обязательные-параметры>
[<групповой-параметр>] <действия>)

<обязательные-параметры> ::= <выражение-простое-поле>

<групповой-параметр> ::= <выражение-составное-поле>

Определение П1.29. Синтаксис конструктора defgeneric

(defgeneric <имя-функции> [комментарии])

Определение П1.30. Синтаксис конструктора defmethod

(defmethod <имя-функции>
[<индекс>]
[<комментарии>]
(<ограничения-параметра>*
[<групповой-параметр>])
<действие>*)
<ограничения-параметров> ::= <простая-переменная> |
(<простая-переменная>
<ограничение-по-типу>*
[<ограничение-по-запросу>])
<групповой-параметр> ::= <составная-переменная> |
(<составная-переменная>
<ограничение-по-типу>*
[<ограничение-по-запросу >])
<ограничение-по-типу> ::= <имя-класса>

<ограничение-по-запросу> ::= <глобальная-переменная> |
<вызов-функции>

Определение П1.31. Синтаксис конструктора defclass

(defclass <имя-класса> [<комментарии>]
(is-a <список-суперклассов>+)
[<роль-класса>]
[<активность~класса >]
<слот>*
<объявление-обработчика-сообщений>*)
<роль-класса> ::= (role concrete | abstract)
<активность-класса> ::= (pattern-match reactive I non-reactive)
<слот> ::= (slot <имя> <грани>*) |
(single-slot <имя> <грани>*) |
(multislot <имя> <грани>*)
<грань> ::= < значение-по-умолчанию > |
<грань-хранения> |<грань-доступа> |
<грань-распространения> |
<грань-источника> |
<грань-сопоставления-образцов> |
<грань-видимости> |
<грань-создания-аксессоров> |
<грань-переопределения-сообщений> |
<ограничения-атрибутов>
<значение-по-умолчанию> ::=(default ?DERIVE | ?NONE |
<выражение>*) |
(default-dynamic <выражение>*)
<грань-хранения> ::=(storage local | shared)
<грань-доступа> ::=(access read-write | read-only | initialize-only)
<грань-распространения> ::=(propagation inherit | no-inherit)
<грань-источника> ::=(source exclusive | composite)
<грань-сопоставления-образцов> ::=(pattern-match reactive | non-reactive)
<грань-видимости> ::=(visibility private | public)
<грань-создания-аксессоров> ::=(create-accessor ?NONE | read | write| read – write)
<грань-переопределения-сообщений>::=(override-message ?DEFAULT | <имя-сообщения>)
<объявление-обработчика-сообщений>::=(message-handler <имя-обработчика> [тип-обработчика])
< тип-обработчика > ::=primary | around | before | after

Определение П 1.32. Синтаксис конструктора defmessage-handler

defmessage-handler <имя-класса>
<имя-сообщения>
[<тип-обработчика>]
[<комментарии>]
(<обязательные-параметры>
[<групповой-параметр>])
<действия>)
< тип-обработчика > ::= around | before | primary | after
<обязательный-параметр>::=<простое-значение>
<групповой-параметр>::=<составное-значение>

Определение П 1.33. Синтаксис системных обработчиков класса USER

(defmessage-handler USER init primary ())
(defmessage-handler USER delete primary ())

```
(defmessage-handler USER print primary ( ) )

(defmessage-handler USER direct-modify primary
                        (?slot-override-expressions))
(defmessage-handler USER message-modify primary
                        (?slot-override-expressions)
(defmessage-handler USER direct-duplicate primary
                        (?new-instance-name ?slot-override-expressions))
(defmessage-handler USER message-duplicate primary
                        (?new-instance-name ?slot-override-expressions)
```

Определение П 1.34. Синтаксис конструктора def instances

```
 (definstances     <имя> [active] [<комментарии>]
                        <шаблоны-объектов>)
<шаблон-объекта>   ::= (<определение-объекта>)
```

Определение П 1.35. Синтаксис предопределенного класса и экземпляра этого класса

```
 (defclass     INITIAL-OBJECT
                  (is-a USER)
                  (role concrete)
                  (pattern-match reactive))
(definstances initial-object
                  (initial-object of INITIAL-OBJECT))
```

Определение П 1.36. Синтаксис шаблона набора объектов

```
<шаблон-набора-объектов>       : : =  (<члены-шаблона-наборов-объекта>)
<член-шаблона-наборов-объекта>   ::=(<переменная-набора-объектов> <ограничения-классов>)
<переменная-набора-объектов>    ::= <простая-переменная>
<ограничения-классов>        ::= <имена-классов>
```

Определение П 1.37. Синтаксис запроса

```
<запрос>   : : =  (<логическое-выражение>)
```

Определение П1.38. Синтаксис обращения к переменным

```
<переменная-набора-объектов>:<имя-слота>
```

Определение П 1.39. Синтаксис определения действия

```
<действие>   :: = <функция>
```

Определение П1.40. Синтаксис функции any-instancep

```
 (any-instancep  <шаблон-набора-объектов>  <запрос>)
```

Определение П1.41. Синтаксис функции find-instance

```
 (find-instance <шаблон-набора-объектов> <запрос>)
```

Определение П 1.42. Синтаксис функции find-all-instance

(find-all-instance <шаблон-набора-объектов> <запрос>)

Определение П 1.43. Синтаксис функции do-for-instance

(do-for-instance <шаблон-набора-объектов> <запрос> <действие>)

Определение П 1.44. Синтаксис функции do-for-all-instance

(do-for-all-instanco <шаблон-набора-объектов> <запрос> <действие>)

Определение П 1.45. Синтаксис функции delayed-do-for-all-inatance

(delayed-do-for-all-instance
<шаблон-набора-объектов> <запрос> <действие>)

Определение П 1.46. Синтаксис конструктора defmodule

(defmodule <имя-модуля>
[<комментарии>] <спецификации-импорта-экспорта>*)

<спецификация-импорта-экспорта> ::=
(export <элемент-спецификации>) |
(import <имя-модуля> <элемент-спецификации>)
<элемент-спецификации> ::= ?ALL |
?NONE |
<конструктор> ?ALL |
<конструктор> ?NONE |
<конструктор> <имя-конструктора>
<конструкция> ::= deftemplate | defclass |
defglobal | deffunction |
defgeneric

Определение П 1.47. Предопределенный конструктор модуля MAIN

(defmodule MAIN)

Определение П 1.48. Синтаксис атрибутов ограничений

<атрибуты-ограничений> ::= <атрибут-типа> |
<константный-атрибут> |
<атрибут-диапазона> |
<атрибут-мощности>

Определение П1.49. Синтаксис атрибута ограничения типа

<атрибут-типа> ::= (type <спецификация-типа>)

<спецификация-типа> ::= <допустимые-типы> | ?VARIABLE

<допустимый-тип> ::= SYMBOL | STRING | LEXEME |
INTEGER I FLOAT | NUMBER |
INSTANCE-NAME | INSTANCE-ADDRESS |
INSTANCE | EXTERNAL-ADDRESS | FACT-ADDRESS

Определение П 1.50. Синтаксис константного атрибута

<константный-атрибут> ::=
(allowed-symbols <список-зутЬо1-значений>) |
(allowed-strings <список-string-значений>) |
(allowed-lexemes <список-1exete-значений> |
(allowed-integers <список-integer-значений>) |
(allowed-floats <список-f1oat-значений>) |
(allowed-numbers <список-number-значений>) |
(allowed-instance-names <список-instance-значений>)
(allowed-values <список-значений>)
<список-symbo1-значений> ::= <symbol>+ | ?VARIABLE
<список-string-значений> ::= <string>+ | ?VARIABLE
<список-1exeme-значений> ::= <lexeme>+ | ?VARIABLE
<список-integer-значений> ::=<integer>+ I ?VARIABLE
<список-float -значений> ::=<float>+ | ?VARIABLE
<список-number-значений> ::=<number>+ | ?VARIABLE
<cписок-instance-значений> ::=<instance-name>+ | ?VARIABLE
<список-значений> ::=<constant>+ | ?VARIABLE

Определение П 1.51. Синтаксис атрибута диапазона

<атрибут-диалазона> ::= (range <граница-диапазона>
<граница-диапазона>)
<граница-диапазона> ::= <число> | ?VARIABLE

Определение П 1.52. Синтаксис атрибута мощности

<атрибут-мощности> ::= (cardinality <граница-мощности>
<граница-мощности>)
<граница-мощности> ::= <целое-значение> | ?VARIABLE

ПРИЛОЖЕНИЕ 2 Список основных сообщений об ошибках системы CLIPS

Сообщения об ошибках, возникающие при работе с системой CLIPS, делятся на два типа. Первому типу принадлежат ошибки, возникающие при создании различных конструкторов, ко второму относятся ошибки, возникающие при работе этих конструкторов. Данное приложение содержит список наиболее важных сообщений об ошибках сообщений системы CLIPS.

Каждое сообщение об ошибке имеет уникальный идентификатор, заключенный в квадратные скобки, после которого следует описание возникшей ситуации.

[AGENDAl] Salience value must be an integer value.

[AGENDA2] Salience value out of range <min> to <max>

[AGENDA3] This error occurred while evaluating the salience [for rule <name>]

[ANALYSISl] Duplicate pattern-address <variable name> found in CE <CE number>.

[ANALYSIS2] Pattern-address <variable name> used in CE #2 was previously bound within a pattern CE.

[ANALYSIS3] Variable <variable name> is used as both a single and multifield variable.

[ANALYSIS4] Variable <variable name> [found in the expression <expression>] was referenced in CE <CE number> <field or slot identifier> before being defined

[ARGACCES1] Function <name> expected at least <minimum> and no more than <maximum> argument(s)

[ARGACCES2] Function <function-name> was unable to open file <file-name>

[ARGACCES3] Function -enamel> received a request from function <name2> for argument #<number> which is non-existent

[ARGACCES4] Function <name> expected exactly <number> argument(s)

[ARGACCES4] Function <name> expected at least <number> argument(s)

[ARGACCES4] Function <name> expected no more than <number> argument(s)

[ARGACCES5] Function <name> expected argument #<number> to be of type <data-type>

[ARGACCES6] Function <namel> received a request from function <name2> for argument #<number> which is not of type <data-type>

[BLOAD1] Cannot load <construct type> construct with binary load in effect.

[BLOAD2] File <file-name> is not a binary construct file

[BLOAD3] File <file-name> is an incompatible binary construct file

[BLOAD4] The CLIPS environment could not be cleared. Binary load cannot continue.

[BLOAD5] Some constructs are still in use by the current binary image:

<construct-name 1>

<construct-name 2>

…

<construct-name N>

Binary <operation> cannot continue.

[BLOAD6] The following undefined functions are referenced by this binary image:
<function-name 1>
<function-name 2>
…

<function-name N>

[BSAVE1] Cannot perform a binary save while a binary load is in effect.

[CLASSEXM1] Inherited slot <slot-name> from class <slot-name> is not valid for function slot-publicp

[CLASSFUN1] Unable to find class <class name> in function <function name>.

[CLASSFUN2] Maximum number of simultaneous class hierarchy traversals exceeded <number>.

[CLASSPSR1] An abstract class cannot be reactive.

[CLASSPSR2] Cannot redefine a predefined system class.

[CLASSPSR3] <name> class cannot be redefined while outstanding references to it still exist.

[CLASSPSR4] Class <attribute> already declared.

[CLSLTPSR1] Duplicate slots not allowed.

[CLSLTPSR2] <name> facet already specified.

[CLSLTPSR3] Cardinality facet can only be used with multifield slots

[CLSLTPSR4] read-only slots must have a default value

[CLSLTPSR5] read-only slots cannot have a write accessor

[CLSLTPSR6] no-inherit slots cannot also be public

[COMMLINE1] Expected a ' (' , constant, or global variable

[COMMLINE2] Expected a command.

[CONSCOMP1] Invalid file name <fileName> contains ' . '

[CONSTRCT1] Some constructs are still in use. Clear cannot continue.

[CSTRCPSR1] Expected the beginning of a construct.

[CSTRCPSR2] Missing name for <construct-type> construct

[CSTRCPSR3] Cannot define <construct-type> <construct-name> because of an import/export conflict.

[CSTRCPSR3] Cannot define defmodule <defmodule-name> because of an import/export conflict cause by the <construct-type> <construct-name>.

[CSTRCPSR4] Cannot redefine <construct-type> <construct-name> while it is in use.

[CSTRNCHK1] Message Varies

[CSTRNPSR1] The <first attribute name> attribute conflicts with the <second attribute name> attribute.

[CSTRNPSR2] Minimum <attribute> value must be less than or equal to the maximum <attribute> value.

[CSTRNPSR3] The <first attribute name> attribute cannot be used in conjunction with the <second attribute name> attribute.

[CSTRNPSR4] Value does not match the expected type for the <attribute name> attribute.

[CSTRNPSR5] The cardinality attribute can only be used with multifield slots.

[DEFAULT1] The default value for a single field slot must be a single field value

[DFFNXPSR1] Deffunctions are not allowed to replace constructs.

[DFFNXPSR2] Deffunctions are not allowed to replace external functions.

[DFFNXPSR3] Deffunctions are not allowed to replace generic functions.

[DFFNXPSR4] Deffunction <name> may not be redefined while it is executing.

[DFFNXPSR5] Defgeneric <name> imported from module <module name> conflicts with this deffunction.

[DRIVE1] This error occurred in the join network Problem resides in join #<pattern-number> in rule(s): <problem-rules>+

[EMATHFUN1] Domain error for <function-name> function

[EMATHFUN2] Argument overflow for <function-name> function

[EMATHFUN3] Singularity at asymptote in <function-name> function

[EVALUATN1] Variable <name> *is* unbound

[EVALUATN2] No function, generic function or deffunction of name <name> exists for external call.

[EXPRNPSR1] A function name must be a symbol

[EXPRNPSR2] Expected a constant, variable, or expression

[EXPRNPSR3] Missing function declaration for <name>

[EXPRNPSR4] $ Sequence operator not a valid argument for <name>.

[FACTMCH1] This error occurred in the pattern network Currently active fact: <newly assert fact> Problem resides in slot <slot name> Of pattern #<pattern-number> in rule(s): <problem-rules>+

[FACTMNGR1] Facts may not be retracted during pattern-matching

[FACTMNGR2] Facts may not be retracted during pattern-matching

[FACTRHS1] Template <name> does not exist for assert.

[GENRCCOM1] No such generic function <name> in function undefmethod.

[GENRCCOM2] Expected a valid method index in function undefmethod.

[GENRCCOM3] Incomplete method specification for deletion.

[GENRCCOM4] Cannot remove implicit system function method for generic function <name>.

[GENRCEXE1] No applicable methods for <name>.

[GENRCEXE2] Shadowed methods not applicable in current context.

[GENRCEXE3] Unable to determine class of <value> in generic function <name>.

[GENRCEXE4] Generic function <name> method |<index> is not applicable to the given arguments.

[GENRCFUN1] Defgeneric <name> cannot be modified while one of its methods is executing.

[GENRCFUN2] Unable to find method <name> tt<index> in function <name>.

[GENRCFUN3] Unable to find generic function <name> in function <name>.

[GENRCPSR1] Expected ')' to complete defgeneric.

[GENRCPSR2] New method #<indexl> would be indistinguishable from method I<index2>.

[GENRCPSR3] Defgenerics are not allowed to replace constructs.

[GENRCPSR4] Deffunction <name> imported from module <module name> conflicts with this defgeneric.

[GENRCPSR5] Generic functions are not allowed to replace deffunctions.

[GENRCPSR6] Method index out of range.

[GENRCPSR7] Expected a '(' to begin method parameter restrictions.

[GENRCPSR8] Expected a variable for parameter specification.

[GENRCPSR9] Expected a variable or '(' for parameter specification.

[GENRCPSR10] Query must be last in parameter restriction.

[GENRCPSRll] Duplicate classes/types not allowed in parameter restriction.

[GENRCPSR12] Binds are not allowed in query expressions.

[GENRCPSR13] Expected a valid class/type name or query.

[GENRCPSR.14] Unknown class/type in method.

[GENRCPSR15] <name> class is redundant.

[GENRCPSR16] The system function <name> cannot be overloaded.

[GENRCPSR17] Cannot replace the implicit system method #<integer>.

[GLOBLDEF1] Global variable <variable name> is unbound.

[GLOBLPSR1] Global variable <variable name> was referenced, but is not defined.

[INCRRSET1] The incremental reset behavior cannot be changed with rules loaded.

[INHERPSR1] A class may not have itself as a superclass.

[INHERPSR2] A class may inherit from a superclass only once.

[INHERPSR3] A class must be defined after all its superclasses.

[INHERPSR4] Must have at least one superclass.

[INHERPSR5] Partial precedence list formed: <classa> <classb> … <classc> Precedence loop in superclasses: <classl> <class2> … <classn> <classl>

[INHERPSR6] A user-defined class cannot be a subclass of <name>.

[INSCOM1] Undefined type in function <name>.

[INSFILE1] Function <function-name> could not completely process file <name>.

[INSFILE2] <file-name> file is not a binary instances file.

[INSFILE3] <file-name> file is not a compatible binary instances file.

[INSFILE4] Function bload-instances unable to load instance <instance-name>.

[INSFUN1] Expected a valid instance in function <name>.

[INSFUN2] No such instance <name> in function <name>.

[INSFUN3] No such slot <name> in function <name>.

[INSFUN4] Invalid instance-address in function <name>.

[INSFUN5] Cannot modify reactive instance slots while pattern-matching is in process.

[INSFUN6] Unable to pattern-match on shared slot <name> in class <name>.

[INSFUN7] <multifield-value> illegal for single-field slot <name> of instance <name> found in <function-call or message-handler>.

[INSFUN8] Void function illegal value for slot <name> of instance <name> found in <function-call or message-handler>.

[INSMNGR1] Expected a valid name for new instance.

[INSMNGR2] Expected a valid class name for new instance.

[INSMNGR3] Cannot create instances of abstract class <name>.

[INSMGNR4] The instance <name> has a slot-value which depends on the instance definition.

[INSMNGR5] Unable to delete old instance <name>.

[INSMNGR6] Cannot delete instance <name> during initialization.

[INSMNGR7] Instance <name> is already being initialized.

[INSMNGR8] An error occurred during the initialization of instance <name>.

[INSMNGR9] Expected a valid slot name for slot-override.

[INSMNGR10] Cannot create instances of reactive classes while pattern-matching is in process.

[INSMNGRll] Invalid module specifier in new instance name.

[INSMNGR12] Cannot delete instances of reactive classes while pattern-matching is in process.

[INSMNGR13] Slot <slot-name> does not exist in instance <instance-name>.

[INSMNGR14] Override required for slot <slot-name> in instance <instance-name>.

[INSMNGR15] init-slots not valid in this context.

[INSMODDP1] Direct/message-modify message valid only in modify-instance.

[INSMODDP2] Direct/message-duplicate message valid only in duplicate-instance.

[INSMODDP3] Instance copy must have a different name in duplicate-instance.

[INSMULT1] Function <name> cannot be used on single-field slot <name> in instance <name>.

[INSQYPSR1] Duplicate instance-set member variable name in function <name>.

[INSQYPSR2] Binds are not allowed in instance-set query in function <name>.

[INSQYPSR3] Cannot rebind instance-set member variable <name> in function <name>.

[IOFUN1] Illegal logical name used for <function name> function.

[IOFUN2J Logical name <logical name> already in use.

[MEMORY1] Out of memory

[MEMORY2] Release error in genfree

[MEMORY3] Unable to allocate memory block > 32K

[MISCFUN1] expand? must be used in the argument list of a function call.

[MODULDEF1] Illegal use of the module specifier. '

[MODULPSR1] Module <module name> does not export any constructs.

[MODULPSR1] Module <module name> does not export any <construct type> constructs.

[MODULPSR1] Module <module name> does not export the <construct type> <construct name>.

[MSGCOMl] Incomplete message-handler specification for deletion.

[MSGCOM2] Unable to find message-handler <name> <type> for class <name> in function <name>.

[MSGCOM3] Unable to delete message-handlers.

[MSGFUN1] No applicable primary message-handlers found for <message>.

[MSGFUN2] Message-handler <name> <type> in class <name> expected exactly/at least <number> argument(s) .

[MSGFUN3] <name> slot in instance <name>: write access denied.

[MSGFUN4] <function> may only be called from within message-handlers.

[MSGFUN5] <function> operates only on instances.

[MSGFUN6] Private slot <slot-name> of class <class-name> cannot be accessed directly by handlers attached to class <class-name>

[MSGFUN7] Unrecognized message-handler type in defmessage-handler.

[MSGFUN8] Unable to delete message-handler(s) from class <name>.

[MSGPASS1] Shadowed message-handlers not applicable in current context.

[MSGPASS2] No such instance <name> in function <name>.

[MSGPASS3] Static reference to slot <narne> of class <name> does not apply to <instance-name> of <class-name>.

[MSGPSR1] A class must be defined before its message-handlers.

[MSGPSR2] Cannot (re)define message-handlers during execution of other message-handlers for the same class.

[MSGPSR3] System message-handlers may not be modified. [MSGPSR4] Illegal slot reference in parameter list.

[MSGPSR5] Active instance parameter cannot be changed.

[MSGPSR6] No such slot <name> in class <name> for ?self reference.

[MSGPSR7] Illegal value for ?self reference.

[MSGPSR8] Message-handlers cannot be attached to the class <name>.

[MULTIFUN1] Multifield index <index> out of range l..<end range> in function <name>

[MULTIFUN1] Multifield index range <start>...<end> out of range l..<end range> in function <name>

[MULTIFUN2] Cannot rebind field variable in function progn$.

[OBJRTBLD1] No objects of existing classes can satisfy pattern.

[OBJRTBLD2] No objects of existing classes can satisfy <attribute-name> restriction in object pattern.

[OBJRTBLD3] No objects of existing classes can satisfy pattern #<pattern-num>.

[OBJRTBLD4] Multiple restrictions on attribute <attribute-name> not allowed.

[OBJRTBLD5] Undefined class in object pattern.

[OBJRTMCH1] This error occurred in the object pattern network Currently active instance: <instance-name> Problem resides in slot <slot name> field l<field-index> Of pattern #<pattern-number> in rule(s): <problem-rules>+

[PATTERN1] The symbol <symbol name> has special meaning and may not be used as a <use name>.

[PATTERN2] Single and multifield constraints cannot be mixed in a field constraint

[PRCCODE1] Attempted to call a <construct> which does not exist.

[PRCCODE2] Functions without a return value are illegal as <construct> arguments.

[PRCCODE3] Undefined variable <name> referenced in <where>. [PRCCODE4] Execution halted during the actions of <construct> <name>. [PRCCODE5] Variable <name> unbound [in <construct> <name>].

[PRCCODE6] This error occurred while evaluating arguments for the <construct> <name>.

[PRCCODE7] Duplicate parameter names not allowed.

[PRCCODE8] No parameters allowed after wildcard parameter.

[PRCDRPSR1J Cannot rebind count variable in function loop-for-count.

[PRCDRPSR2] The return function is not valid in this context.

[PRCDRPSR2] The break function is not valid in this context.

[PRCDRPSR3] Duplicate case found in switch function.

[PRNTUTIL1] Unable to find <item> <item-name>

[PRNTUTIL2] Syntax Error: Check appropriate syntax for <item>

[PRNTUTIL3]
*** CLIPS SYSTEM ERROR ***
ID = <error-id>
CLIPS data structures are in an inconsistent or corrupted state.
This error may have occurred from errors in user defined code.

[PRNTUTIL4] Unable to delete <item> <item-name>

[PRNTUTIL5] The <item> has already been parsed.

[PRNTUTIL6] Local variables cannot be accessed by <function or construct>.

[PRNTUTIL7] Attempt to divide by zero in <function-name> function.

[ROUTER1] Logical name <logical_name> was not recognized by any routers

[RULECSTR1] Variable <variable name> .in CE t<integer> slot <slot name> has constraint conflicts which make the pattern unmatchable.

[RULECSTR1] Variable <variable name> in CE #<integer> field #<integer> has constraint conflicts which make the pattern unmatchable.

[RULECSTR1] CE l<integer> slot <slot name> has constraint conflicts which make the pattern unmatchable.

[RULECSTR1] CE tt<integer> field #<integer> has constraint conflicts which make the pattern unmatchable

[RULECSTR2] Previous variable bindings of <variable name> caused the type restrictions for argument #<integer> of the expression <expression> found in CE#<integer> slot <slot name> to be violated.

[RULECSTR3] Previous variable bindings of <variable name> caused the type restrictions for argument #<integer> of the expression <expression> found in the rule's RHS to be violated.

[RULELHS1] The logical CE cannot be used with a not/exists/forall CE.

[RULELHS2] A pattern CE cannot be bound to a pattern-address within a not CE

[RULEPSR1] Logical CEs must be placed first in a rule

[RULEPSR2] Gaps may not exist between logical CEs

STRNGFUN1] Function build does not work in run time modules.

[STRNGFUN1] Function eval does not work in run time modules.

[STRNGFUN2] Some variables could not be accessed by the eval function.

[SYSDEP1] No file found for -f option.

[TEXTPR01] Unable to access help file.

[TEXTPRO2] Unable to load file. Explanatory text>.

[TMPLTDEF1] Invalid slot <slot name> not defined in corresponding deftemplate <deftemplate name>

[TMPLTDEF2] The single field slot <slot name> can only contain a single field value.

[TMPLTFUN1] Fact-indexes can only be used by <command name> as a top level command.

[TMPLTFUN2] Attempted to assert a multifield value into the single field slot <slot name> of deftemplate <deftemplate name>.

[TMPLTRHS1] Slot <slot name> requires a value because of its (default 7NONE) attribute.

ПРИЛОЖЕНИЕ 3 Список основных предупреждений системы CLIPS

Предупреждающие сообщения CLIPS делятся на два типа. К первому типу предупреждающих сообщений относятся потенциально ошибочные ситуации, складывающиеся при выполнении различных конструкций среды. Второй тип представляют собой сообщения, возникающие при не критических ошибках загрузки конструкторов в CLIPS. Данное приложение содержит список наиболее важных предупреждающих сообщений системы.

Каждое сообщение имеет уникальный идентификатор, заключенный в квадратные скобки, после которого следует ключевое слово WARNING и описание возникшей ситуации.

[CONSCOMPl] WARNING: Base file name exceeds 3 characters. This may cause files to be overwritten if file name length is limited on your platform.

[CONSCOMP2] WARNING: Array name <arrayName> exceeds б characters in length. This variable may be indistinguishable from another by the linker.

[CSTRNBIN1] WARNING: Constraints are not saved with a binary image when dynamic constraint checking is disabled.

[CSTRNCMP1] WARNING: Constraints are not saved with
a constructs-to-c image when dynamic constraint checking is disabled

[DFFNXFUN1] WARNING: Deffunction <name> only partially deleted due to usage by other constructs.

[GENRCBIN1] WARNING: COOL not installed! User-defined class in method restriction substituted with OBJECT.

[OBJBIN1] WARNING: Around message-handlers are not supported in this environment.

[OBJBIN1] WARNING: Around message-handlers are not supported in this environment.

ПРИЛОЖЕНИЕ 4 Зарезервированные имена CLIPS

В данном приложении приводится полный список имен системных функций CLIPS. Имена, перечисленные здесь, нельзя использовать в качестве имен пользовательских функций, за исключением случаев перегрузки с помощью родовых функций.

!=
*
**
+
-/
<
<=
<>
=
>
>=
abs
acos
acosh
acot
acoth
acsc
acsch
active-duplicate-instance
active-initialize-instance
active-make-instance
active-message-duplicate-instance
active-message-modify-instance
active-modify-instance
agenda
and
any-instancep
apropos
asec
asech
asin
asinh
assert
assert-string
atan
atanh
batch
batch*
bind
bload
bload-instances
break
browse-classes
bsave
bsave-instances
build
call-next-handler
call-next-method
call-specific-method
class
class-abstractp
class-existp
class-reactivep
class-slots
class-subclasses
class-superclasses
clear
clear-focus-stack
close
conserve-mem
constructs-to-c
cos
cosh
cot
coth
create$
esc
csch
defclass-module
deffacts-module
deffunction-module
defgeneric-module
defglobal-module
definstances-module
defrule-module
deftemplate-module
deg-grad
deg-rad
delayed-do-for-all-instances
delete$
delete-instance
dependencies
dependents
describe-class
direct-mv-delete
direct-mv-insert
direct-mv-replace
div
do-for-all-instances
do-for-instance
dribble-off
dribble-on
duplicate
duplicate-instance
dynamic-get
dynamic-put

edit
eq
eval
evenp
exit
exp
expand$
explode$
fact-existp
fact-index
fact-relation
fact-slot-names
fact-slot-value
facts
fetch
find-all-instances
find-instance
first$
float
floatp
focus
format
gensym
gensym*
get
get-auto-float-dividend
get-current-module
get-defclass-list
get-deffacts-list
get-deffunction-list
get-defgeneric-list
get-defglobal-list
get-definstances-list
get-defmessage-handler-list
get-defmethod-list
get-defmodule-list
get-defrule-list
get-deftemplate-list
get-dynamic-constraint-checking
get-fact-duplication
get-fact-list
get-focus
get-focus-stack
get-function-restrictions
get-incremental-reset
get-method-restrictions
get-reset-globals
get-salience-evaluation
get-sequence-operator-recognition
get-static-constraint-checking
get-strategy
grad-deg
halt
help
help-path
if
implode$
init-slots
initialize-instance
insert$
instance-address
instance-addressp
instance-existp
instance-name
instance-name-to-symbol
instance-namep
instancep
instances
integer
integerp
length
length$
lexemep
list-defclasses
list-deffacts
list-deffunctions
list-defgenerics
list-defglobals
list-definstances
list-defmessage-handlers
list-defmethods
list-defmodules
list-defrules
list-deftemplates
list-focus-stack
list-watch-items
load
load*
load-facts
load-instances
log
loglO
loop-for-count
lowcase
make-instance
matches
max
mem-requests
mem-used
member
member$
message-duplicate-instance
message-handler-existp
message-modify-instance
min
mod
modify
modify-instance
multifieldp
mv-append

str-cat
str-compare
str-explode
str-implode
str-index
str-length
stringp
sub-string
subclassp
subseq$
subset
subsetp
superclassp
switch
sym-cat
symbol- to-instance-name
symbolp
system
tan
tanh
time
toss
type
undefclass
undef facts
undef function
undef generic
undef global
undef instances
undef message-handler
undefmethod
undef rule
undef template
unmake-instance
unwatch
upcase
watch
while
wordp

ПРИЛОЖЕНИЕ 5 Глоссарий

Abstraction (Абстракция)
Способ представления данных на определенном уровне некоторой конкретной предметной области.

Action (Действие)
Функция, выполняемая произвольным конструктором (например, правой частью правила), которая обычно не возвращает никакого значения, но выполняет некоторое полезное действие (например, вывод на печать).

Activation (Активация)
Процесс помещения правила, все условия которого удовлетворены, но которое еще не было выполнено, в план решения задачи.

Active instance (Активный объект)
Текущий объект. Фактически понятие активного объекта было введено для реализации возможности объекта посылки сообщений самому себе.

Agenda (План решения задачи)
Список активированных правил с данными, вызвавшими эту активацию, готовых в настоящий момент к запуску. Правила сортируются согласно заданному приоритету и текущей выбранной стратегии разрешения конфликтов. Правило, находящееся на вершине списка, будет запущено первым.

Antecedent (Предпосылка)
Левая часть правила.

Bind (Связывание)
Действие по сохранению (присваиванию) значения в переменной.

Class (Класс)
Шаблон для описания общих свойств (слоты) и поведения (обработчики сообщений) группы объектов, называемых экземплярами данного класса.

Class precedence list (Список предшествования классов)
Линейное упорядочение суперклассов, которое описывает порядок наследованию свойств и поведения.

Command (Команда)
Функция, которая вводится пользователем в среду при помощи клавиатуры. Обычно такая функция не возвращает никакого значения.

Command prompt (Командная строка)
Подсказка интерактивной среды CLIPS, которая говорит о том, что CLIPS готов к работе.

Condition (Условие)
Логический элемент.

Conditional element (Условный элемент)
Ограничение, определяемое в левой части правила, которое должно быть соблюдено для активации правила.

Conflict resolution strategy (Стратегия разрешения конфликтов)
Метод для определения порядка, в котором правила должны запускаться, среди правил с одинаковым приоритетом. В CLIPS существует семь стратегий разрешения конфликтов: *стратегия глубины* (depth strategy), *стратегия ширины* (breadth strategy), *стратегия упрощения* (simplicity strategy), *стратегия усложнения* (complexity strategy), *LEX* (LEX strategy), MEA (iMEA strategy) и *случайная стратегия* (random strategy).

Consequent (Следствие)
Правая часть правила.

Constant (Константа)
Выражение с не изменяющимся значением.

Constraint (Ограничение)
Дополнительное условие, которое налагается на значение поля факта или объекта.

Construct (Конструктор)
Элемент CLIPS, используемый для добавления в базу знаний новых элементов.

Current focus (Текущий фокус)
Модуль, находящийся на вершине стека фокусов.

Current module (Текущий модуль)
Модуль, к которому добавляются конструкции, не имеющие спецификатора модуля, используемый по умолчанию для команд, которые поддерживают использование произвольного параметра в имени модуля.

Daemon (Демон)
Обработчик сообщений, который выполняется неявно всякий раз, когда над объектом производится некоторое действие, например такое, как инициализация, удаление или изменение в слота.

DeiTunction (Функция, определенная пользователем)
Функция, созданная пользователем в среде CLIPS с помощью соответствующего конструктора.

Deftemplate fact (Шаблон)
Определяет набор именованных слотов определенного типа.

Deftemplate pattern (Образец шаблона)
Ограничения, накладываемые на факты, принадлежащие некоторому шаблону.

Delimiter (Разделитель)
Знак, который указывает на конец элемента одного из примитивных типов данных. Обычно в качестве разделителей используются следующие знаки: любой непечатаемый знак ASCII (включая пробелы, табуляцию, возвраты каретки и переводы строки), двойные кавычки, открывающая и закрывающая круглые скобки "(" и ")", амперсанд "&", вертикальная черта "", меньше чем "<", точка с запятой ";", и тильда "~".

Dynamic binding (Динамическое связывание)
Связывание обработчика сообщения с объектом во время выполнения программы.

Encapsulation (Инкапсуляция)
Скрытие деталей реализации класса.

External-address (Внешний адрес)
Адрес внешней структуры данных, возвращенной функцией (написанной на языке типа C или Ada), интегрированной с CLIPS.

External function (Внешняя функция)
Функция, написанная на внешнем языке (например, C) и подключенная к среде CLIPS.

Facet (Грань)
Компонент, определяющий свойства слота класса.

Fact (Факт)
Единица представления данных в экспертной системе.

Fact-address (Адрес факта)
Указатель на факт, полученный в левой части правила.

Fact-identifier (Идентификатор факта)
Сокращенное описание факта. Состоит из символа "f", следующей за ним черты и индекса факта.

Fact-index (Индекс факта)
Уникальный целочисленный индекс, используемый для идентификации факта.

Fact-list (Список фактов)
Набор фактов, присутствующий в рабочей памяти системы.

Field (Поле)
Место для хранения некоторого значения.

Fire (Запуск правила)
Процедура выполнения правила, все предпосылки которого удовлетворены.

Float (Вещественное значение)
Вещественное значение, заданное в экспоненциальной форме или с помощью числа фиксированной точкой.

Function (Функция)
Участок исполняемого программного кода, идентифицируемый при помощи имени.

Generic dispatch (Родовое связывание)
Процесс определения конкретного метода родовой функции, применимого к заданному набору аргументов.

Generic function (Родовая функция)
Функция, написанная в CLIPS, позволяющая выполнять различные действия, в зависимости от числа и типа заданных аргументов.

Inference engine (Механизм логического вывода)
Механизм, поддерживаемый CLIPS, который автоматически производит сопоставление текущего состояния списка фактов с существующими в системе правилами.

Inheritance (Наследование)
Процесс приобретения подклассом свойств и поведения суперкласса.

Instance (Объект)
Экземпляр некоторого класса.

Instance-address (Адрес образца)
Адрес объекта класса, созданного пользователем.

Instance-name (Имя образца)
Символьный идентификатор объекта.

Instance-set (Набор объектов)
Упорядоченный набор объектов, созданный в результате выполнения некоторого запроса.

Instance-set distributed action (Действие над набором объектов)
Определяемое пользователем действие, которое выполняется над каждым членом заданного набора объектов.

Instance-set query (Запрос)
Определяемое пользователем логическое выражение, используемое для формирования набора объектов.

Integer (Целое)
Целочисленное значение.

LHS (Левая часть правила)
Набор условий, которые должны быть удовлетворены для выполнения действий, заданных в правой части правила.

Logical name (Логическое имя)
Символическое имя устройства ввода/вывода.

Message (Сообщение)
Механизм, позволяющий управление объектом.

Message dispatch (Связывание сообщений)
Процесс, посредством которого для сообщения выбирается и запускается соответствующий обработчик сообщения.

Message-handler (Обработчик сообщений)
Участок кода, активизирующийся при получении объектом соответствующего сообщения.

Message-handler precedence (Приоритет обработчиков сообщений)
Свойство, позволяющее выбирать порядок выполнения обработчиков сообщения.

Method (Метод)

Один из вариантов поведения родовой функции при получении соответствующего набора сообщений.

Method index (Индекс метода)
Целочисленный идентификатор метода.

Method precedence (Приоритет методов)
Свойство, использующее конкретный метод некоторой группы, способной обработать заданный набор аргументов.

Module (Модуль)
Рабочее пространство, позволяющее группировать конструкторы базы знаний и управлять доступом к ним.

Module specifier (Спецификатор модуля)
Способ идентификации модуля в определении различных конструкторов системы.

Multifield value (Составная величина)
Последовательность неименованных данных.

Non- ordered fact (Неупорядоченный факт)
Шаблонный факт.

Number (Число)
Целое число или число с плавающей точкой.

Ordered fact (Упорядоченный факт)
Факт, состоящий из последовательности неименованных полей.

Overload (Перегрузка)
Процесс, в результате которого родовая функция позволяет выполнять различные действия в зависимости от числа и типа аргументов.

Pattern (Образец)
Условный элемент в левой части правила, который используется для сопоставления с фактами из списка фактов.

Pattern entity (Сопоставленный образец)
Элемент, подошедший определенному образцу, заданному в левой части правила.

Pattern-matching (Сопоставление образцов)
Процесс сопоставления фактов или объектов с образцами, заданными в левой части правил.

Polymorphism (Полиморфизм)
Способность разных объектов реагировать на одно и то же сообщение различным образом.

RHS (Правая часть правила)
Действия, выполняемые правилом при его запуске.

Rule (Правило)
Набор условий и действий, выполняемых в случае, если все условия удовлетворены.

Shadowed message-handler (Неявный обработчик сообщений)
Обработчик сообщения, который для выполнения должен быть явно вызван другим обработчиком сообщений.

Shadowed method (Неявный метод)
Метод, который для выполнения должен быть явно вызван другим методом.

Single-field value (Простая величина)
Один из примитивных типов данных: символ, строка, целое число или число с плавающей точкой, внешний адрес, имя или адрес.

Slot (Слот)
Именованная простая или составная величина, содержащаяся в некотором шаблоне или классе.

Slot-accessor (Акцессор, аксессор)
Средство доступа к слоту, неявный обработчик сообщений, который предоставляет доступ к слоту для чтения и записи данных.

Subclass (Подкласс)

Если первый класс является наследником второго класса, то он является и его подклассом.

Superclass (Суперкласс)

Если первый класс является наследником второго класса, то второй класс, в свою очередь, является суперклассом первого.

Value (Значение)

Простое или составное значение.