Библиотека
ГУМАНИТАРНОГО
УНИВЕРСИТЕТА

Л. В. Путькина Т. Г. Пискунова

ИНТЕЛЛЕКТУАЛЬНЫЕ ИНФОРМАЦИОННЫЕ СИСТЕМЫ

УЧЕБНОЕ ПОСОБИЕ

Библиотека
ГУМАНИТАРНОГО
УНИВЕРСИТЕТА
Выпуск 37

Л. В. ПУТЬКИНА, Т. Г. ПИСКУНОВА

ИНТЕЛЛЕКТУАЛЬНЫЕ ИНФОРМАЦИОННЫЕ СИСТЕМЫ

Допущено Учебно-методическим объединением
по образованию в области прикладной информатики
в качестве учебного пособия для студентов высших
учебных заведений, обучающихся по специальности
«Прикладная информатика (по областям)»
и другим специальностям

САНКТ-ПЕТЕРБУРГ
2008

ББК 32.97
П90

Ответственный за выпуск

Т. Г. Пискунова, доцент кафедры информатики
Санкт-Петербургского Гуманитарного университета профсоюзов,
кандидат педагогических наук

Рецензенты:

М. И. Барабанова, заместитель декана факультета РИТММ
Санкт-Петербургского государственного университета экономики
и финансов, кандидат экономических наук, доцент;
Н. В. Лашманова, заведующая кафедрой управления СПбГУП,
доктор технических наук, профессор

Рекомендовано к публикации
редакционно-издательским советом СПбГУП,
протокол № 2 от 16.10.07

Путькина Л. В., Пискунова Т. Г.

П90 Интеллектуальные информационные системы: учеб. пособие. — СПб.: Изд-во СПбГУП, 2008. — 228 с. (Б-ка гуманитарного университета; Вып. 37).

ISBN 978-5-7621-0425-8

Книга освещает современное состояние интеллектуальных информационных систем, применяемых в экономике и социокультурной сфере.

В учебном пособии рассматриваются практические примеры и их реализация на языках программирования Бейсик, Лисп и Пролог. Книга содержит введение в теорию искусственного интеллекта, средства решения задач искусственного интеллекта и язык теории предикатов, а также общие сведения об экспертных системах.

Издание предназначено для студентов, обучающихся по специальности «Прикладная информатика (по областям)», для аспирантов, преподавателей и специалистов в области искусственного интеллекта.

ББК 32.97

ISBN 978-5-7621-0425-8

ВВЕДЕНИЕ

Данное учебное пособие по дисциплине «Интеллектуальные информационные системы» предназначено для студентов, обучающихся по специальности 080801.65 «Прикладная информатика (по областям)», для аспирантов, преподавателей и для специалистов в области искусственного интеллекта.

Учебное пособие содержит шесть разделов, в которые включены примеры задач и их реализация на языках программирования БЕЙСИК, ЛИСП и ПРОЛОГ.

В разделе I дается введение в теорию искусственного интеллекта. В разделе II описаны средства решения задач искусственного интеллекта. В разделе III представлены общие сведения об экспертных системах. В разделе IV описан язык теории предикатов, предназначенный для описания основных свойств предметной области. В разделах V и VI предлагаются методы поиска, применяемые для рассуждения, алгоритмы и структуры данных, используемые для реализации этого поиска.

Изучение основ языка логического программирования ПРОЛОГ предлагается с помощью выполнения четырех лабораторных работ. Язык ПРОЛОГ может использоваться для построения экспертных систем, настраиваемых баз данных, естественно-ориентированных интерфейсов и интеллектуальных систем управления.

Основная цель этих работ — на основе понятия предикатов получить навыки построения программ, основанных на логике и имеющих возможность использования естественного языка. Также необходимо показать возможности применения таких программ для создания баз знаний и их обработки.

Тематика лабораторных работ на языке программирования ЛИСП ориентирована на приложения в области искусственного интеллекта.

Первая лабораторная работа предполагает усвоение технологии написания ЛИСП-программ с использованием списков, методику их тестирования и отладки.

Целью второй лабораторной работы является изучение методов и алгоритмов сопоставления с образцом, их реализация на языке ЛИСП.

Центральное место занимает третья лабораторная работа, посвященная построению Пролог-интерпретатора на языке ЛИСП. Приводится описание базового варианта Пролог-интерпретатора, на основе которого предлагается разработать более развитые версии.

Для каждой лабораторной работы приведены варианты индивидуальных заданий.

Выполнение лабораторных работ ориентировано на системы Xlisp и Golden Common Lisp (стандарт языка Common Lisp) на ПЭВМ IBM PC AT/XT.

В приложении к лабораторным работам приведены теоретические основы языка программирования ЛИСП.

I. ВВЕДЕНИЕ В СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Понятие искусственного интеллекта

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

Традиционное программирование

Компьютерные программы, как правило, предназначены для решения строго определенных задач. Приспособить программу к решению новых задач можно, внеся в нее изменения, а для этого ее нужно всю внимательно просмотреть. Но подробный просмотр занимает слишком много времени, а при внесении изменений в программе могут возникнуть дополнительные ошибки.

Искусственный интеллект, как следует из самого названия, придает компьютеру черты разума. Методы ИИ упрощают объединение программ и дают возможность заложить в систему искусственного интеллекта способность к самообучению и накоплению новой, полезной в дальнейшем информации. Человек может накапливать знания, не изменяя способ мышления и не забывая уже известных фактов. Система ИИ работает почти точно так же. После прочтения данной книги читателю станет ясно, что модифицировать программы ИИ значительно проще, чем традиционные программы.

Влияние ИИ на программирование

Методы ИИ предполагают высокую степень независимости отдельных частей программы, каждая из которых реализует определенный шаг решения одной или нескольких задач. Рассмотрим это более подробно. Независимые части программы можно сравнить с отдельными блоками информации в человеческой памяти. Выбирая нужную информацию, человеческий мозг автоматически подключает только относящиеся к делу факты, не перебирая все доступные ему знания.

Одну и ту же задачу можно запрограммировать, используя либо традиционные методы, либо методы искусственного интеллекта. Применение методов искусственного интеллекта позволяет существенно упростить и ускорить разработку программ. В программах обоих типов отдельные части выполняют строго определенные действия, однако программы ИИ обладают особым свойством, похожим на характерное свойство человеческого интеллекта: изменение любой, даже небольшой части информации не влияет на структуру всей программы. Такая гибкость придаст процессу программирования большую эффективность, дает возможность создавать программы, умеющие «понимать», то есть обладающие чертами разума.

Человеческое мышление

Искусственный интеллект опирается на знания о процессе человеческого мышления. Конечно, точно не известно, как работает человеческий мозг, ученые только начинают постигать сложный механизм интеллекта. Однако для разработки программ искусственного интеллекта имеющихся знаний вполне достаточно.

Цели

В основе человеческой деятельности лежит мышление. Когда утром звонит будильник, мозг человека дает команду руке выключить его. Это не автоматическая реакция, решение конкретной задачи требует определенного ответа мозга. *Целью* называется конечный результат, на который направлены мыслительные процессы человека. Как только цель (выключить будильник) до-

стигнута, перед человеческим мозгом сразу встают новые цели, например пойти в ванную, почистить зубы, одеться, позавтракать, выйти на автобусную остановку и т. д. Осуществление всех этих целей приводит к достижению главной цели — вовремя попасть на работу. Мысли, ведущие к конечному результату, не случайны, а строго обоснованы. Каждый шаг на пути к главной цели имеет свою локальную цель. Мозг всегда сосредоточен на цели независимо от того, выполняет ли человек простую физическую работу или решает сложную интеллектуальную задачу. Цель заставляет человека думать. Целью может быть, например, следующее:

1. Определить кратчайший путь между Нью-Йорком и Бостоном.
2. Выбрать вино, больше всего подходящее к определенной рыбе.
3. Научиться завязывать шнурки у ботинок.
4. Найти способ оценки успехов ребенка в арифметике.

При проектировании систем ИИ всегда следует помнить о цели, для достижения которой они предназначены. Запомните — люди делают что-либо не потому, что думают, а думают, потому что должны что-либо сделать. Теперь, когда в общих чертах понятен ход человеческих мыслей при достижении цели, рассмотрим, каким образом человек решает множество повседневных задач.

Факты и правила

Человеческий мозг — это огромное хранилище знаний. Человеку свойственно приобретать новые знания и применять их к возникающим ситуациям. В общем, интеллект можно представить как совокупность фактов и способов их применения для достижения цели. Отчасти цели достигаются с помощью правил использования всех известных фактов. Приведем несколько примеров фактов и правил их использования.

Пример 1

Факт 1. Зажженная плита горячая. Правило 1. ЕСЛИ положить руку на зажженную плиту, ТО можно обжечься.

Пример 2

Факт 2. В час пик на улице много машин.

Правило 2. ЕСЛИ попытаться в час пик перейти шоссе, ТО можно попасть под машину.

Пример 3

Факт 3а. Тихие темные улицы опасны.

Факт 3б. Пожилые люди обычно не совершают дерзких преступлений.

Факт 3в. Полиция защищает людей от преступников.

Правило 3а. ЕСЛИ на тихой темной улице встретится пожилой человек, ТО можно не очень беспокоиться.

Правило 3б. ЕСЛИ на тихой темной улице вы видите полицейского, ТО можно чувствовать себя в безопасности.

Пример 4

Факт 4. При сложении двух чисел, сумма которых больше 9, нужен перенос.

Правило 4. ЕСЛИ складывается столбец чисел, сумма которых больше 9, ТО для правильного выполнения сложения нужно обратиться к факту 4.

Заметим, что в приведенных примерах все правила выражены условным отношением ЕСЛИ–ТО, то есть ЕСЛИ выполняется некоторое условие, ТО последует определенное действие или какая-либо другая реакция, факты и правила могут быть разной сложности. Обычно при достижении цели люди связывают сложные совокупности и фактов и правил.

Упрощение

Когда человеческий мозг приступает к решению даже самой простой задачи, для выбора нужных действий в его распоряжении имеется огромный объем информации. Например, направляясь на работу, человек выходит из дома и идет на угол улицы. Пока он выбирает момент для перехода улицы, в его мозг поступает самая разнородная информация. Прежде чем пересечь улицу, человек анализирует скорость и объем движения, расстояние до противоположного тротуара, сигналы светофора на перекрестке. Одновременно его мозг обрабатывает впечатления, не име-

ющие прямого отношения к переходу улицы, например, погодные условия, цвет и модели проезжающих машин, породу и высоту деревьев, растущих у дороги, вид расположенных неподалеку зданий. Несомненно, человек также думает о месте, куда он идет, о том, как скоро ему надо там быть, кого он может встретить и т. д.

Если бы человек, прежде чем шагнуть на проезжую часть, анализировал все факты, имеющие прямое, косвенное или вообще не имеющие никакого отношения к его цели перейти улицу, он простоял бы на тротуаре несколько лет. Каким же образом человеческий мозг из огромного разнообразия фактов и правил быстро выбирает подмножество, подходящее только к конкретной ситуации? Дело в том, что в мозгу существует сложная система, руководящая выбором правильной реакции на конкретную ситуацию. Такой выбор называется *упрощением.* Механизм упрощения блокирует мысли, не имеющие отношения к решаемой в данный момент задаче. Точно так же, как удаление у дерева ненужных веток помогает ему расти, механизм упрощения способствует достижению цели, игнорируя все бесполезные для этого факты. Когда человек сталкивается с какой-то ситуацией, механизм упрощения заставляет его мозг сосредоточиться только на фактах и правилах, нужных для достижения поставленной цели. На рис. 1.1 приведена схема работы механизма упрощения при выборе необходимых правил и фактов. Здесь механизм упрощения используется при выборе правил для ситуации А или Б. В процессе упрощения в соответствии с отношением ЕСЛИ–ТО проверяется некоторое условие. Если условие удовлетворяет состоянию А, выбираются правила для А, если Б — правила для Б. Действие механизма упрощения можно сравнить с действием руководителя учреждения, распределяющего работу в соответствии с ее важностью. Секретарь, машинистка, клерк и другие служащие заняты своими делами. Если же какую-то работу нужно закончить срочно, руководитель должен решить, кто и когда будет ею заниматься.

Если не сделать этого, то одни сотрудники будут сидеть без дела, ожидая, пока другие завершат свою часть работы, в то время как кто-то будет тратить драгоценное время, выполняя несрочную

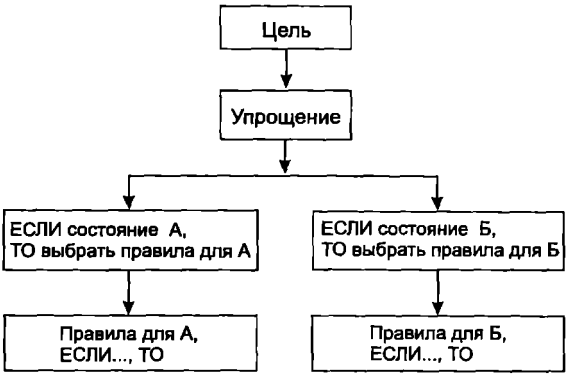


Рис. 1.1. Работа механизма упрощения

работу. Можно проделать огромную работу, но она окажется бессмысленной, если не будет достигнута конечная цель.

Без механизма упрощения человеческий мозг будет парализован, так же как учреждение без руководителя. Человеку при решении каждой задачи нужно будет проверять все известные ему правила. Человеческий мозг подобно учреждению с полным штатом сотрудников различных профессий хранит все знания, необходимые для принятия решения, но если требуемую информацию не выбрать вовремя — все знания бесполезны. Механизм упрощения направляет мысль человека в нужное русло, без него жизнь не только замедлится, но станет просто невозможной.

Механизм вывода

Достигая цель, человек не только приходит к решению поставленной перед ним задачи, но одновременно приобретает новые знания. Рассмотрим такой пример:

1. Джон и Мери — родители Джима.
2. Джон и Мери — родители Джейн.

Цель заключается в том, чтобы определить, кем приходятся друг другу Джим и Джейн. Механизм упрощения заставляет человека обратиться к хранящемуся в его мозгу правилу: ЕСЛИ у девочки и мальчика одни и те же родители, ТО мальчик и девочка — брат и сестра. Цель мгновенно достигнута.

Ответ на вопрос о степени родства Джима и Джейн получен из известного ранее правила. Кроме того, в процессе достижения цели получен новый факт: Джим и Джейн — брат и сестра. Часть интеллекта, которая помогает извлекать новые факты, называется *механизмом вывода.* Именно механизм вывода позволяет человеку учиться на опыте, так как он дает возможность генерировать новые факты из уже существующих, применяя имеющиеся знания к новой ситуации. В следующей главе будет рассмотрено, как механизм вывода помогает обнаружить ошибки в рассуждениях и совершенствовать правила, используемые при достижении целей.

Резюме

Основные идеи, рассмотренные в этой главе, используются на протяжении всей книги при обсуждении процесса создания систем ИИ. Перечислим их еще раз:

1. Цель заставляет человека думать.

2. Человеческий мозг хранит огромное число фактов и правил их использования. Для достижения определенной цели надо только обратиться к нужным фактам и правилам.

3. Механизм упрощения быстро и эффективно выбирает факты и правила, нужные для достижения ближайшей цели.

4. Механизм вывода завершает мыслительный процесс, выполняя заключения на основании правил, отобранных механизмом упрощения, и генерируя новые факты, которые добавляются к знаниям человека.

II. РАЗРАБОТКА СИСТЕМ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Для того чтобы можно было говорить об искусственном интеллекте, программная система должна иметь все элементы, составляющие процесс принятия решения человеком, цели, факты, правила, механизмы вывода и упрощения.

В этом учебном пособии рассматриваются системы ИИ, основанные на правилах. Все они однотипны; основные компоненты таких систем показаны на рис. 2.1. Главное отличие системы ИИ от традиционных программных систем заключается в том, что различные компоненты ее структуры определяются раздельно и модификация любой ее части не затрагивает общей структуры. Благодаря такому подходу можно выделить отдельные составляющие мыслительного процесса человека, решающего задачу, и включить их в систему ИИ. Определив, как человек мыслит на каждой стадии процесса принятия решения, в программу легко можно включить блок, реализующий действия, аналогичные человеческому мышлению на этой же стадии.

Определение целей

При проектировании системы ИИ прежде всего нужно определить цели, для достижения которых она предназначена. Приступая к разработке программы, необходимой для решения некоторой задачи, надо знать, к какому классу относится данная задача, и уметь описать ее в нужных терминах.

Вновь обратимся к примеру, рассматривавшемуся в разделе I. Предположим, что человек быстро и безопасно перешел улицу. Стоя на автобусной остановке, он за несколько секунд должен решить, ждать ли ему экспресс или пройти еще полквартала, где останавливается местный автобус, на котором при определенных

обстоятельствах можно добраться быстрее. Какой же автобус выбрать?

Прежде чем принять решение, человек взвешивает различные факторы. Например, через сколько минут прибудет экспресс? В каком автобусе вероятнее всего будут свободные места? Некоторые правила возникают в результате упрощений («ЕСЛИ мне придется ждать экспресс больше 10 мин, ТО быстрее я доберусь до работы местным автобусом»).

В процессе принятия решения постоянно используются правила, в том числе и специальные правила механизма упрощения. Для человека факты и правила, относящиеся к задачам, связанным с работой, не имеют никакого отношения к поездке на работу. Аналогично этому и в системе ИИ несущественными считаются все факты и правила, не относящиеся к достижению сформулированной цели.

Рассмотрим теперь задачу, которая более подробно будет описана в разделе III при обсуждении проектов реальных систем ИИ. Цель заключается в том, чтобы определить, есть ли у ребенка трудности при изучении арифметики. Эта задача сильно отличается от предыдущей, но именно поэтому она и выбрана. Задача поможет понять, что основные мыслительные процессы человека остаются одинаковыми при решении многих задач, а одну базовую систему ИИ можно использовать при решении огромного числа задач, какими бы разными они ни казались.

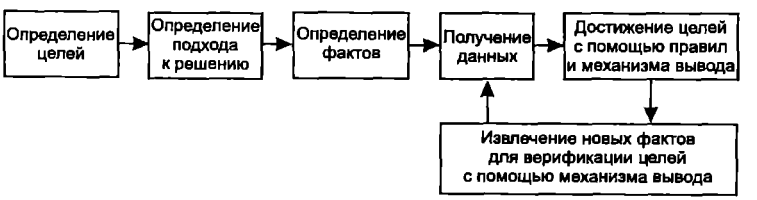


Рис. 2.1. Компоненты системы ИИ

Последнюю цель можно разбить на подцели, относящиеся к разным разделам арифметики, и сформулировать их следующим образом:

Испытывает ли PERSON трудности при изучении СЛОЖЕНИЯ?

Испытывает ли PERSON трудности при изучении ВЫЧИТАНИЯ?

Испытывает ли PERSON трудности при изучении УМНОЖЕНИЯ?

Переменная PERSON введена для достижения общности. При решении частной задачи ей присваивается имя конкретного ребенка. Ниже приводится предложение языка БЕЙСИК для ввода значения в переменную PERSON:

INPUT «Введите имя»; PERSON

При выполнении компьютером этого предложения на экране появится надпись:

Введите имя?

В ответ надо ввести имя ребенка, допустим Эндрю Джонс. После этого всякий раз при обращении к переменной PERSON в фактах, правилах или целях будет использоваться имя Эндрю Джонс. Когда переменной присваивается значение, ее называют *проинициализированной.* Теперь уже цель — узнать, есть ли трудности при изучении какого-либо раздела арифметики у Эндрю Джонса.

Определение фактов

Факты — важная часть системы ИИ, без них нельзя достичь цели. Выбирая автобус, человеку требовались только нужные для этого факты. Очевидно, что поскольку у каждой нестандартной ситуации — свой сценарий, у каждой цели — свои факты. Для того чтобы разобраться, как используются факты, рассмотрим одну из подцелей последней задачи:

Испытывает ли PERSON трудности при изучении вычитания?

Вычитание требует определенных математических навыков. Навыки и являются теми фактами, которые рассматриваются при достижении цели. Для выполнения вычитания ребенок должен уметь:

1. Вычитать два числа без переноса (2).
2. Вычитать два числа с переносом в одном столбце (1).
3. Вычитать два числа с переносом в нескольких столбцах (1).

Числа в скобках указывают относительную важность (вес) фактов, использующихся при оценке трудностей, возникающих у ре-

бенка при изучении арифметики. Можно провести аналогию с обычными тестами, в которых одни вопросы важнее других. Каждый факт соответствует определенному навыку, необходимому для правильного выполнения операции вычитания, а отсутствие какого-либо навыка означает, что ребенок полностью вычитания не освоил. Чем больше назначенный факту вес, тем большее значение имеет этот факт при решении задачи. Использование весов фактов для достижения цели кратко поясняется ниже.

Получение данных

После того как определены общие факты, необходимые для достижения цели, надо получить конкретные данные и присвоить значения переменным. Другими словами, вначале определяются навыки, требуемые для выполнения вычитания, а затем нужны данные об этих навыках у Эндрю Джонса. Прежде всего факты надо представить в форме вопросов, ответив на которые можно получить необходимую информацию. Ниже приведены факты в форме вопросов, ответы на которые дадут информацию, нужную для оценки успехов ребенка в вычитании.

Может ли PERSON вычесть два числа без переноса?

Может ли PERSON вычесть два числа с переносом в одном столбце?

Может ли PERSON вычесть два числа с переносом в нескольких столбцах?

Точно так же, как переменная PERSON заменяется на имя Эндрю Джонс, факты, представленные в виде вопросов, можно заменить на ответы. Факты, содержащие конкретную информацию, становятся данными. Иначе говоря, это ответы в форме да/нет на вопросы. Затем данные заносятся в базу данных, и их можно будет использовать при оценке знаний арифметики Эндрю Джонсом. Для вывода вопросов и получения ответов нужна соответствующая программа, которую целесообразно продемонстрировать на языке БЕЙСИК. Благодаря нумерации строк и простой семантике языка легко проследить логику решения задачи. Например:

```
5 REM Переменная PERSON — имя ребенка
10 INPUT «Введите имя ребенка»; PERSON
```

15 REM в S1$ будет храниться ответ (да/нет) на следующий вопрос:

20 PRINT «Умеет ли "PERSON" вычитать два числа без переноса»

25 INPUT S1$

30 REM в S2$ будет храниться ответ (да/нет) на следующий вопрос:

35 PRINT «Умеет ли "PERSON" вычитать два числа с переносом в одном столбце»

40 INPUT S2$

45 REM в S3$ будет храниться ответ (да/нет) на следующий вопрос:

50 PRINT «Умеет ли "PERSON" вычитать два числа с переносом в нескольких столбцах»

55 INPUT S3$

Программа в режиме диалога выводит на дисплей вопросы, ответы на которые будут использоваться для оценки трудностей при изучении вычитания конкретным ребенком. Ниже приводится весь диалог с компьютером. Прописными буквами пишутся вопросы, задаваемые компьютером, а строчными — возможные ответы пользователя.

ВВЕДИТЕ ИМЯ РЕБЕНКА? Эндрю Джонс

УМЕЕТ ЭНДРЮ ДЖОНС ВЫЧИТАТЬ ДВА ЧИСЛА БЕЗ ПЕРЕНОСА? Да

УМЕЕТ ЭНДРЮ ДЖОНС ВЫЧИТАТЬ ДВА ЧИСЛА С ПЕРЕНОСОМ В ОДНОМ СТОЛБЦЕ? Нет

УМЕЕТ ЭНДРЮ ДЖОНС ВЫЧИТАТЬ ДВА ЧИСЛА С ПЕРЕНОСОМ В НЕСКОЛЬКИХ СТОЛБЦАХ? Нет

Теперь, как показано на рис. 2.2, все проинициализированные переменные становятся частью базы данных. До сих пор все было довольно просто: формулировались факты, необходимые для выполнения вычитания; разрабатывалась программа, задающая вопросы, относящиеся к этим фактам, а полученные ответы сохранялись для последующего анализа. Выбирая между местным автобусом и экспрессом, человеческий мозг автоматически делает то же самое. Человек сам задает себе все необходимые вопросы,

опираясь на уже известные ему факты, касающиеся этих автобусов и автобусов вообще. Факты формулируются в виде вопросов, ответы на которые помогают человеку принять окончательное решение. Факты и правила, которые будут обсуждаться в дальнейшем, хранятся в компьютере в так называемой *базе знаний.*

Когда человек сталкивается с проблемой, какой выбрать автобус, не все соображения для него равноценны, например, если надо попасть на работу вовремя, он не будет слишком заботиться о том, чтобы сидеть в автобусе. Можно сказать, что человек «взвешивает» различные соображения. Числа в скобках поэтому и называются весами или весовыми факторами фактов.

Ответ «нет» на любой из вопросов (см. рис. 2.2) говорит о том, что у Эндрю Джонса нет соответствующего навыка, необходимого для выполнения вычитания, а ответ «да» — что этот навык есть. Получив сумму весовых факторов для отрицательных ответов, можно узнать, как велики у Эндрю Джонса трудности при изучении вычитания, например:

S1$ (2) да 0 S2$ (1) нет 1 S3$ (1) нет 1 Общий весовой фактор=2

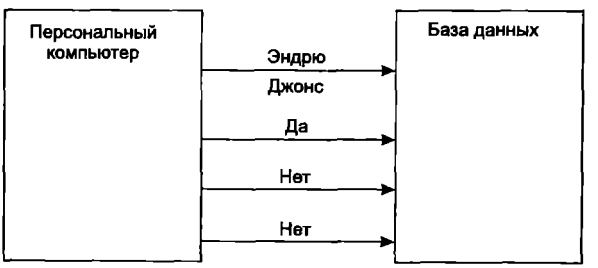


Рис. 2.2. Ввод данных

Итоговое число, оценивающее трудности при изучении вычитания у Эндрю Джонса, называется *общим весовым фактором.* Общий весовой фактор можно использовать, чтобы определить, есть ли у Эндрю Джонса трудности при изучении вычитания и как они велики, то есть это еще и некоторая количественная оценка. Общий весовой фактор в последнем примере при ответах «нет» на все вопросы имел бы максимальное значение, которое указывало

бы на очень серьезные трудности при изучении вычитания. В примере с автобусом наихудший результат получился бы в том случае, если бы пришлось ждать экспресс, всю дорогу до работы стоять и при этом опоздать на работу на час. Рассматривать наихудшие последствия принимаемых решений всегда полезно.

Правила и выводы

Как отмечалось в первой главе, известные факты применяются к возникшей ситуации в соответствии с правилами. Правила в системе ИИ помогают верно оценить данные и достичь цели, например узнать, есть ли у ребенка трудности при изучении арифметики. Вернемся к примеру с вычитанием и сформулируем правило, позволяющее оценить трудности, возникающие при его изучении:

ЕСЛИ общий весовой фактор больше 1, ТО у PERSON есть трудности при изучении вычитания.

Результирующий весовой фактор у Эндрю Джонса равен 2, и в соответствии с приведенным выше правилом можно сделать вывод — Эндрю Джонс действительно испытывает трудности при изучении вычитания. Правило обеспечило практическое применение накопленных данных. Значение 1 в правиле называется *граничным уровнем решения*.

Ранее уже говорилось, что методы ИИ делают программирование более легким и гибким, рассмотрим теперь, каким образом это достигается. Вместо того чтобы формулировать правило, можно было бы воспользоваться фактами, охватывающими все возможные варианты, при которых общий весовой фактор больше 1. Ниже приводятся факты, которые, будем надеяться, удовлетворяют поставленному условию, и результирующие факторы для них. Ребенок не умеет вычитать, если:

1. (S1$=нет) И (S2$=да) И (S3$=да) общий весовой фактор=2
2. (S1$=нет) И (S2$=да) И (S3$=нет) общий весовой фактор=3
3. (S1$=нет) И (S2$=нет) И (S3$=да) общий весовой фактор=3
4. (S1$=нет) И (S2$=нет) И (S3$=нет) общий весовой фактор=4
5. (S1$=да) И (S2$=нет) И (S3$=нет) общий весовой фактор=2

Отметим, что для всех совокупностей фактов общий весовой фактор больше 1. В списке представлено пять различных сово-

купностей фактов, а сформулированное правило уже включает в себя все эти факты, и их не надо специально программировать. Очевидно, что намного легче запрограммировать одно правило, включающее все факты, чем несколько отдельных фактов, применение которых ограничено. Если бы не было правил, человек все еще стоял бы на углу улицы и анализировал бы огромное число фактов и связанные с ними данные, не зная, что ему делать. Ему не надо было бы думать об автобусе, поскольку он никогда бы не дошел до остановки!

До сих пор разработка программы состояла из:

1. Определения целей.
2. Определения фактов, имеющих отношение к этим целям.
3. Получения данных, соответствующих фактам, характерным для заданной ситуации или объекта.
4. Оценки данных с использованием правил и механизмов вывода.

Процесс достижения целей описанным способом называется *прямой цепочкой рассуждений,* то есть цепочкой от данных к логическому заключению. Он позволяет логически переходить от одного шага к другому, как представлено на рис. 2.1.

Верификация целей механизмом вывода

Рассмотрим пример, иллюстрирующий проверку выводов. Предположим, совершено преступление: в квартире был обнаружен труп с тремя пулевыми ранениями.

Медицинский эксперт по виду ранений (упрощение в действии) сделал вывод о невозможности самоубийства, и полиция начала расследование. Первое, чем поинтересовались полицейские, — кто, кроме потерпевшего, имел ключ от квартиры? Расспросив хозяина дома и некоторых соседей, они узнали, что у убитого был приятель, который часто пользовался его квартирой. Дальнейшее расследование показало, что друзья недавно поссорились.

Теперь у полиции был подозреваемый. Опросив свидетелей, полиция могла сделать вывод, что приятель потерпевшего и является вероятным убийцей (прямая цепочка рассуждений); для того чтобы прийти к такому заключению, полицейские пользовались информацией, полученной от соседей, но для завершения дела

были необходимы неопровержимые улики. Вероятная возможность арестовать преступника — найти его оружие. Полицейские получили разрешение на обыск квартиры приятеля потерпевшего. Осмотрев всю квартиру, они ничего не обнаружили. Наконец, один из полицейских в соседнем переулке в мусорном баке нашел ружье. Снятие отпечатков пальцев показало, что на ружье есть отпечатки пальцев подозреваемого, а баллистические эксперименты подтвердили, что человека убили из этого ружья. Преступление таким образом было раскрыто. Получая новые данные и проверяя, согласуются ли они с изначальным заключением, полиция верифицировала цель — идентификацию убийцы. Процесс, в котором заключение используется для поиска подтверждающих его данных, называется *обратной цепочкой рассуждений.* В рассмотренном примере заключение — это подозреваемый, а данные — это оружие. В системе ИИ цель верифицируется аналогично. Был ли верен вывод, что у Эндрю Джонса есть трудности при изучении вычитания? Цель была достигнута, но чтобы проверить ее правильность, необходимо проанализировать задачу снова с данными и правилами. Для проверки, есть ли у ребенка трудности при изучении вычитания, можно воспользоваться следующим правилом:

ЕСЛИ за выполнение простого арифметического примера на вычитание ребенок получил самую низкую оценку (единицу), ТО общий весовой фактор по вычитанию будет у него больше 1.

Для того чтобы верифицировать цель, то есть подтвердить правильность вывода о наличии у ребенка серьезных трудностей с вычитанием, нужно проверить, получил ли он за пример на вычитание самую низкую оценку. Это можно сделать с помощью следующей программы:

```
5 REM Оценка за пример на вычитание
10 INPUT «Введите оценку за пример»; GT
15 REM GS — общий весовой фактор
20 INPUT «Введите общий весовой фактор»; GS 25 G=GS —
GT 30 IF GO THEN VER=1
```

Еще одним примером обратной цепочки рассуждений может быть выбор механизма вывода новых данных для подтверждения заключения, которое изначально предполагается верным. Как

следует из самого названия, обратная цепочка рассуждений идет в сторону, противоположную прямой цепочке, то есть от заключения к данным. Обратная цепочка возникает после того, как цель уже достигнута. Рисунок 2.3 иллюстрирует прямую и обратную цепочки рассуждений в рассмотренной модели системы ИИ.

Если установлено, что Эндрю Джонс не умеет вычитать, механизм вывода проверит, получил ли он также самый низкий балл за пример на вычитание. Если это так, то цель достигнута верно. Если же оценка Эндрю Джонса за пример на вычитание не очень низкая, значит, для достижения цели не хватает данных и необходимо сделать дополнительные шаги для ее повторной проверки, то есть надо устранить противоречие: с одной стороны, Эндрю Джонс не знает некоторых правил вычитания, а с другой — он имеет удовлетворительную оценку за пример на вычитание.

Упрощение

В мозгу человека механизм упрощения руководит поиском дополнительных правил для верификации цели до тех пор, пока не будут проверены все возможные способы ее достижения. Мозг хранит огромные объемы информации, поэтому для верификации одной цели может существовать много правил, каждое из которых в любой момент может включиться в работу.

Механизм упрощения программы позволяет компьютеру пропустить или обработать какую-то часть данных из базы знаний в зависимости от ее важности для достижения определенной цели. Если Эндрю Джонс получил очень хороший балл за простой пример на вычитание, то механизм упрощения пропустит информацию из базы знаний, нужную для оценки арифметических навыков. Но если Эндрю Джонс не справился с примером, то механизм упрощения заставит программу обратиться к этой информации. Такой подход позволяет не рассматривать пути, не ведущие к достижению цели.

Напомним, что механизм упрощения можно представить с помощью набора правил, используемых при работе (см. раздел I). Для примера с вычитанием можно воспользоваться, скажем, следующим правилом:

ЕСЛИ отметка за пример на вычитание
превышает результирующий фактор на 2,
ТО оценивать трудности при вычитании не нужно.

На языке БЕЙСИК это правило можно запрограммировать следующим образом:

Рис. 2.3. Конфигурация и работа системы ИИ

```
110 INPUT «Отметка за пример на вычитание»; SS
115 INPUT «Результирующий весовой фактор»; CL
120 REM если отметка за пример превышает общий
121 REM весовой фактор на 2 или больше,
122 REM не оценивать трудности при вычитании
125 G=SS (GL+2)
130 IF G=0 THEN GOTO
140 135 GOSUB 150
140 REM трудности с другими действиями арифметики 145...
150 REM оценить трудности при вычитании
```

И в мозгу человека, и в системах искусственного интеллекта механизм упрощения просто игнорирует ненужные и не относящиеся к делу рассуждения. Правило упрощения для примера с поездкой на работу можно сформулировать так:

ЕСЛИ идет сильный дождь, ТО следует проигнорировать любую дополнительную информацию о местном автобусе или экспрессе и воспользоваться первым же подходящим автобусом.

III. ОБЩИЕ СВЕДЕНИЯ ОБ ЭКСПЕРТНЫХ СИСТЕМАХ

В предыдущем разделе рассматривалось решение задач в системе искусственного интеллекта. Для достижения целей система использовала правила, факты, механизмы вывода и упрощения. Конкретные сферы человеческой деятельности, в которых могут применяться системы ИИ, называются *предметными областями.* Примерами предметных областей могут служить оценка эффективности обучения и выбор маршрута автобуса. На первый взгляд кажется, что создание единой системы ИИ, охватывающей все предметные области, возможно. Это глубокое заблуждение. Прежде всего следует учитывать, что для решения всех возможных задач во всех предметных областях необходимо бесконечное число фактов и правил. Даже если бы такая система была создана, понадобилось бы длительное время на наполнение ее знаниями. Более того, сегодня еще нет вычислительной машины, способной хранить и обрабатывать такой объем информации, поэтому пока нужно ограничиться только проблемными областями, в которых объем информации не слишком велик, и ее можно обработать в программе. Хотя в настоящее время возможности технологии ИИ ограничены, все же, как видно из примеров предыдущей главы, уже сегодня практическое применение систем ИИ реально, и для узких проблемных областей можно разрабатывать работоспособные системы ИИ. Примеры таких проблемных областей приведены на рис. 3.1.

Исходя из практических соображений для работы надо выбирать проблемные области, в которых объем информации поддается управлению. Система ИИ, созданная для решения задач в конкретной проблемной области, называется *экспертной системой.*

Источником знаний для наполнения экспертных систем служат эксперты в соответствующей предметной области. Весовые факторы, о которых говорилось в разделе II, тоже выбираются не случайно, они представляют собой знания, полученные в результате исследования проблемной области. Работа всех экспертных систем основана строго на экспертной информации, полученной в конкретной проблемной области.

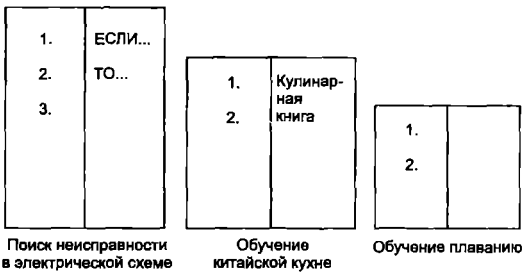


Рис. 3.1. Примеры предметных областей

Эвристические правила

Допустим, что собрана группа экспертов для решения следующей задачи: в реке обнаружены пятна нефти, сбросить которую могло любое из предприятий, расположенных у реки. Цель состоит в том, чтобы точно узнать, какое предприятие сбрасывает нефть, и устранить последствия загрязнения реки. Прежде всего экспертная система, решающая такую задачу, должна уметь выполнять расчеты.

Приблизительно определить место сброса нефти можно, например, используя время растворения нефти в воде, направление и скорость течения реки и т. д. После того как эта информация собрана и определено место сброса нефти (скажем, с точностью до мили), можно воспользоваться специальными правилами, например правилами Э1 и Э2, описанными ниже, для точного определения виновника. Эти правила называются эвристическими. Их главное отличие от правил, рассмотренных в предыдущих главах, состоит в следующем: эвристические правила формулируются не на основании обычных, признанных знаний, а на основании практических знаний эксперта.

Правило Э1: ЕСЛИ в верховье реки в пределах одной мили от вычисленного места сброса нефти находится одно предприятие, ТО это предприятие и сбросило нефть в реку.

Правило Э2: ЕСЛИ в верховье реки в пределах одной мили от вычисленного места сброса нефти расположено несколько предприятий, ТО предполагается, что нефть сбросило предприятие, использующее наибольшее количество нефти.

В разделах I и II описывалось, как упрощение повышает эффективность мышления, сокращая число рассматриваемых вариантов при достижении цели. В экспертных системах механизм упрощения, использующий эвристические правила, называется *эвристическим механизмом поиска*. На рис. 3.2 приведен пример эвристического механизма поиска, похожий на механизм упрощения, описанный в разделе II.

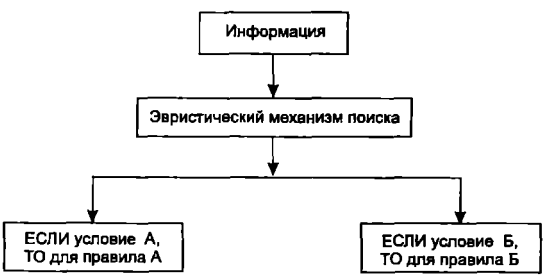


Рис. 3.2. Механизм упрощения в экспертной системе

После того как эвристический механизм поиска выделил наиболее вероятного виновника загрязнения воды, для проверки выдвинутой гипотезы экспертная система может использовать другой набор правил. Прежде всего будут рассматриваться наиболее уязвимые места в системе нефтепровода предприятия. Здесь может оказаться полезным такое эвристическое правило:

ЕСЛИ давление в фабричном нефтепроводе на участке 1 меньше давления воды в реке, ТО следует проверить участок 2.

Такое правило может сформулировать только человек, практически знающий именно это предприятие, то есть оно эвристическое.

Рабочая область

Задача зачастую бывает слишком сложна для одной экспертной системы. Например, при решении задачи с нефтью одна экспертная система способна найти место сброса, но для определения состава нефти нужен уже другой тип экспертизы. Экспертная система, определяющая состав нефти, должна взять часть информации из экспертной системы поиска места сброса нефти. Затем, выполнив свои вычисления и используя для решения задачи эвристический механизм поиска, она в конечном счете может порекомендовать оптимальное химическое вещество, нейтрализующее нефть данного сорта в конкретных условиях. Обмен информацией между экспертными системами осуществляется через рабочую область. Рабочая область — это место в памяти ЭВМ, куда каждая экспертная система помещает свою информацию для использования другой экспертной системой.

Рабочая область представляет собой информационную структуру, доступную всем совместно работающим экспертным системам. Дисциплина использования этой информации каждой экспертной системой зависит от конкретного приложения. Рабочая область подобна школьной доске, на которой секретарь, знающий о болезни учителя, может написать сообщение о слиянии двух классов на один урок. Для совместно работающих экспертных систем идея рабочей области, в которой можно оставить, просмотреть или изменить информацию, очень важна.

Разработать экспертную систему может каждый

При создании экспертной системы группа, состоящая из эксперта и инженера по знаниям, собирает факты, правила и эвристические правила, относящиеся к проблемной области, а затем включает их в программу ИИ. Однако тех понятий ИИ, которые были даны в первых двух разделах, вполне достаточно для того, чтобы начать разработку собственной экспертной системы. Тот, кто знает, как ремонтировать автомобили, готовить еду или плавать, может стать инженером по знаниям, способным разработать экспертную систему в одной из этих или сходных проблемных областях.

IV. ОБРАБОТКА ЕСТЕСТВЕННОГО ЯЗЫКА

Общение человека и компьютера — это задача, над которой работают множество исследователей. Конечная цель (до которой еще очень далеко) заключается в том, чтобы пользователи могли «разговаривать» с компьютером на естественном языке, например русском, и компьютер отвечал им на том же языке. Можно надеяться, что когда-нибудь в простой беседе люди будут объяснять компьютеру задачу, и он сможет ответить им.

Может показаться, что это просто, но только потому, что мы с детства привыкли разговаривать и слышать и воспринимаем человеческую речь как само собой разумеющееся большинство сложных свойств языка. Те, кому доводилось работать с языком БЕЙСИК или другим языком программирования, знают, насколько важно правильно составить программу, чтобы заставить компьютер сделать то, что нужно. Компьютеры умны ровно настолько, насколько такими их сделали люди: поскольку сами по себе они не умеют «думать», необходимо дать им очень точные инструкции о том, какие действия они должны выполнять вплоть до мельчайших шагов. Человек рождается «запрограммированным» на изучение языка; для него половина дела уже сделана. Чтобы компьютер понимал язык, нужно не только разбить язык на его основные элементы и ввести эту информацию в компьютер, но сначала надо разработать компьютеры и программы, которые смогут воспринимать эту информацию. Для того чтобы общение между человеком и компьютером стало возможным, нужно создать систему обработки естественного языка. Цель данного раздела — дать некоторое представление об этом важном и увлекательном аспекте искусственного интеллекта.

Давайте посмотрим, насколько трудна эта задача. Представим себе, что есть робот, обладающий искусственным интеллектом, умеющий чинить автомобили.

Ему можно дать одну из следующих команд:

1. Почини машину около дома со спущенной покрышкой.

2. Почини машину около дома с красной занавеской.

Хотя первое предложение интерпретируется двояко, каждый может уловить его смысл — все знают, что у дома не может быть спущенной покрышки. Человек замечает неточность, но еще важнее то, что он ее в уме исправляет; ведь очевидно — спущенная покрышка у машины, а не у дома. Роботу нужно знать нечто большее, чем только значения слов и их связь друг с другом, иначе ему придется искать дом со спущенной покрышкой. Так как оба предложения имеют одинаковую структуру, робот должен знать грамматику, а также уметь соотносить описания и объекты. Важно сознавать, что правила человеческого языка имеют смысл только для людей, а роботу, чтобы понимать, о чем мы говорим, необходимы специальные правила.

Искусственный интеллект, которым обладает наш робот, должен уметь анализировать предложения в их связи с другими предложениями. Возьмем для примера такие два предложения:

1. Джон пьет молоко. 2. Затем он надевает пальто.

Слово «он» во втором предложении относится к слову «Джон» в первом предложении. Без первого предложения второе не имело бы смысла. Все естественные языки называются контекстуальными языками. Иначе говоря, чтобы полностью понять второе предложение, необходимо знать первое, а это и есть контекстуальная зависимость. Языки, в которых интерпретация предложения может быть выполнена без знания других предложений, называются *контекстуально независимыми,* то есть безразличными к последовательности событий и действий.

Для того чтобы заставить компьютер понимать человеческую речь, необходимо построить анализатор естественного языка. Основными функциями анализа языка являются:

1. Лексический анализ (анализ слов).

2. Синтаксический анализ (анализ порядка слов в предложении с учетом правил грамматики).

3. Семантический анализ (анализ значения предложения самого по себе и в его связи с другими предложениями).

Лексический анализ

Деление предложения на слова с использованием знаков препинания (при написании) или пауз (в разговоре) называется *лексическим анализом.* Кроме того, в составе слов можно выделить корни, приставки и окончания. Например, очень простое слово «дорога» можно последовательно разделить на:

дорога (слово) дорог (корень) а (окончание).

Слово «предлог» состоит из: пред (приставка) лог (корень). Набор слов можно взять из словаря, но объяснить компьютеру их смысл в общем контексте — более сложная задача.

Синтаксический анализ

Для того чтобы научить компьютер воспринимать человеческий язык, необходимо прежде всего научить его выполнять разбор предложения. Надо перевести правила грамматики и синтаксиса в форму, которую компьютер мог бы понять.

Обычно предложение (П) состоит из группы существительного (ГС) и группы глагола (ГГ), что можно представить как: П → ГС, ГГ. Группа существительного может быть разбита на определение (например прилагательное, притяжательное местоимение и т. д.) и существительное: ГС → О, С. Группу глагола можно разбить на глагол, за которым следует дополнение, выраженное существительным (то есть другая группа существительного): ГГ → Г, ГС. Группа существительного может быть представлена единственным членом: ГС →С. Графически синтаксическая структура предложения может быть представлена в виде «дерева». Например, предложение: «Старый дровосек рубит деревья» имеет структуру, показанную на рис. 4.1. Предложение разбивается на слова, а слова классифицируются по типу. Слово «старый» — это определение (О), выраженное прилагательным, «дровосек» — существительное (С), «рубит» — глагол (Г) и «деревья» — существительное (С).

Семантический анализ

Разбив предложение на составные части, компьютер проводит его семантический анализ, то есть пытается понять его смысл. В системах искусственного интеллекта применяется некоторая совокупность правил, позволяющая компьютеру понять смысл предложения: О С Г С — Старый дровосек рубит деревья. Для интерпретации предложения в базе знаний семантического анализатора должен быть следующий набор правил.

Правило 1: ЕСЛИ определение стоит на первом месте и за ним идет существительное, ТО существительное является подлежащим.

Правило 2: ЕСЛИ за подлежащим идет глагол, ТО этот глагол является сказуемым и поясняет, что делает подлежащее.

Правило 3: ЕСЛИ за подлежащим идет сказуемое, а за ним следует существительное, ТО это существительное является дополнением.

Правило 4: ЕСЛИ предложение имеет следующий порядок слов: подлежащее, глагол, дополнение, ТО вся фраза говорит о том, что подлежащее делает (действие, выраженное сказуемым) по отношению к дополнению.

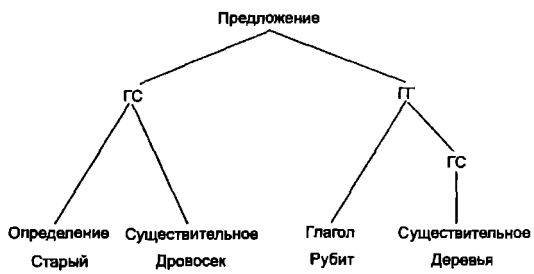


Рис. 4.1. Синтаксическое дерево предложения

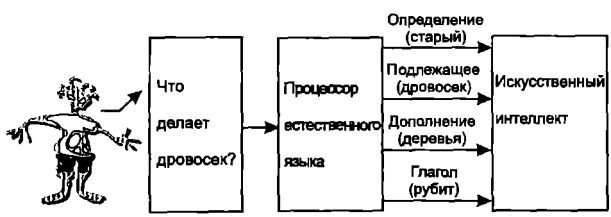


Рис. 4.2. Процессор естественного языка

Поясним сказанное на примере. Предположим, что система искусственного интеллекта должна решить следующую задачу: узнать, что делает дровосек и что является объектом его действия. Семантический анализатор обращается к правилу 1, с помощью которого определяет, что слово «дровосек» — это подлежащее. С помощью правила 2 определяется, что слово «рубит» — это сказуемое. Объект действия, выраженный словом «дерево», устанавливается с помощью правил 3 и 4. Данный пример показывает, как процессор естественного языка обрабатывает или «понимает» предложение, используя лексические, синтаксические и семантические правила своей базы знаний.

Процессор естественного языка может служить промежуточным звеном между пользователем и другой системой искусственного интеллекта, позволяя человеку устно общаться с компьютером (см. рис. 4.2). По существу, обработка естественного языка может освободить пользователя компьютера от необходимости изучать сложные языки программирования. Если удастся создать программы, которые позволят компьютеру и пользователю общаться на естественном языке, то будет сделан огромный шаг на пути создания подлинно «интеллектуального» компьютера.

V. ПРЯМАЯ ЦЕПОЧКА РАССУЖДЕНИЙ

Приступим к разработке реальной экспертной системы, предназначенной для решения задач определенного типа. Задачи каждого типа решаются наиболее оптимальными для них методами, и, следовательно, экспертная система должна быть ориентирована на строго конкретную предметную область, иначе она будет бесполезна. Ремонтируя автомобиль, не берут двуручную пилу, а гаечным ключом не рубят деревья.

Рассмотрим такую ситуацию: во время движения у автомобиля перегревается двигатель. Как отреагирует на это водитель? Конечно, он занервничает. Многие с этим сталкивались жарким днем в час пик. Сформулируем задачу в более общем виде. Имеет место ситуация (перегрев двигателя), требуется предсказать ее последствие (заглохнет ли мотор?). Итак, прежде всего зафиксировано возникновение определенного состояния (перегрев двигателя), а затем в работу включаются относящиеся к нему правила:

Правило 1: ЕСЛИ двигатель перегрелся, ТО мотор заглохнет.

Правило 2: ЕСЛИ мотор заглохнет, ТО это приведет к денежным затратам и позднему возвращению домой.

Каким образом можно прийти к выводу о том, что перегрев двигателя ведет к денежным затратам и позднему возвращению домой? Это можно сделать, используя прямую цепочку рассуждений. Отправной точкой рассуждений служит возникшая ситуация (перегрев двигателя). Затем срабатывает условная часть (часть ЕСЛИ) первого правила. Поскольку возникшая ситуация удовлетворяет содержащемуся в ней условию, согласно констатирующей части этого правила (части ТО) выводится новая ситуация (мотор заглохнет). Цепочка рассуждений продолжается.

Условие, содержащееся в части ЕСЛИ второго правила, удовлетворяется, если уже сработало первое правило (действительно заглох мотор). Следовательно, вывод о денежных затратах и позднем возвращении домой можно сделать при возникновении двух ситуаций: двигатель перегрелся или мотор заглох. Описанная последовательность рассуждений называется прямой цепочкой потому, что констатирующая часть правила (часть ТО) выполняется только в том случае, если удовлетворяется условная часть правила (часть ЕСЛИ). Отправной точкой рассуждений, таким образом, служит уже возникшая ситуация, а затем делаются выводы.

Что же должна делать программа, реализующая прямую цепочку рассуждений? Программа должна отвечать на вопросы пользователя на основе принципов прямой цепочки рассуждений и базы знаний. Другими словами, для приведенного примера она должна запросить у пользователя данные о возникшей ситуации (например о перегреве двигателя) и имя базы знаний, скажем, «Технические проблемы автомобилизма», просмотреть информацию в базе знаний и затем, проведя прямую цепочку рассуждений, сделать вывод либо о том, что мотор заглохнет, либо о том, что за ремонт нужно будет заплатить и придется поздно приехать домой.

Самый простой способ пояснить процесс разработки программного средства — это описать на бумаге все шаги решения задачи без компьютера. Составление подобного описания называется разработкой алгоритма. Алгоритм — это детальный перечень всех логических шагов решения задачи.

Решая задачу, компьютер выполняет те же действия, что и человек, только быстрее, и, следовательно, следующим этапом при разработке системы ИИ должно быть составление программы по этому алгоритму. В настоящей главе подробно обсуждаются оба этапа разработки программ.

Простая экспертная система, реализация которой приведена ниже, написана на стандартном БЕЙСИКе, поэтому ее можно запустить практически на любом персональном компьютере. Читателю надо только построить базу знаний для своей области, и он

сможет воспользоваться предложенным вариантом экспертной системы. Кроме того, это система открытого типа, и ее можно модифицировать для решения более сложных задач.

Здесь же подробно обсуждается проектирование экспертной системы для конкретной задачи. Точно следуя данным рекомендациям и применяя их к известной проблемной области, читатель сможет строить базы знаний.

Пример прямой цепочки рассуждений

Для экспертной системы фондовой биржи можно было бы воспользоваться, например, такими правилами:

10 ЕСЛИ ПРОЦЕНТНЫЕ СТАВКИ=ПАДАЮТ, ТО УРОВЕНЬ ЦЕН НА БИРЖЕ=РАСТЕТ

20 ЕСЛИ ПРОЦЕНТНЫЕ СТАВКИ=РАСТУТ, ТО УРОВЕНЬ ЦЕН НА БИРЖЕ=ПАДАЕТ

30 ЕСЛИ ВАЛЮТНЫЙ КУРС ДОЛЛАРА=ПАДАЕТ, ТО ПРОЦЕНТНЫЕ СТАВКИ=РАСТУТ

40 ЕСЛИ ВАЛЮТНЫЙ КУРС ДОЛЛАРА=РАСТЕТ, ТО ПРОЦЕНТНЫЕ СТАВКИ=ПАДАЮТ

Предположим, создана фирма, дающая на основе этих правил консультации в области биржевых операций. Пусть первый клиент фирмы сообщил, что валютный курс доллара падает по отношению к основным валютам других стран, и попросил совета. Цель, очевидно, заключается в выборе правильного поведения на бирже, но останется ли при этом клиент в выигрыше, зависит от пока еще не определенных условий. Напомним, что в системе, реализующей прямую цепочку рассуждений, прогнозы выполняются следующим образом: если возникшая ситуация удовлетворяет условной части правила (части ЕСЛИ), делается логический вывод, определенный в констатирующей части (части ТО). Для приведенного примера необходимо, чтобы в условной части какого-либо правила содержалось бы условие:

ВАЛЮТНЫЙ КУРС ДОЛЛАРА=ПАДАЕТ

Такое условие содержится только в правиле 30:

30 ЕСЛИ ВАЛЮТНЫЙ КУРС ДОЛЛАРА=ПАДАЕТ,
ТО ПРОЦЕНТНЫЕ СТАВКИ=РАСТУТ

В соответствии с этим правилом можно сделать вывод о росте процентных ставок. О ВАЛЮТНОМ КУРСЕ ДОЛЛАРА упоминается еще в правиле 40. Но условие, записанное в этом правиле,

40 ЕСЛИ ВАЛЮТНЫЙ КУРС ДОЛЛАРА=РАСТЕТ

не соответствует исходному состоянию падения валютного курса доллара, и поэтому правило 40 в дальнейших рассуждениях не будет участвовать. Рассуждения еще не закончены, так как правило 30 в свою очередь порождает новую ситуацию:

ПРОЦЕНТНЫЕ СТАВКИ=РАСТУТ

Необходимо проверить, не приведет ли она к другим выводам. Видно, что в правиле 10

10 ЕСЛИ ПРОЦЕНТНЫЕ СТАВКИ=ПАДАЮТ,
ТО УРОВЕНЬ ЦЕН НА БИРЖЕ=РАСТЕТ

подходящего условия нет, а в правиле 20

20 ЕСЛИ ПРОЦЕНТНЫЕ СТАВКИ=РАСТУТ,
ТО УРОВЕНЬ ЦЕН НА БИРЖЕ=ПАДАЕТ

есть. Возникает новая ситуация:

УРОВЕНЬ ЦЕН НА БИРЖЕ=ПАДАЕТ

и рассуждения продолжаются.

Еще раз выполняется проверка всех правил, но ни в одном правиле в условной части не упоминается уровень цен на бирже, и на этом рассуждения заканчиваются. Клиенту можно сказать следующее: «Когда обменный курс доллара падает, растут процентные ставки и уровень цен на бирже падает».

В реальной жизни такое заключение потребовало бы более сложных правил, однако система, реализующая прямую цепочку рассуждений (впрочем, как и другие системы), оперирует только теми данными, которые есть в базе знаний.

Рассмотренный пример иллюстрирует работу типичной системы прямых рассуждений:

1. Система содержит описание ряда ситуаций.

2. Для каждой ситуации система ищет в базе знаний правила, в условной части которых содержится соответствующее условие.

3. В соответствии с констатирующей частью (частью ТО) каждое правило может генерировать новые ситуации, которые добавляются к уже существующим.

4. Система обрабатывает каждую вновь сгенерированную ситуацию. При наличии хотя бы одной такой ситуации выполняются действия, начиная с пункта 2. Рассуждения заканчиваются, когда больше нет необработанных ситуаций.

База знаний

Покажем на примере, как эксперт может вводить правила непосредственно в базу знаний и использовать их для принятия решений во время биржевых операций. Все необходимые для работы переменные сведем в таблицу (см. рис. 5.1). Применяя содержащиеся в таблице переменные, относящиеся к фондовой бирже, правила можно записать следующим образом:

10 ЕСЛИ INTEREST=ПАДАЕТ, ТО STOCK=РАСТЕТ
20 ЕСЛИ INTEREST=РАСТЕТ, ТО STOCK=ПАДАЕТ
30 ЕСЛИ DOLLAR=ПАДАЕТ, ТО INTEREST=РАСТЕТ
40 ЕСЛИ DOLLAR=РАСТЕТ, ТО INTEREST=ПАДАЕТ
50 ЕСЛИ FEDINT=ПАДАЕТ И FEDMON=ДОБАВИТЬ, ТО INTEREST=ПАДАЕТ

Прежде чем продолжить, отметим, что приведенных выше правил явно недостаточно для исчерпывающего анализа состояния биржи, они просто используются для демонстрации основных принципов работы. Необходимо также помнить, что экспертная система наследует все достоинства и недостатки экспертов, заполняющих базу знаний.

Имя переменной	Значение
INTEREST	Изменение процентных ставок (рост или падение)
DOLLAR	Валютный курс доллара
FEDINT	Процентные ставки федерального резерва
FEDMON	Обращение денег федерального резерва (то есть добавление или изъятие резервов)
STOCK	Изменение уровня цен на бирже

Рис. 5.1. Таблица имен переменных

Работа с базой знаний

После того как база знаний создана, с ней можно работать. Для работы нужно построить еще несколько полезных для реше-

ния поставленной задачи таблиц. По сути, эти таблицы просто одна из форм представления баз знаний. Ниже все используемые структуры данных описаны подробно. На рис. 5.2 показана база знаний со всеми входящими в нее структурами данных.

Во-первых, база знаний содержит список переменных условия, то есть перечень переменных, входящих в условную часть каждого правила. Список этих переменных приводится на рис. 5.3. Числа (от 1 до 18), расположенные слева от имен переменных, — это индексы элементов массива, содержащего эти имена. Для каждого правила резервируется 4 элемента массива. Элементы массива, не используемые правилом, остаются пустыми.

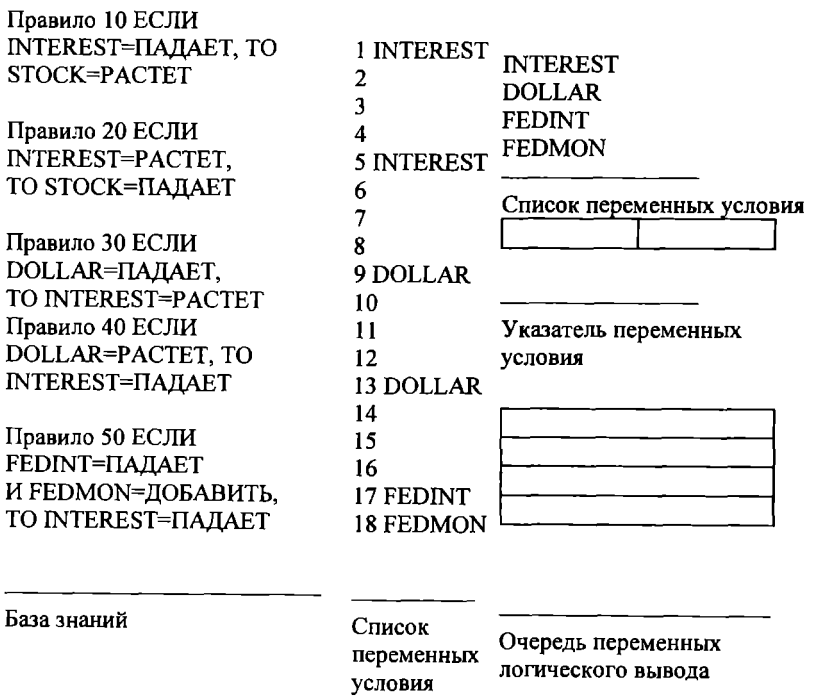


Рис. 5.2. Структуры данных базы знаний экспертной системы фондовой биржи

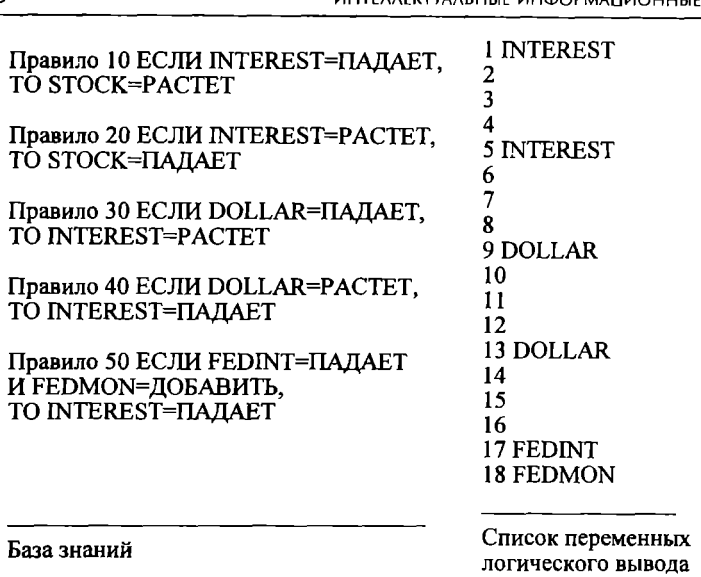


Рис. 5.3. Список переменных условия

Поясним использование списка переменных условия на примере. Предположим, необходимо выяснить влияние на биржу падения курса доллара. Падение курса доллара можно выразить следующим образом:

DOLLAR=ПАДАЕТ

Переменная DOLLAR содержится в условной части правил 30 и 40. Она помещается в структуру данных, называемую очередью переменных логического вывода (см. рис. 5.4). Поясним назначение очереди переменных вывода. В быту очередь может быть, например, в магазине. Принцип работы очереди — «первым пришел — первого обслужили» Покупатели встают в конец очереди, очередь продвигается вперед, их обслуживают, и они покидают очередь. Почему же структура данных для хранения переменных логического вывода называется очередью? Возьмем, например, переменную DOLLAR и посмотрим, в каком из правил, содержащих эту переменную, есть условие DOLLAR=ПАДАЕТ. Такое условие есть в правиле 30:

30 ЕСЛИ DOLLAR=ПАДАЕТ, ТО INTEREST=РАСТЕТ

срабатывает часть ТО этого правила, и текущей становится новая переменная условия INTEREST. После того как отработаны все правила, содержащие переменную DOLLAR, надо посмотреть, к чему приведет повышение процентных ставок. Для этого придется работать с новой переменной INTEREST, которая и помещается в очередь переменных логического вывода после переменной DOLLAR (см. рис. 5.5).

Правило 10 ЕСЛИ INTEREST=ПАДАЕТ,
ТО STOCK=РАСТЕТ
Правило 20 ЕСЛИ INTEREST=РАСТЕТ,
ТО STOCK=ПАДАЕТ
Правило 30 ЕСЛИ DOLLAR=ПАДАЕТ,
ТО INTEREST=РАСТЕТ
Правило 40 ЕСЛИ DOLLAR=РАСТЕТ,
ТО INTEREST=ПАДАЕТ
Правило 50 ЕСЛИ FEDINT=ПАДАЕТ
И FEDMON=ДОБАВИТЬ,
ТО INTEREST=ПАДАЕТ

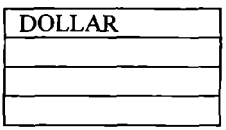
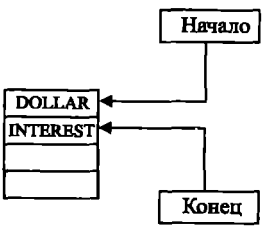


Рис. 5.4. Очередь переменных логического вывода

Переменная DOLLAR после того, как отработаны все правила, содержащие ее в условной части, удаляется из очереди переменных логического вывода (см. рис. 5.6).

Правило 10 ЕСЛИ INTEREST=ПАДАЕТ,
ТО STOCK=РАСТЕТ
Правило 20 ЕСЛИ INTEREST=РАСТЕТ,
ТО STOCK=ПАДАЕТ
Правило 30 ЕСЛИ DOLLAR=ПАДАЕТ,
ТО INTEREST=РАСТЕТ
Правило 40 ЕСЛИ DOLLAR=РАСТЕТ,
ТО INTEREST=ПАДАЕТ
Правило 50 ЕСЛИ FEDINT=ПАДАЕТ
И FEDMON=ДОБАВИТЬ,
ТО INTEREST=ПАДАЕТ



База знаний

Очередь переменных логического вывода

Рис. 5.5. Очередь переменных логического вывода

Выяснив последствия падения курса доллара, посмотрим, что же произойдет при повышении процентных ставок. Для этого из начала очереди выбирается переменная INTEREST, которая обрабатывается аналогично переменной DOLLAR. Когда очередь переменных логического вывода опустеет, прямая цепочка рассуждений закончится, и задача будет решена.

Рассмотрим назначение списка переменных и указателя списка переменных условия. Список переменных содержит значения переменных и признак их инициализации. До начала диалога с программой признак инициализации равен NI и всем переменным присвоены пустые значения (см. рис. 5.7). В процессе диалога с системой переменным присваиваются значения, а значение признака меняется на I. На рис. 5.8 показан список переменных после того, как переменным DOLLAR и INTEREST присвоены значения.

В указателе переменных условия хранится информация о правиле, с которым система работает в данное время. Указатель состоит из номера правила и номера условия в правиле, поскольку условная часть правила в общем случае может содержать несколько условий. Система использует указатель, чтобы отследить текущее положение в цепочке рассуждений.

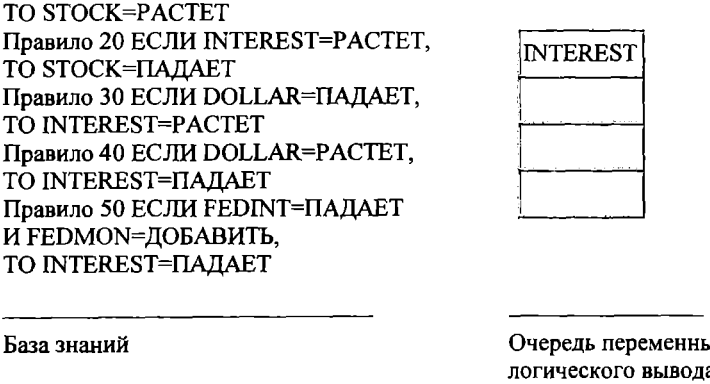


Рис. 5.6. Очередь переменных логического вывода

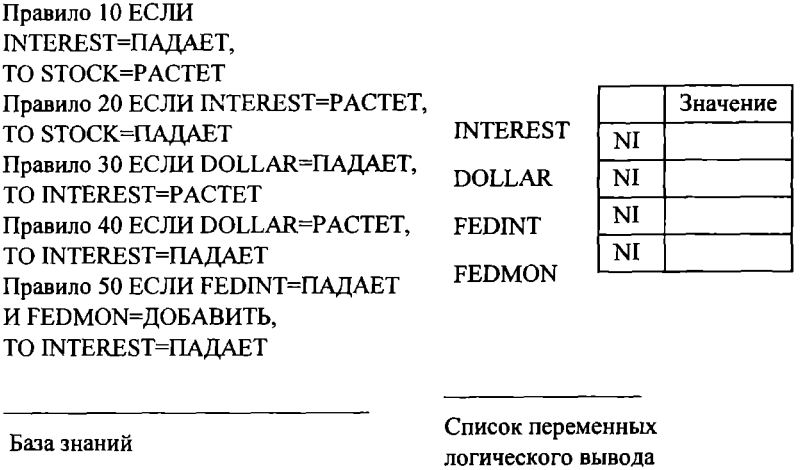


Рис. 5.7. Список переменных

Помните, работа с условной частью правила начинается с выбора переменной из очереди переменных логического вывода. Предположим, что в начале очереди стоит переменная DOLLAR, то есть надо определить последствия изменения курса доллара. Правило 30 — первое, в условной части которого содержится переменная DOLLAR, оно и включается в работу. Указатель переменных условия устанавливается на правило 30 (см. рис. 5.9).

В действительности указатель устанавливается на определенную запись в списке переменных условия. В этой записи имеется текущая переменная условия (в данном случае DOLLAR), стоящая в начале очереди переменных логического вывода.

Прежде чем перейти к примеру, обобщающему работу со всеми перечисленными структурами данных, посмотрим на них с другой точки зрения. На протяжении всего учебного пособия говорится, что искусственный интеллект — это имитация мыслительного процесса человека, решающего определенные задачи.

Правило 10 ЕСЛИ
INTEREST=ПАДАЕТ,
ТО STOCK=РАСТЕТ
Правило 20 ЕСЛИ
INTEREST=РАСТЕТ,
ТО STOCK=ПАДАЕТ
Правило 30 ЕСЛИ DOLLAR=ПАДАЕТ,
ТО INTEREST=РАСТЕТ
Правило 40 ЕСЛИ DOLLAR=РАСТЕТ,
ТО INTEREST=ПАДАЕТ
Правило 50 ЕСЛИ FEDINT=ПАДАЕТ
И FEDMON=ДОБАВИТЬ,
ТО INTEREST=ПАДАЕТ

INTEREST
DOLLAR
FEDINT
FEDMON

	Значение
I	РАСТЕТ
I	ПАДАЕТ
NI	
NI	

База знаний

Список переменных логического вывода

Рис. 5.8. Список переменных

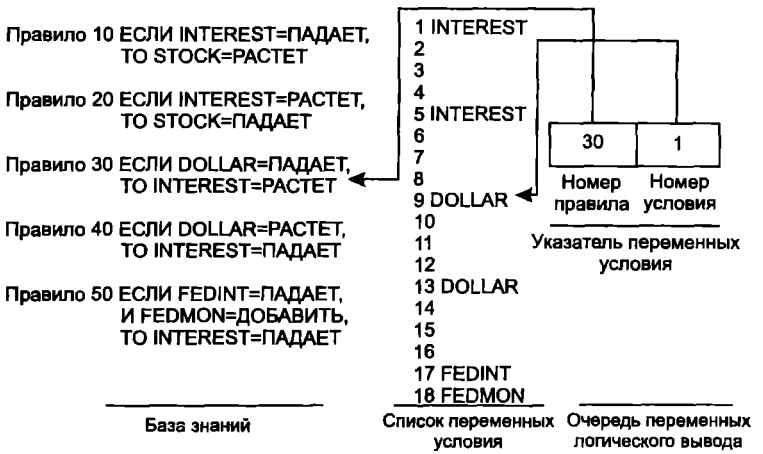


Рис. 5.9. Указатель переменных условия

Соотнесем введенные структуры данных с особенностями, присущими человеческому мышлению.

Человек, решая задачу, которая заключается в прогнозировании последствий некоторых ситуаций, прежде всего определяет

все условия, способные повлиять на конечный результат (список переменных). Затем всем переменным, о которых есть информация, присваиваются значения, то есть задаются исходные условия (например DOLLAR=ПАДАЕТ). Из всех правил, имеющих отношение к задаче, выбираются правила с исходными условиями (список переменных условия). Человек формирует в уме из переменных условия очередь переменных вывода и, перебирая соответствующие правила, обдумывает возможные логические выводы (указатель переменных условия).

Конечно, человек выполняет все эти действия автоматически, очень быстро и не так прямолинейно. Очень часто он даже не осознает, что обрабатывает информацию и делает необходимые выводы. Например, когда человек видит, что показатель нагрева двигателя в автомобиле приближается к максимальной отметке, он немедленно делает вывод о том, что мотор может заглохнуть. Причина быстрой работы человеческого мозга — одновременная работа его нейронов. Используя описанные структуры данных и алгоритм прямой цепочки рассуждений, на компьютере также можно быстро решать сложные задачи, содержащие большое количество переменных. Экспертная система, в которой продуманные структуры данных сочетаются с эффективным алгоритмом логического вывода, вполне способна делать осмысленные заключения.

Пример прямой цепочки рассуждений

Для примера воспользуемся базой знаний, разработанной для фондовой биржи, и спросим систему прямых рассуждений:

Что будет, если добавить в банковскую систему средства из Федерального резервного банка?

Другими словами, FEDMON=ДОБАВИТЬ

Система обратится к базе знаний, содержащей списки, указатели, очереди и расположенной на внешнем носителе (см. рис. 5.10). Поскольку значение переменной FEDMON изначально задано, она помещается в очередь переменных логического вывода. Просмотрев список переменных условия, система определяет, что первый раз переменная FEDMON встречается в условной части

правила 50. Указатель переменных условия устанавливается на первое условие правила 50. В списке переменных условия для правила 50 содержатся имена двух переменных FEDINT и FEDMON. Обратившись к списку переменных, система определяет, что переменной FEDMON присвоено значение ДОБАВИТЬ, а значение переменной FEDINT не задано (признак инициализации равен NI). Система запросит у пользователя значение переменной FEDINT:

Растет или падает уровень процентных ставок Федерального резервного банка?

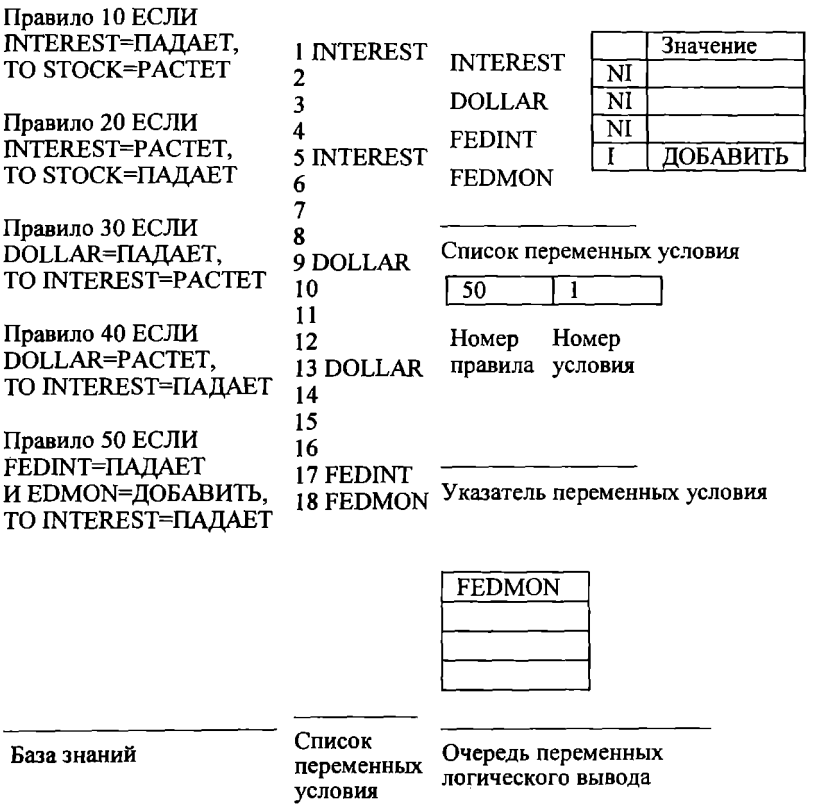


Рис. 5.10. Система, реализующая прямую цепочку рассуждений

Предположим, что пользователь ответил: «ПАДАЕТ»; это значение присваивается переменной FEDINT и заносится в список переменных. Номер условия в указателе переменных условия увеличивается и становится равным 2. Теперь все переменные условной части правила 50 проинициализированы, и система может приступить к ее анализу.

Истинны оба условия: и FEDINT=ПАДАЕТ и FEDMON= =ДОБАВИТЬ, поэтому согласно части ТО правила переменной INTEREST присваивается значение ПАДАЕТ. Переменная INTEREST помещается в очередь переменных логического вывода, поскольку она может стать новым условием (см. рис. 5.11), и соответствующим образом изменяется список переменных (см. рис. 5.12).

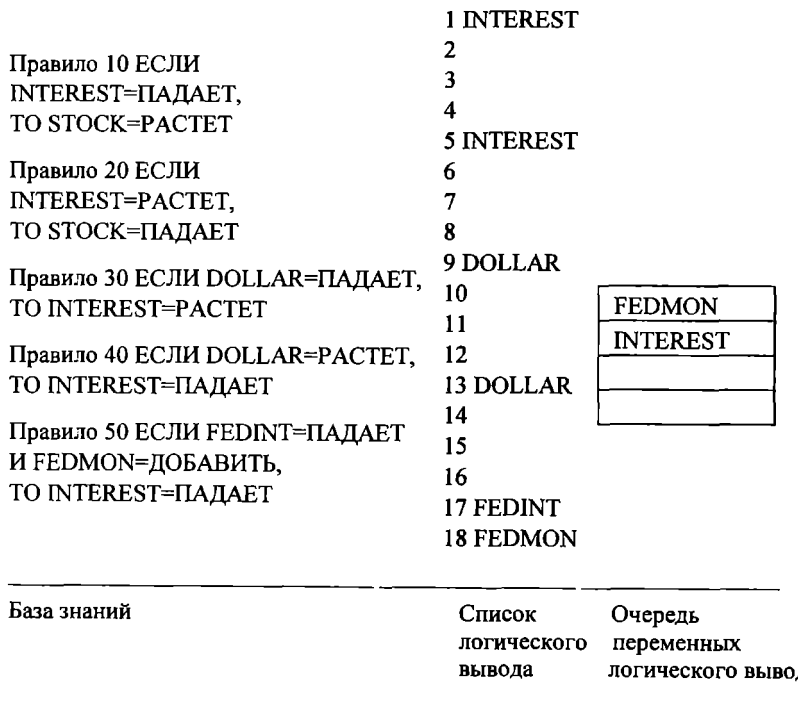


Рис. 5.11. Очередь переменных логического вывода

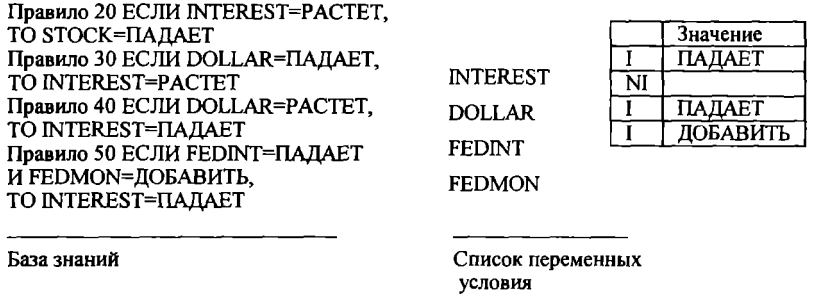


Рис. 5.12. Список переменных

Затем система проверяет, нет ли в остальных правилах переменной FEDMON. Если переменная FEDMON не встречается ни в одном из оставшихся правил, она удаляется из очереди переменных логического вывода. Первой в очереди становится переменная INTEREST. Заново анализируя список переменных условия, система находит переменную INTEREST в условной части правила 10. Состояние структур данных в этот момент показано на рис. 5.13.

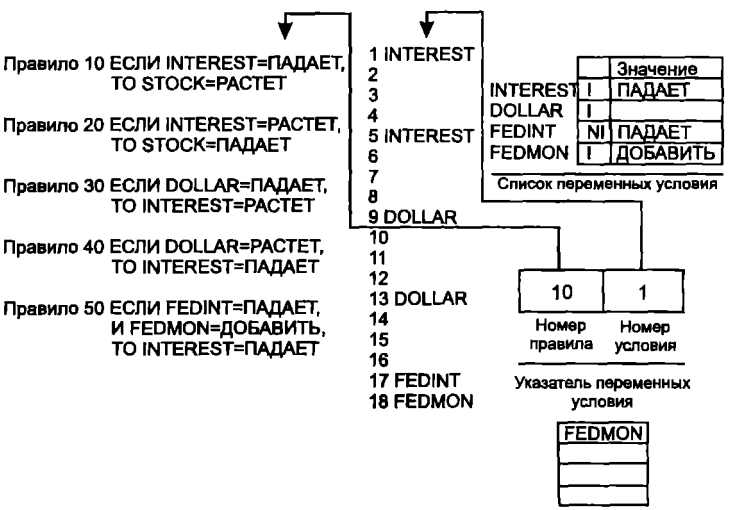


Рис. 5.13. Состояние структур данных системы

Переменная INTEREST входит в условную часть правил 10 и 20:

10 ЕСЛИ INTEREST=ПАДАЕТ, ТО STOCK=РАСТЕТ

20 ЕСЛИ INTEREST=РАСТЕТ, ТО STOCK=ПАДАЕТ

но логически выведенному значению переменной INTEREST= =ПАДАЕТ удовлетворяет только условие правила 10. Система обращается к части ТО этого правила и присваивает переменной STOCK значение РАСТЕТ, а имя переменной заносится в очередь переменных логического вывода (см. рис. 5.14). Правило 20 отвергнуто, потому что оно содержит условие INTEREST=РАСТЕТ, а переменная INTEREST уже имеет значение ПАДАЕТ. Переменная INTEREST не встречается больше ни в одном из оставшихся правил и поэтому удаляется из очереди переменных логического вывода.

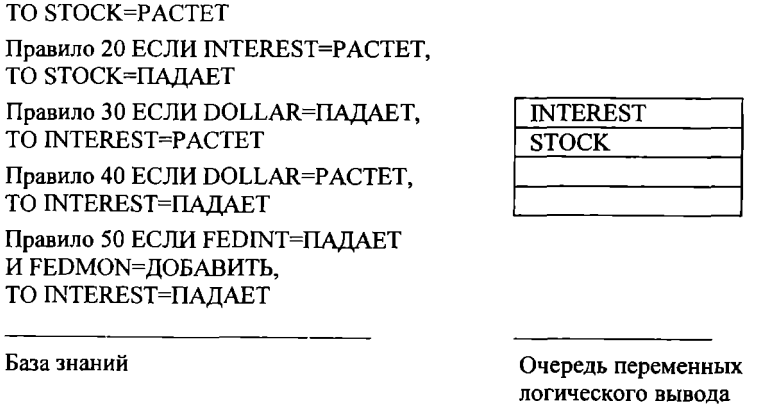


Рис. 5.14. Очередь переменных логического вывода

Переменная STOCK оказывается в начале очереди (см. рис. 5.15). Процесс рассуждений на этом заканчивается, поскольку переменная STOCK не входит в условную часть какого-либо правила. Ответ на поставленный вопрос будет выглядеть следующим образом:

1. Процентные ставки падают.
2. Уровень цен на бирже растет.

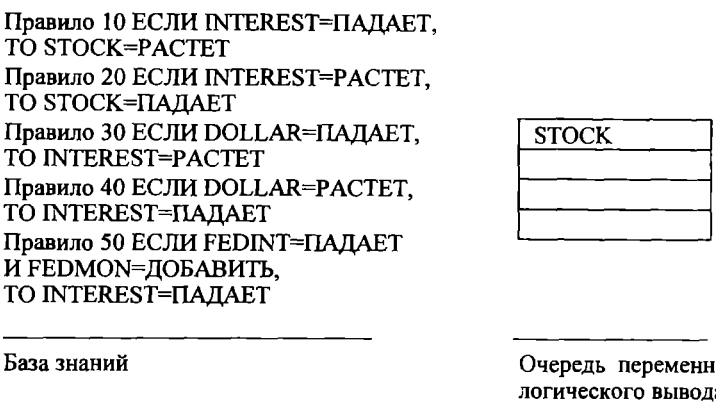


Рис. 5.15. Очередь переменных логического вывода

Обобщенный алгоритм работы системы

Обобщенный алгоритм работы системы, реализующей прямую цепочку рассуждений, можно свести к следующему:

1. Определить исходное состояние.

2. Занести переменную условия в очередь переменных логического вывода, а ее значение — в список переменных.

3. Просмотреть список переменных и найти ту переменную, имя которой стоит в начале очереди переменных логического вывода. Если переменная найдена, записать в указатель переменных условия номер правила и число 1. Если переменная не найдена, перейти к шагу 6.

4. Присвоить значения непроинициализированным переменным условной части найденного правила (если такие есть). Имена переменных содержатся в списке переменных условия. Проверить все условия правила и в случае их истинности обратиться к части ТО правила.

5. Присвоить значение переменной, входящей в часть ТО правила, и поместить ее в конец очереди переменных логического вывода.

6. Удалить переменную, стоящую в начале очереди переменных логического вывода, если она больше не встречается в условной части какого-либо правила.

7. Закончить процесс рассуждений, как только опустеет очередь переменных логического вывода. Если же в очереди еще есть переменные, вернуться к шагу 3.

Этим алгоритмом можно воспользоваться для программирования на БЕЙСИКе простой системы, реализующей прямую цепочку рассуждений.

Пример программы

Программа (листинг 5.1), приведенная в качестве примера, может работать с любой базой знаний. Нужные правила можно вставить между предложениями 1500 и 1610, а саму программу запустить на выполнение командой RUN.

Пояснение к программе

Программа работает с уже известными правилами фондовой биржи. Так, правило:

ЕСЛИ процентные ставки падают,
ТО уровень цен на бирже растет

запрограммировано предложениями программы 1500–1510.

А правило

ЕСЛИ валютный курс доллара падает,
ТО процентные ставки растут —

предложениями 1540–1550.

Список переменных формируется в предложении 367. Они представляют собой переменные условия, с которых начинается рассуждение. Список переменных условия формируется в предложениях 407 и 408. В программе используется ряд сокращений, например ST=ПАДАЕТ означает, что уровень цен на бирже падает, IN=РАСТЕТ означает, что процентные ставки растут.

Листинг программы

Листинг 5.1

```
1 REM ***ПРЯМАЯ ЦЕПОЧКА РАССУЖДЕНИЙ***
2 REM ***
10 REM УСЛОВНЫЕ ЧАСТИ ДОПОЛНИТЕЛЬНЫХ ПРАВИЛ
11 REM ВСТАВИТЬ ПОСЛЕ ПРЕДЛОЖЕНИЯ 1500
20 REM НАПРИМЕР: 1500 IF (A1=2) AND (A6=6) OR (A3$=«YES») THEN S=1
22 REM 1502 RETURN
```

```
30 REM 1520 IF (JS=«НЕТ») THEN S=1
32 REM 1522 RETURN
40 REM ЧАСТЬ ТО ЭТИХ ПРАВИЛ ВСЕГДА СОДЕРЖИТ: THEN S=1
50 REM НАЧИНАЯ С ПРЕДЛОЖЕНИЯ 1510 МОЖНО ВСТАВИТЬ ЧАСТИ ТО
52 REM ПРАВИЛ ПОЛЬЗОВАТЕЛЯ
200 REM ***БЛОК ИНИЦИАЛИЗАЦИИ***
210 REM V$ — СПИСОК ПЕРЕМЕННЫХ, IN — ПРИЗНАКИ ИНИЦИАЛИЗАЦИИ
230 REM CV$ — ПЕРЕМЕННЫЕ УСЛОВИЯ И ЛОГИЧЕСКОГО ВЫВОДА
240 DIM Q$ (10), V$ (10), CV$ (40), IN (10)
255 FP=1: BP=1: REM FP и BP=УКАЗАТЕЛИ НАЧАЛА И КОНЦА ОЧЕРЕДИ
260 FOR I=1 TO 10
270 Q$ (I)=«»: V$ (I)=«»: IN (I)=«»: SS (I)=0: CS (I)=0
280 NEXT I
340 REM ВВОД ПЕРЕМЕННЫХ, ВХОДЯЩИХ В УСЛОВНЫЕ ЧАСТИ ПРАВИЛ В
350 REM НУЖНОМ ПОРЯДКЕ (НЕ БОЛЬШЕ 3 ПЕРЕМЕННЫХ НА ПРАВИЛО)
360 REM ИМЕНА ПЕРЕМЕННЫХ НЕ ДОЛЖНЫ ПОВТОРЯТЬСЯ
365 REM КОГДА ВВЕДЕНЫ ВСЕ ИМЕНА, НАДО НАЖАТЬ КЛАВИШУ ВК
367 V$ (1)=«DO»: V$ (2)=«FT»: V$ (3)=«FM»: V$ (4)=TN»: V$ (5)=«ST»
368 PRINT ***«СПИСОК ПЕРЕМЕННЫХ»***
370 FOR I=1 TO 10
380 PRINT «ВВЕДИТЕ ПЕРЕМЕННУЮ»; I
385 PRINT V$ (I)
395 NEXT I
396 INPUT «ДЛЯ ПРОДОЛЖЕНИЯ НАЖМИТЕ»; I
400 REM ПЕРЕМЕННЫЕ ВВОДЯТСЯ В ПОРЯДКЕ ИХ ПОЯВЛЕНИЯ В ЧАСТЯХ
401 REM ЕСЛИ ПРАВИЛ
```

```
405 REM В ОДНОМ ПРАВИЛЕ МОЖНО ЗАДАТЬ НЕ БОЛЬШЕ
3 ПЕРЕМЕННЫХ
406 REM КОГДА ВВЕДЕНЫ ВСЕ ПЕРЕМЕННЫЕ, НАЖАТЬ КЛА-
ВИШУ ВК
407 CV$ (1)=«IN»: CVS (5)=«IN»: CV$ (9)=«DO»: CV$ (13)=«DO»
408 CV$ (17)=«FT»: CV$ (18)=«FM»
409 PRINT «***СПИСОК ПЕРЕМЕННЫХ УСЛОВИЯ***»
410 FOR I=1 TO 8: PRINT «УСЛОВИЕ»; I
420 FOR J=1 TO 4: PRINT «ПЕРЕМЕННАЯ»; I
423 K=4*(I–1)+J
424 PRINT CV$ (K)
430 NEXT J
432 IF I=4 THEN INPUT «ДЛЯ ПРОДОЛЖЕНИЯ НАЖМИТЕ ВК»; Z
435 NEXT I
450 REM ***БЛОК ЛОГИЧЕСКОГО ВЫВОДА***
465 REM INPUT «ВВЕДИТЕ ПЕРЕМЕННУЮ ЛОГИЧЕСКОГО
ВЫВОДА»; C$
470 REM ЗАМЕНА ПЕРЕМЕННОЙ УСЛОВИЯ (C$) В ОЧЕРЕДИ
ПЕРЕМЕННЫХ ВЫВОДА (Q$)
475 Q$ (BP)=C$: BP=BP+1
476 REM ПЕРЕДВИНУТЬ НАЗАД УКАЗАТЕЛЬ КОНЦА ОЧЕРЕДИ
480 REM УСТАНОВИТЬ УКАЗАТЕЛЬ ПЕРЕМЕННЫХ УСЛОВИЯ,
СОСТОЯЩИЙ
482 REM ИЗ НОМЕРА ПРЕДЛОЖЕНИЯ (SN) И НОМЕРА УСЛО-
ВИЯ (CN)
485 SN=1: CN=1
490 REM ПОИСК НОМЕРА СЛЕДУЮЩЕГО ПРАВИЛА, СОДЕР-
ЖАЩЕГО
491 REM ПЕРЕМЕННУЮ УСЛОВИЯ, КОТОРАЯ
492 REM НАХОДИТСЯ В НАЧАЛЕ ОЧЕРЕДИ [Q$ (FP)]. НОМЕР
ПРАВИЛА
494 REM РАСПОЛОЖЕН В СПИСКЕ ПЕРЕМЕННЫХ УСЛОВИЯ
(CVS)
495 F=1: REM ПОВТОРИТЬ СНАЧАЛА
496 GOSUB 3000: CN=1: KEM УКАЗЫВАЕТ НА ПЕРВОЕ УСЛО-
ВИЕ ПРАВИЛА
```

```
500 IF SN=O THEN GOSUB
700 REM ПРАВИЛ БОЛЬШЕ НЕТ
505 I=4*(SN-1)+CN: REM АДРЕС УСЛОВИЯ ПРАВИЛА
510 V$=CV$ (I): REM ПЕРЕМЕННАЯ УСЛОВИЯ
560 REM ПРАВИЛО СОДЕРЖИТ ДРУГИЕ УСЛОВИЯ?
565 IF V$=« » THEN GOTO
580 REM В ПРАВИЛЕ УСЛОВИЙ БОЛЬШЕ НЕТ
570 GOSUB 2400
571 REM ПРОВЕРИТЬ, ПРИСВОЕНО ЛИ ЗНАЧЕНИЕ ПЕРЕМЕННОЙ
575 CN=CN+1: GOTO
505 REM ПРОВЕРИТЬ СЛЕДУЩЕЕ УСЛОВИЕ ПРАВИЛА
580 REM УСЛОВИЙ В ПРАВИЛЕ БОЛЬШЕ НЕТ
585 S=0: ON SN GOSUB 1500, 1520, 1540, 1560, 1580, 1600
590 REM ВЫПОЛНИТЬ ЧАСТЬ ТО ПРАВИЛА, ТО ЕСТЬ S=1
600 IF S=1 THEN GOTO
630 REM ПЕРЕХОД К ЧАСТИ ТО ПРАВИЛА
605 REM ВЫБОР СЛЕДУЮЩЕГО ЭЛЕМЕНТА ИЗ
607 REM СПИСКА ПЕРЕМЕННЫХ УСЛОВИЯ
610 F=SN+1; GOTO 496
630 REM ЕСЛИ УСЛОВИЕ ИСТИННО — ВЫПОЛНИТЬ ЧАСТЬ ТО ПРАВИЛА
635 ON SN GOSUB 1510, 1530, 1550, 1570, 1590, 1610
650 F=SN+1: GOTO
496 REM ПРАВИЛ БОЛЬШЕ НЕТ
700 REM СПИСОК ПЕРЕМЕННЫХ УСЛОВИЯ ПУСТ (CV$)
705 REM ЕСЛИ В НАЧАЛЕ ОЧЕРЕДИ [Q$ (FR)] ЕСТЬ ПЕРЕМЕННАЯ — ОНА
710 REM УДАЛЯЕТСЯ И НА ЕЕ МЕСТО ПОМЕЩАЕТСЯ СЛЕДУЮЩАЯ
712 REM ПЕРЕМЕННАЯ [Q$ (EP+1)].
720 REM ЕСЛИ В ОЧЕРЕДИ НЕТ ПЕРЕМЕННЫХ, ЗАКОНЧИТЬ ПРОГРАММУ
725 FP=FP+1: REM СЛЕДУЮЩАЯ ПЕРЕМЕННАЯ ИЗ ОЧЕРЕДИ
730 IF FP=BP THEN STOP
731 REM ОЧЕРЕДЬ ПЕРЕМЕННЫХ ВЫВОДА ПУСТА — КОНЕЦ
```

```
735 GOTO 495 REM ПРОВЕРКА ТЕКУЩЕЙ ПЕРЕМЕННОЙ УСЛОВИЯ
1490 REM ***УСЛОВНЫЕ ЧАСТИ ПРАВИЛ НАЧИНАЮТСЯ С 1500***
1492 REM ПРИМЕРЫ ПРАВИЛ БАЗЫ ЗНАНИЙ ФОНДОВОЙ БИРЖИ
1494 REM ДЛЯ ТОГО ЧТОБЫ ПРИМЕР БЫЛ СОВМЕСТИМ СО ВСЕМИ
1496 REM ВЕРСИЯМИ БЕЙСИКА ИСПОЛЬЗУЮТСЯ ИМЕНА ПЕРЕМЕННЫХ
1497 REM СОСТОЯЩИЕ ИЗ ДВУХ БУКВ
1498 REM DOLLAR — DO, STOCK — ST, FEDINT — FT, FEDMON — FM
1499 REM ***ПРАВИЛО 1
1500 IF IN$=«ПАДАЕТ» THEN S=1
1502 RETURN
1505 REM ПЕРЕМЕННАЯ ПОМЕЩАЕТСЯ В ОЧЕРЕДЬ ПЕРЕМЕННЫХ ВЫВОДА
1510 ST$=«РАСТЕТ»: PRINT «ST=РАСТЕТ»: V$=«ST»: GOSUB 3100: RETURN
1519 REM ***ПРАВИЛО 2
1520 IF IN$=«РАСТЕТ» THEN S=1
1522 RETURN
1530 ST$=«ПАДАЕТ»: PRINT «ST=ПАДАЕТ»: GOSUB 3100: RETURN
1539 REM ***ПРАВИЛО 3
1540 IF DO$=«ПАДАЕТ» THEN S=1
1542 RETURN
1545 REM ПОМЕСТИТЬ ПЕРЕМЕННУЮ В ОЧЕРЕДЬ ПЕРЕМЕННЫХ ВЫВОДА
1550 IN$=«РАСТЕТ»: PRINT «IN=РАСТЕТ»: V$=«IN»: GOSUB 3100: RETURN
1559 REM ***ПРАВИЛО 4
1560 IF DOS=«РАСТЕТ» THEN S=1
1562 RETURN
1565 REM ПОМЕСТИТЬ ПЕРЕМЕННУЮ В ОЧЕРЕДЬ ПЕРЕМЕННЫХ ВЫВОДА
```

```
1570 IN$=«ПАДАЕТ»: PRINT «IN=ПАДАЕТ»: V$=IN»: GOSUB
3100: RETURN
1579 REM ***ПРАВИЛО 5
1580 IF (FT$=«ПАДАЕТ») AND (FM$=«ДОБАВИТЬ») THEN S=1
1582 RETURN
1585 REM ПОМЕСТИТЬ ПЕРЕМЕННУЮ В ОЧЕРЕДЬ ВЫВОДА
1590 IN$=«ПАДАЕТ»: PRINT «IN=ПАДАЕТ»: V$=«IN»: GOSUB
3100: RETURN
1599 REM ***ПРАВИЛО 6
1600 IF (QU$=«ДА») AND (GR 3.5) THEN S=1
1602 RETURN
1610 PO$=«ДА»: PRINT «PO=ДА»: RETURN
1611 REM ***КОНЕЦ БАЗЫ ЗНАНИЙ***
1690 REM ***МЕТКИ 1700 РАСПОЛОЖЕНЫ ПРЕДЛОЖЕНИЯ
ВВОДА***
1699 REM ПРЕДЛОЖЕНИЯ ВВОДА ДЛЯ БАЗЫ ЗНАНИЙ ПРИ-
МЕРА
1705 INPUT «ПРОЦЕНТНЫЕ СТАВКИ РЕЗЕРВА ПАДАЮТ ИЛИ
РАСТУТ»; FT$: RETURN
1710 INPUT «ДОБАВИТЬ ИЛИ ИЗЪЯТЬ РЕЗЕРВ»; FM$: RETURN
1715 INPUT «ПРОЦЕНТНЫЕ СТАВКИ ПАДАЮТ ИЛИ РАСТУТ»;
IN$
1720 INPUT «УРОВЕНЬ ЦЕН ПАДАЕТ ИЛИ РАСТЕТ»; ST$:
RETURN
1750 REM ***КОНЕЦ БЛОКА ВВОДА В БАЗУ ЗНАНИЙ***
1999 REM ***БЛОК ПОДПРОГРАММ***
2400 REM ПОДПРОГРАММА ПРИСВАИВАЕТ ЗНАЧЕНИЯ ПЕРЕ-
МЕННОЙ (V$)
2402 REM IN — ПРИЗНАК ИНИЦИАЛИЗАЦИИ ПЕРЕМЕННЫХ
2404 REM IN=0 — ПЕРЕМЕННОЙ НЕ ПРИСВОЕНО ЗНАЧЕНИЕ
2405 REM IN=1 — ПРИСВОЕНО
2405 REM СПИСОК ПЕРЕМЕННЫХ (VL$) СОДЕРЖИТ ПЕРЕ-
МЕННУЮ (V$)
2420 FOR I=1 TO 10: REM ВЫБОР ПЕРЕМЕННОЙ ИЗ СПИСКА
2430 IF V$=V$ (I) THEN GOTO 2445: REM ПЕРЕМЕННАЯ НАЙ-
ДЕНА
```

```
2440 NEXT I
2445 IF IN (I)=1 THEN RETURN: REM ПЕРЕМЕННОЙ УЖЕ ПРИ-
СВОЕНО ЗНАЧЕНИЕ
2450 IF IN (I)=1 THEN RETURN: REM ПЕРЕМЕННОЙ УЖЕ ПРИ-
СВОЕНО ЗНАЧЕНИЕ
2460 REM ВВОД ЗНАЧЕНИЙ ПЕРЕМЕННЫХ НАЧИНАЕТСЯ
С МЕТКИ 1700
2464 REM ПОСЛЕ КАЖДОГО ПРЕДЛОЖЕНИЯ INPUT СТОИТ
RETURN
2465 IN (I)=1: REM ПРИЗНАК ТОГО, ЧТО ПЕРЕМЕННОЙ ПРИ-
СВОЕНО ЗНАЧЕНИЕ
2470 ON I GOSUB 1700, 1705, 1710, 1715, 1720
2480 RETURN
3000 REM ПОИСК В СПИСКЕ ПЕРЕМЕННЫХ УСЛОВИЯ (CV$)
ИМЕНИ ПЕРЕМЕННОЙ, СТОЯЩЕЙ
3005 REM В НАЧАЛЕ ОЧЕРЕДИ (Q$), ЕСЛИ ТАКОЙ ПЕРЕМЕН-
НОЙ НЕТ, ТО
3010 REM ВОЗВРАЩАЕТСЯ НОМЕР ПРЕДЛОЖЕНИЯ (SN), НА-
ПРИМЕР SN=0.
3015 REM ПОИСК НАЧИНАЕТСЯ С ПРЕДЛОЖЕНИЯ С НОМЕ-
РОМ F
3025 FOR SN=F TO 10
3030 FOR CN=1 TO 4
3035 K=(SN-1)*4+CN
3040 IF CV$ (K)=Q$ (FP) THEN RETURN
3045 NEXT CN
3050 NEXT SN, 3
055 SN=0: RETURN
3100 REM ПОДПРОГРАММА ПРИСВАИВАЕТ ЗНАЧЕНИЕ ПЕРЕ-
МЕННОЙ (V$) И ПОМЕЩАЕТ ЕЕ В
3105 REM ОЧЕРЕДЬ Q$ (BP), ЕСЛИ ЕЕ ТАМ НЕТ, 3
110 REM ПОИСК ПЕРЕМЕННОЙ В СПИСКЕ ПЕРЕМЕННЫХ
[V$ (10)]
3115 FOR I=l TO 10
3117 IF V$=V$ (I) THEN GOTO 3125
3120 NEXT I
```

```
3125 IN (I)=1: REM ПЕРЕМЕННОЙ ПРИСВОЕНО ЗНАЧЕНИЕ
3130 REM ОПРЕДЕЛИТЬ, ЕСТЬ ЛИ ПЕРЕМЕННАЯ V$ В ОЧЕРЕДИ (Q$)
3135 FOR I=1 TO 10: IF V$=Q$ (I) THEN RETURN
3140 NEXT I
3155 REM ПЕРЕМЕННОЙ НЕТ В ОЧЕРЕДИ, ПОЭТОМУ ОНА ТУДА ПОМЕЩАЕТСЯ
3160 Q$ (BP)=V$: BP=BP+1
3170 FOR I=1 TO 10
3176 NEXT I
3180 RETURN
```

Примеры выполнения программы

Программа на БЕЙСИКе выполняется по команде RUN. Ниже приводится результат выполнения программы.

```
RUN
***СПИСОК ПЕРЕМЕННЫХ***
ВВЕДИТЕ ПЕРЕМЕННУЮ 1 DO
ВВЕДИТЕ ПЕРЕМЕННУЮ 2 FT
ВВЕДИТЕ ПЕРЕМЕННУЮ 3 FM
ВВЕДИТЕ ПЕРЕМЕННУЮ 4 IN
ВВЕДИТЕ ПЕРЕМЕННУЮ 5 ST
ВВЕДИТЕ ПЕРЕМЕННУЮ 6
ВВЕДИТЕ ПЕРЕМЕННУЮ 7
ВВЕДИТЕ ПЕРЕМЕННУЮ 8
ВВЕДИТЕ ПЕРЕМЕННУЮ 9
ВВЕДИТЕ ПЕРЕМЕННУЮ 10
ДЛЯ ПРОДОЛЖЕНИЯ НАЖМИТЕ ВК
***СПИСОК ПЕРЕМЕННЫХ УСЛОВИЯ***
***УСЛОВИЕ 1
ПЕРЕМЕННАЯ 1 IN
ПЕРЕМЕННАЯ 2
ПЕРЕМЕННАЯ 3
ПЕРЕМЕННАЯ 4
***УСЛОВИЕ 2
ПЕРЕМЕННАЯ 1 IN
ПЕРЕМЕННАЯ 2
```

ПЕРЕМЕННАЯ 3
ПЕРЕМЕННАЯ 4
***УСЛОВИЕ 3
ПЕРЕМЕННАЯ 1 DO
ПЕРЕМЕННАЯ 2
ПЕРЕМЕННАЯ 3
ПЕРЕМЕННАЯ 4
***УСЛОВИЕ 4
ПЕРЕМЕННАЯ 1 DO
ПЕРЕМЕННАЯ 2
ПЕРЕМЕННАЯ 3
ПЕРЕМЕННАЯ 4
***УСЛОВИЕ 5
ПЕРЕМЕННАЯ 1 FT
ПЕРЕМЕННАЯ 2 FM
ПЕРЕМЕННАЯ 3
ПЕРЕМЕННАЯ 4
***УСЛОВИЕ 6
ПЕРЕМЕННАЯ 1
ПЕРЕМЕННАЯ 2
ПЕРЕМЕННАЯ 3
ПЕРЕМЕННАЯ 4
***УСЛОВИЕ 7
ПЕРЕМЕННАЯ 1
ПЕРЕМЕННАЯ 2
ПЕРЕМЕННАЯ 3
ПЕРЕМЕННАЯ 4
***УСЛОВИЕ 8
ПЕРЕМЕННАЯ 1
ПЕРЕМЕННАЯ 2
ПЕРЕМЕННАЯ 3
ПЕРЕМЕННАЯ 4

Пример диалога системы с пользователем
ВВЕДИТЕ УСЛОВНУЮ ПЕРЕМЕННУЮ: DO
КУРС ДОЛЛАРА РАСТЕТ ИЛИ ПАДАЕТ? РАСТЕТ

IN=ПАДАЕТ
ST=РАСТЕТ
Break In 730
Ok

ВВЕДИТЕ ПЕРЕМЕННУЮ УСЛОВИЯ: DO
КУРС ДОЛЛАРА РАСТЕТ ИЛИ ПАДАЕТ? ПАДАЕТ
IN=РАСТЕТ
ST=ПАДАЕТ
Break In 730
Ok

ВВЕДИТЕ ПЕРЕМЕННУЮ УСЛОВИЯ: IN
ПРОЦЕНТНАЯ СТАВКА РАСТЕТ ИЛИ ПАДАЕТ? ПАДАЕТ
ST=РАСТЕТ
Break In 730
Ok

ПРОЦЕНТНЫЕ СТАВКИ РЕЗЕРВА РАСТУТ ИЛИ ПАДАЮТ? РАСТУТ
ДОБАВИТЬ ИЛИ ИЗЪЯТЬ РЕЗЕРВ? ДОБАВИТЬ
Break In 730
Ok

Для последних приведенных условий нет выводов, поскольку им не удовлетворяет ни одно правило, то есть

ЕСЛИ FT=ПАДАЕТ,
FM=ДОБАВИТЬ,
ТО IN=ПАДАЕТ
ВВЕДИТЕ ПЕРЕМЕННУЮ УСЛОВИЯ: FM
ПРОЦЕНТНЫЕ СТАВКИ РЕЗЕРВА РАСТУТ ИЛИ ПАДАЮТ? ПАДАЮТ
IN=ПАДАЕТ
ST=РАСТЕТ
Break In 730
Ok

Вспомогательные таблицы

Для формирования новой базы знаний или модификации существующей полезно использовать приведенную ниже таблицу. Таблица упрощает проверку корректности отношений базы знаний и структур данных экспертной системы. В принципе можно создать специальный компилятор для обработки таких таблиц и автоматически преобразовывать базу знаний в нужные структуры данных. Но эта задача требует полного понимания механизма работы экспертной системы. С помощью таблицы

IF		=		AND
	Переменная 11	=	Значение 11	AND
	Переменная 12	=	Значение 12	
THEN	Переменная 13	=	Значение 13	
IF	Переменная 14	=	Значение 14	AND
	Переменная 21	=	Значение 21	AND
	Переменная 22	=	Значение 22	
THEN	Переменная 23	=	Значение 23	
IF	Переменная 24	=	Значение 24	AND
	Переменная 31	=	Значение 31	AND
	Переменная 32	=	Значение 32	
THEN	Переменная 33	=	Значение 33	
IF	Переменная 34	=	Значение 34	AND
	Переменная 41	=	Значение 41	AND
	Переменная 42	=	Значение 42	
THEN	Переменная 43	=	Значение 43	
IF	Переменная 44	=	Значение 44	AND
	Переменная 51	=	Значение 51	AND
	Переменная 52	=	Значение 52	
THEN	Переменная 53	=	Значение 53	
	Переменная 54	=	Значение 54	

проще создавать и преобразовывать переменные (переменная 11, переменная 12 и т. д.) и значения (значение 11, значение 12 и т. д.) новой базы знаний в предложения программы. В одном правиле можно использовать до трех переменных. Для того чтобы сформировать первую таблицу, необходимо записать правила в форме ЕСЛИ–ТО. Затем можно преобразовывать их в другие таблицы и получать фрагменты программы.

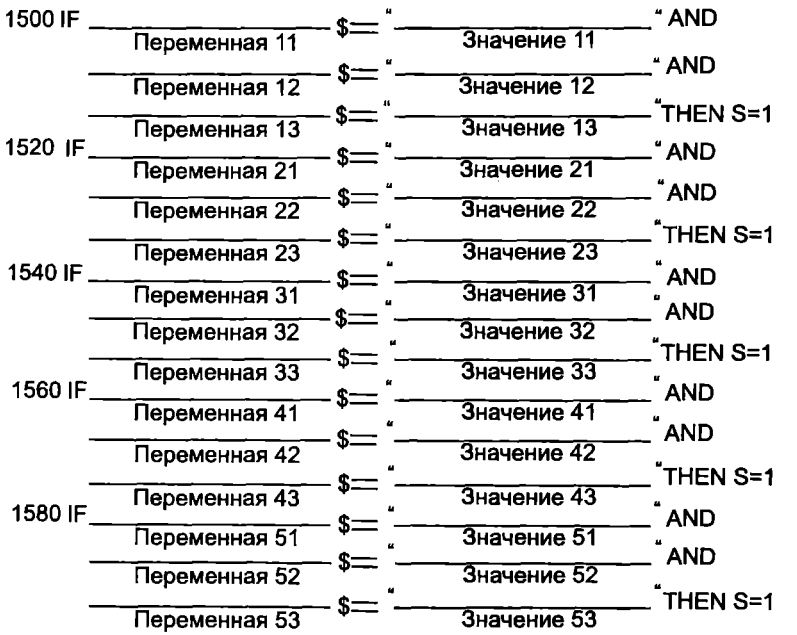
Ниже приводится рабочая таблица списка переменных условия (предложения 407 и 408 программы).

CV$(1)	=	Переменная 11
CV$(2)	=	Переменная 12
CV$(3)	=	Переменная 13
CV$(5)	=	Переменная 21
CV$(6)	=	Переменная 22
CV$(7)	=	Переменная 23
CV$(9)	=	Переменная 31
CV$(10)	=	Переменная 32
CV$(11)	=	Переменная 33
CV$(13)	=	Переменная 41
CV$(14)	=	Переменная 42
CV$(15)	=	Переменная 43
CV$(17)	=	Переменная 51
CV$(18)	=	Переменная 52
CV$(19)	=	Переменная 53

Теперь рассмотрим рабочую таблицу списка переменных (предложение 367 программы). Имена переменных должны быть уникальны.

Рабочая таблица правил, записанных на БЕЙСИКе (предложения 1490–1611 программы). Сначала приводится заготовка для условных частей правил, а потом для констатирующих.

Условные части (ЕСЛИ) правил:



Констатирующие части (ТО) правил:

VI. ОБРАТНАЯ ЦЕПОЧКА РАССУЖДЕНИЙ

Предположим, что автомобиль не тронулся с места. В чем же дело — сел аккумулятор или неисправен стартер? Рассмотрим задачу в более общем виде: по известному результату (автомобиль не тронулся с места) нужно определить условия, которые к нему привели, то есть по симптомам найти причины. Она отличается от задачи в предыдущем разделе (прямая цепочка рассуждений) тем, что там уже были известны условия (перегрев двигателя), но последствия, к которым они приведут, известны не были. Задача заключалась в предсказании возможного результата. Здесь же результат известен, и нужно найти вызвавшие его причины.

Для решения задачи опять понадобятся правила. Приведем несколько подходящих для выбранного примера правил.

Правило 1:

ЕСЛИ автомобиль не заводится И сел аккумулятор,
ТО не подается ток в стартер

Правило 2:

ЕСЛИ в стартер не подается ток,
ТО автомобиль не тронется с места

Каким же образом можно найти условия, при которых автомобиль не тронется с места? С помощью обратной цепочки рассуждений. Известный результат (автомобиль не трогается с места) повлечет за собой цепочку рассуждений, которая приведет нас к вызвавшим его причинам. Причины возникают раньше следствий, поэтому в процессе обратной цепочки рассуждений просматриваются логические выводы, ставятся условия, которые к ним привели, и определяется, связаны ли эти условия с предыдущими логическими выводами. Например, в приведенной зада-

че сначала надо обратиться ко второму правилу, поскольку содержащийся в нем логический вывод: «ТО автомобиль не тронется с места» соответствует реально возникшей ситуации. Обратная цепочка рассуждений всегда начинается со следствия (часть ТО правила). Причина, по которой автомобиль не трогается с места, содержится в условной части правила 2: «ЕСЛИ в стартер не подается ток». Рассуждения продолжаются, так как надо выяснить, почему же в стартер не подается ток. Ответ на этот вопрос дает правило 1. В условной части правила записано: «ЕСЛИ автомобиль не заводится И сел аккумулятор». Если эти условия выполняются, то можно выявить причину, по которой автомобиль не трогается с места. В противном случае придется проверить другие правила, относящиеся к этой ситуации, проследить еще одну цепочку.

Если в правилах, относящихся к проблемной области, не удается найти условную часть с выполняющимися условиями, необходимо обратиться к специалистам и запросить дополнительную информацию. Другими словами, если условные части всех входящих в систему ИИ правил имеют значение «ложь», то в систему надо добавить логические выводы, которые могут помочь при решении задачи.

Теперь понятно, что здесь слово «цепочка» означает процедуру логической связи ряда правил. В приведенном примере цепочка рассуждений начинается со второго правила и заканчивается на первом. Давайте посмотрим, почему же такая цепочка рассуждений называется обратной. До того как автомобиль не смог тронуться с места, он не завелся, и у него сел аккумулятор. Последствия выполнения некоторых условий известны — автомобиль не поехал, осталось определить эти условия, то есть найти причины, по которым автомобиль не тронулся с места. Обратной цепочка рассуждений называется потому, что начинается с уже происшедшего события и идет к его истокам. Программные средства, работающие по принципу обратной цепочки рассуждений, предназначены для поиска причин по уже известному результату. Ниже приведен пример создания базы знаний и обработки хранящейся в ней информации с использованием обратной цепочки рассуждений. Цепочка выполняется с помощью серии вопросов,

которые система ИИ задает человеку. Система, реализующая прямую цепочку рассуждений, на основании имеющихся условий делает возможные логические выводы; система, реализующая обратную цепочку рассуждений по имеющимся выводам, ищет необходимые для них условия.

Хотя реализация обратной цепочки рассуждений не сложнее реализации прямой цепочки рассуждений, в этой главе приводится полный цикл разработки системы ИИ от алгоритма до программы, которую можно применять в реальной экспертной системе.

Разработка базы знаний: дерево решений

Прежде всего, поставим задачу, для решения которой будет разрабатываться экспертная система. Подходящей задачей, при решении которой можно использовать обратную цепочку рассуждений, может быть задача, вытекающая из следующей ситуации: к директору крупной технической фирмы пришел человек, желающий устроиться на работу. Директор располагает сведениями о его квалификации, о потребностях фирмы в специалистах и общем положении дел в фирме. Ему нужно решить, какую должность в фирме может занять посетитель.

На первый взгляд задача не очень сложная, но на решение директора влияет много факторов. Допустим, претендент работает в данной области недавно, но уже сделал важное открытие, или он закончил учебное заведение с посредственными оценками, но несколько лет работал по специальности. В данной ситуации люди ведут себя по-разному, и хотя для того чтобы получить работу, необходимо удовлетворять определенным критериям, в биографии претендента могут быть самые различные факты, анализ которых поможет подобрать для него соответствующую должность.

Поскольку в задаче надо выбрать один из нескольких возможных вариантов (должностей), для ее решения можно воспользоваться обратной цепочкой рассуждений. В действительности ответ уже существует. Перед директором сидит человек и всеми силами старается произвести на него хорошее впечатление. Если директора этот человек устраивает, для него нужно подобрать подходящую должность. Директору необходимо задать посети-

телю такие вопросы, ответы на которые дадут возможность сделать правильный выбор.

Итак, задача поставлена. Теперь нужно наглядно ее представить. Для описания подобных задач обычно используются диаграммы, которые называются деревьями решений. Деревья решений дают необходимую наглядность и позволяют проследить ход рассуждений.

Диаграммы называются деревьями решений, потому что подобно настоящему дереву имеют ветви. Ветви деревьев решений заканчиваются логическими выводами. Для рассматриваемого примера вывод заключается в том, предложит ли директор должность соискателю, и если да, то какую. Многие задачи сложны, и их непросто представить (или для их решения не собираются использовать экспертную систему). Дерево решений помогает преодолеть эти трудности.

На рис. 6.1 показано дерево решений для примера с приемом на работу. Видно, что диаграмма состоит из кружков и прямоугольников, которые называются вершинами. Каждой вершине

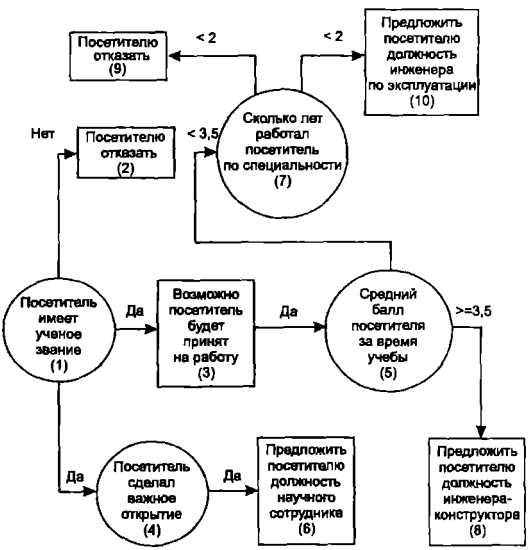


Рис. 6.1. Дерево решений для выбора должности

присваивается номер. На вершины можно ссылаться по этим номерам. Линии, соединяющие вершины, называются дугами или ветвями. Кружки, содержащие вопросы, называются вершинами решений. Прямоугольники содержат цели диаграммы и означают логические выводы. Линии показывают направление диаграммы. Многие вершины имеют сразу по нескольку ветвей, связывающих их с другими вершинами. Выбор выходящей из вершины ветви определяется проверкой условия, содержащегося в вершине.

Например, вершина 5 (см. рис. 6.2) содержит вопрос, на который есть два возможных ответа, и поэтому у нее два пути в зависимости от среднего балла посетителя за время учебы, то есть возможен выбор одной из двух ветвей. Если средний балл равен 3,1, то будет выбран первый путь, так как 3,1 меньше 3,5. В программе под средний балл сначала отводится переменная, а затем ей присваивается значение. Можно сказать, что вершины содержат переменные, а пути — это условия, в соответствии с которыми переменным присваиваются значения. После того как для проблемной области сформулированы правила, эти условия становятся условными частями (ЕСЛИ) правила. Прямоугольники

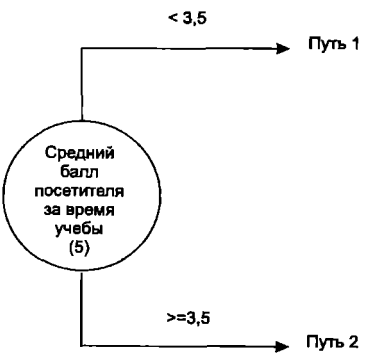


Рис. 6.2. Определение пути к вершине решения

Рис. 6.3. Логический вывод

содержат частные или общие выводы. Например, прямоугольник на рис. 6.3 может содержать ответ на вопрос, будет ли посетителю предложена работа. Общая цель системы, в которой реализованы обратные рассуждения, — получить окончательный ответ. Локальной целью может быть содержащийся в прямоугольнике на рис. 6.4 ответ на вопрос, будет ли посетителю предложена должность. Однако эта вершина имеет и исходящие ветви, и, следовательно, через нее может проходить путь к следующему логическому выводу. В последнем случае, поскольку исходящая ветвь не содержит условия и она только одна, говорят, что вершина содержит локальный вывод для другой цели. Локальный вывод — это также составляющая условной части правила. Ниже будет приведен пример, содержащий локальные выводы.

Преобразование дерева решений в правила

Как мы уже говорили, правило ЕСЛИ–ТО состоит из двух частей. Часть ЕСЛИ может включать несколько условий, которые связываются между собой логическими операторами И, ИЛИ и НЕ. Часть ТО правила включается в работу только в том случае, если истинны все значения в условной части. В дереве решений обеим частям правила соответствуют связанные между собой вершина решения (кружок) и вершина логического вывода (прямоугольник). Условная часть содержит все вершины решения, находящиеся на пути к логическому выводу, то есть каждая вершина решения на пути к выводу — это одно условие части ЕСЛИ. Вывод же составляет часть ТО правила (см. рис. 6.5).

Рис. 6.4. Пример логического вывода

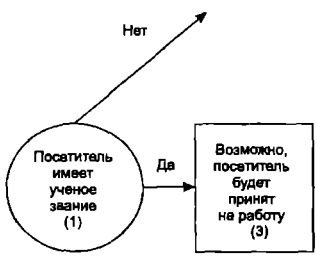


Рис. 6.5. Преобразование части дерева в правило

Для того чтобы найти условия, при которых посетителю может быть предложена работа, нужно просмотреть все пути, ведущие к прямоугольнику 3. К нему ведет только один путь от вершины решения 1, которая содержит вопрос: «Посетитель имеет ученое звание?» Правило, соответствующее этому пути, имеет вид:

ЕСЛИ посетитель имеет ученое звание=да,
ТО посетитель, возможно, будет принят на работу=да

Длинную фразу «посетитель имеет ученое звание» можно заменить переменной, принимающей значения «да» или «нет». В действительности все вершины содержат переменные, имеющие уникальные имена.

Список имен переменных, текст, который они заменяют, и номера вершин приведены в табл. 6.1. Использование переменных вместо полного текста упрощает формирование и запись правил.

Процесс формирования правил для всех возможных логических выводов состоит из следующих шагов:

1. Выбрать из дерева решений вершину вывода (прямоугольник) и зафиксировать ее.

2. Найти вершину решения (кружок), расположенную слева от выбранной вершины вывода и связанную с ней ветвью, и зафиксировать ее.

Таблица 6.1

Таблица имен переменных

Имя переменной	Условия	Вершина(ы)
DEGREE	Посетитель имеет ученое звание?	1
DISCOVERY	Посетитель сделал важное открытие?	4
EXPERIENCE	Каков опыт работы в этой области?	7
GRADE	Каков средний балл посетителя за время учебы?	5
POSITION	Какая должность предложена посетителю?	2, 6, 8, 9, 10
QUALIFY	Возможно, посетитель будет принят на работу	3

3. Повторять шаг 2 до тех пор, пока не будут исчерпаны все вершины решения, расположенные левее зафиксированной вершины

вывода, или не встретится новая вершина вывода. Если встретилась вершина вывода, то ее надо зафиксировать и прекратить выполнение шага 2. Выполнение также прекращается, если исчерпаны все вершины.

4. Каждая вершина решения, составляющая путь, — это одна из переменных части ЕСЛИ правила. Значение, связанное с ветвью, представляет собой условие. Переменные условной части правила объединяются логическим оператором И.

5. Выбранный логический вывод перенести в часть ТО правила.

Создание правил

В качестве примера рассмотрим путь, показанный на рис. 6.6. Выполнив описанные шаги, получим:

Вершина вывода 6 Путь 6, 4, 1.

Поскольку используются обратные рассуждения, то выполнение начинается с вывода и дерево решения просматривается в обратную сторону. Применив полученный путь и имена переменных из табл. 6.1, можно создать правило:

ЕСЛИ DEGREE=ДА И DISCOVERY=ДА,
ТО POSITION=НАУЧНЫЙ СОТРУДНИК

Таким образом, если посетитель имеет ученое звание и сделал важное открытие, то ему будет предложена должность научного сотрудника.

Рис. 6.6. Пример пути

Воспользовавшись уже известными принципами, приступим к разработке базы знаний. Читатель аналогично может построить базу знаний для своей проблемной области; ему только надо учесть, что должно быть хотя бы одно правило для каждого пути, ведущего к цели. В табл. 6.2 собраны все правила для дерева

решений, показанного на рис. 6.1. Номера правил выбраны произвольно и служат только для удобства идентификации.

Правила соответствуют всем шести путям, ведущим к шести возможным целям дерева решений.

Таблица 6.2

Правила ЕСЛИ–ТО

Правило	Путь
10 ЕСЛИ DEGREE=НЕТ, ТО POSITION=НЕТ	1, 2
	1, 3
20 ЕСЛИ DEGREE=ДА, ТО QUALIFY=ДА	1, 4,
	6
30 ЕСЛИ DEGREE=ДА И DISCOVERY=ДА,	3, 5,
ТО POSITION=НАУЧНЫЙ СОТРУДНИК	7
40 ЕСЛИ QUALIFY=ДА И AVERAGE=3.5 И	10
EXPERIENCE=2,	3, 5
ТО POSITION=ИНЖЕНЕР ПО ЭКСПЛУАТАЦИИ	
50 ЕСЛИ QUALIFY=ДА И AVERAGE=3.5 И	7, 9
EXPERIENCE=2,	3, 5
ТО POSITION=НЕТ	
60 ЕСЛИ QUALIFY=ДА И AVERAGE=3.5,	8
ТО POSITION=ИНЖЕНЕР-КОНСТРУКТОР	

Из приведенного примера видно, как удобно применять деревья решений. Они позволяют просто и наглядно формировать правила для базы знаний, а без базы знаний экспертную систему не построить.

Работа с базой знаний

Приступим к реализации обратной цепочки рассуждений. База знаний должна использоваться для определения пути, ведущего к какому-либо логическому выводу. Например, если путь завершается в вершине 9 дерева решений, вывод будет: «Отказать посетителю в работе». Выводы, содержащиеся в вершинах решения, представляют собой переменные частей ТО правил. Путь, который надо проделать для получения вывода, помогает понять, почему сделан именно такой вывод. Иными словами, часть ТО является решением, а условия части ЕСЛИ — причинами, приведшими к решению.

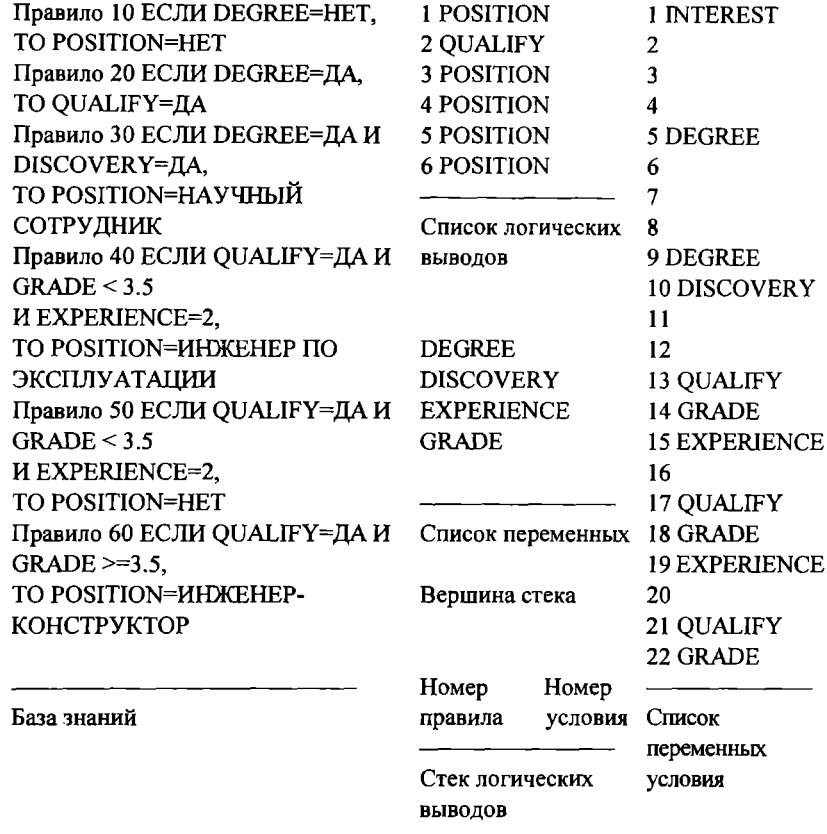


Рис. 6.7. База знаний и структуры данных

При создании экспертной системы для упрощения ответа на вопросы и решения поставленной задачи в систему включается ряд полезных таблиц или структур данных. Структуры данных нужны для работы с базой знаний. На рис. 6.7 показаны база знаний и необходимые структуры данных. После определения метода решения выбранного круга задач можно приступить к разработке системы.

Прежде чем начать писать программы, нужно составить алгоритм и создать некоторые специальные структуры данных.

Список логических выводов

Список логических выводов — это структура данных, содержащая упорядоченный список возможных логических выводов. Список состоит из номера правила, логического вывода, связанного с этим правилом, и условий, которые формируют вывод. На каждое правило базы знаний в списке приходится одна запись. На рис. 6.8 приведен полностью сформированный список логических выводов для всех правил базы знаний. Создание записи списка поясним на примере правила 10. Часть ТО правила 10 содержит переменную POSITION, то есть переменная POSITION связана с логическим выводом правила 10.

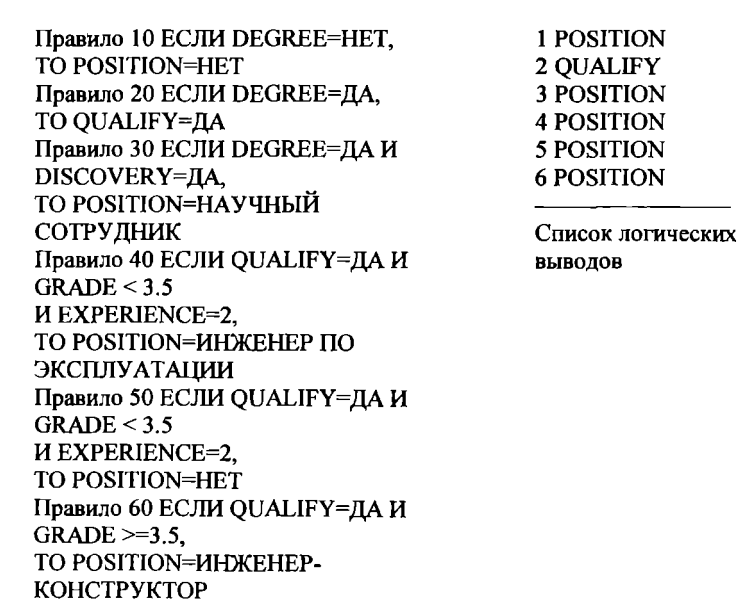


Рис. 6.8. Список логических выводов

Список считается сформированным, когда логический вывод каждого правила помещен в запись с номером, совпадающим с номером правила.

Список логических выводов используется исключительно для поиска вывода по номеру правила. Когда условия части ЕСЛИ истинны, вызывается часть ТО, тем самым переменной логического вывода присваивается значение. Например, если надо узнать, будет ли посетителю предложена работа, в списке ищется переменная POSITION. Она содержится в первой же записи, то есть в правиле 10:

ЕСЛИ DEGREE=НЕТ, ТО POSITION=НЕТ

Посетитель не будет принят на работу, если переменная DEGREE имеет значение НЕТ. Если же переменная DEGREE имеет значение ДА, обращаться к части ТО правила нельзя, поскольку не выполняется условие части ЕСЛИ (DEGREE=ДА). Поэтому надо продолжить поиск правила, содержащего в части ТО переменную POSITION. Однако не будем торопиться, сначала рассмотрим другие структуры данных, а потом опишем их взаимодействие.

Список переменных

Список переменных содержит имена переменных для всех условных частей правил базы знаний и признак их инициализации. Признак инициализации показывает, присвоено ли переменной значение. Список переменных приведен на рис. 6.9. Независимо от того, в скольких условиях встречается переменная, в список переменных она включается всего один раз. В этот список также нельзя включать переменные логического вывода, поскольку их значения определяются с помощью правил. Первоначально предполагается, что переменным значения еще не присвоены и признак инициализации для всех переменных равен NI. По мере того как полученная от посетителя информация передается системе и переменным присваиваются значения, признак инициализации меняется на I.

До того как правило включено в работу, все переменные, входящие в его условную часть, должны быть проинициализированы. Определить, присвоено ли переменной условия значение, можно, просмотрев список переменных. Если переменная отмечена как NI, то прежде чем начать работать с правилом, ей надо

присвоить значение. Как только от посетителя получена информация и переменной присвоено значение, она помечается как I. После этого значение переменной сравнивается с правой частью соответствующего условия, в которое входит эта переменная. Рассмотрим, например, правило 10:

Правило 10 ЕСЛИ DEGREE=НЕТ,
ТО POSITION=НЕТ
Правило 20 ЕСЛИ DEGREE=ДА,
ТО QUALIFY=ДА
Правило 30 ЕСЛИ DEGREE=ДА И
DISCOVERY=ДА,
ТО POSITION=НАУЧНЫЙ
СОТРУДНИК
Правило 40 ЕСЛИ QUALIFY=ДА И
GRADE < 3.5
И EXPERIENCE=2,
ТО POSITION=ИНЖЕНЕР ПО
ЭКСПЛУАТАЦИИ
Правило 50 ЕСЛИ QUALIFY=ДА И
GRADE < 3.5
И EXPERIENCE=2,
ТО POSITION=НЕТ
Правило 60 ЕСЛИ QUALIFY=ДА И
GRADE >=3.5,
ТО POSITION=ИНЖЕНЕР-
КОНСТРУКТОР

База знаний

DEGREE
DISCOVERY
EXPERIENCE
GRADE

Список переменных

	Значение
NI	
NI	
NI	
NI	

Рис. 6.9. Список переменных

10 ЕСЛИ DEGREE=НЕТ, ТО POSITION=НЕТ

Переменная DEGREE находится в первой строке списка, но ей еще не присвоено значение (NI), и, значит, обратиться к условной части правила нельзя. Для того чтобы присвоить значение переменной DEGREE, нужно узнать у посетителя, имеет ли он ученое звание. Ответ посетителя и будет значением переменной. Теперь можно приступить к работе с условной частью правила:

ЕСЛИ DEGREE=НЕТ

Часть ТО можно выполнить, если переменная DEGREE имеет значение НЕТ. В любом случае после присвоения переменной DEGREE значения для нее в соответствующей строке списка переменных NI меняется на I:

Имя переменной Признак Значение DEGREE I НЕТ

С этого момента, в каком бы правиле в условной части не встретилась переменная DEGREE, она будет считаться проинициализированной. Это означает, как только от посетителя получена информация, ее можно использовать для работы с любыми правилами.

Список переменных условия

Условная часть правила (ЕСЛИ) может содержать несколько переменных. В проектируемой системе в правиле может быть до четырех переменных условия. Хотя переменные могут быть связаны любым из логических операторов (И, ИЛИ, НЕ), здесь рассматривается лишь оператор И как наиболее распространенный. Например, правило 30 состоит из двух условий, связанных оператором И:

ЕСЛИ DEGREE=ДА И DISCOVERY=ДА

А правила 40, 50 и так далее содержат по три условия. Необходимо твердо помнить, если условия части ЕСЛИ правила связаны оператором И, то обратиться к части ТО можно только, когда все переменные условия проинициализированы. Например, в правиле 30:

30 ЕСЛИ DEGREE=ДА И DISCOVERY=ДА,
ТО POSITION=НАУЧНЫЙ СОТРУДНИК

прежде чем обратиться к части ТО, нужно присвоить значения переменным DEGREE и DISCOVERY. Поэтому следует сформировать для всех правил список переменных, входящих в их условные части, чтобы можно было проверить, проинициализирована переменная или нет, и только потом приступить к работе с правилом.

Правило 10 ЕСЛИ DEGREE=НЕТ,	1 INTEREST
ТО POSITION=НЕТ	2
Правило 20 ЕСЛИ DEGREE=ДА,	3
ТО QUALIFY=ДА	4

Правило 30 ЕСЛИ DEGREE=ДА И	5 DEGREE
DISCOVERY=ДА,	6
ТО POSITION=НАУЧНЫЙ	7
СОТРУДНИК	8
Правило 40 ЕСЛИ QUALIFY=ДА И	9 DEGREE
GRADE < 3.5	10 DISCOVERY
И EXPERIENCE=2,	11
ТО POSITION=ИНЖЕНЕР ПО	12
ЭКСПЛУАТАЦИИ	13 QUALIFY
Правило 50 ЕСЛИ QUALIFY=ДА И	14 GRADE
GRADE < 3.5	15 EXPERIENCE
И EXPERIENCE=2,	16
ТО POSITION=НЕТ	17 QUALIFY
Правило 60 ЕСЛИ QUALIFY=ДА И	18 GRADE
GRADE >=3.5,	19 EXPERIENCE
ТО POSITION=ИНЖЕНЕР-	20
КОНСТРУКТОР	21 QUALIFY
	22 GRADE
База знаний	Список переменных условия

Рис. 6.10. Список переменных условия

Структура данных, содержащая перечень всех переменных для всех условных частей правил, называется списком переменных условия. На рис. 6.10 показан список переменных условия для шести правил рассматриваемой базы знаний. Для простоты программирования предполагается, что каждое правило не может содержать больше четырех переменных условия. Слева от имен переменных даны числа (1–22), указывающие индекс элемента массива (по четыре на правило), в который помещается имя соответствующей переменной. Незанятые элементы массива, отведенные правилу, остаются пустыми. В принципе можно запрограммировать любое число переменных для каждого правила.

Однако при отведении места под переменные условия лучше для каждого правила резервировать одинаковое число элементов массива. Это упростит вычисление индекса первого элемента, отведенного правилу в списке. Его можно вычислить с помощью простой формулы:

4*(номер правила/10–1)+1

Например, переменные правила 50 будут размещаться начиная с 17-го элемента массива:

$$4*(50/10-1)+1=17$$

Теперь посмотрим, каким образом три описанные структуры данных соотносятся с мыслительной деятельностью человека в процессе обратной цепочки рассуждений. Прежде всего человек просматривает все возможные пути, способные привести к решению задачи (список логических выводов). Затем он выделяет условия, составляющие эти пути (список переменных и список переменных условия). Такие структуры данных позволяют быстро обрабатывать информацию, не повторяя одни и те же шаги по нескольку раз, потому что значения переменных можно использовать в определенной ситуации для различных логических выводов. Если же при разговоре с человеком, устраивающимся на работу, у директора нет не только компьютера, но даже карандаша и бумаги, ему придется много раз переспрашивать, ведь сразу просто невозможно запомнить. Разумеется, в конце концов он примет решение, но затратит много сил и времени.

Стек логических выводов

Обсудим последнюю, четвертую структуру данных — стек логических выводов. Это главная структура, так как в экспертной системе, реализующей обратную цепочку рассуждений, она связывает остальные структуры данных. В стеке логических выводов хранится информация о нужных пользователю логических выводах и о переменных условия, инициализацию которых необходимо проверить. Для того чтобы понять, что такое стек логических выводов, вернемся к правилу 40:

40 ЕСЛИ QUALIFY=ДА (условие 1),
AVERAGE < 3.5 (условие 2) И EXPERIENCE=2 (условие 3),
ТО POSITION=ИНЖЕНЕР ПО ЭКСПЛУАТАЦИИ

Правило состоит из условной части с тремя условиями (1, 2, 3) и части ТО. Для того чтобы выяснить, можно ли вызвать часть ТО, необходимо проверить все условия. Если все условия истинны, в работу включается часть ТО, согласно которой посетителя предлагается назначить инженером по эксплуатации оборудова-

ния. Правило надо отбросить, если хоть одно из условий не выполняется. Каким же образом сохранить трассу проверяемых правил и содержащихся в них условий? На рис. 6.11 показано, как это сделать, используя стек логических выводов.

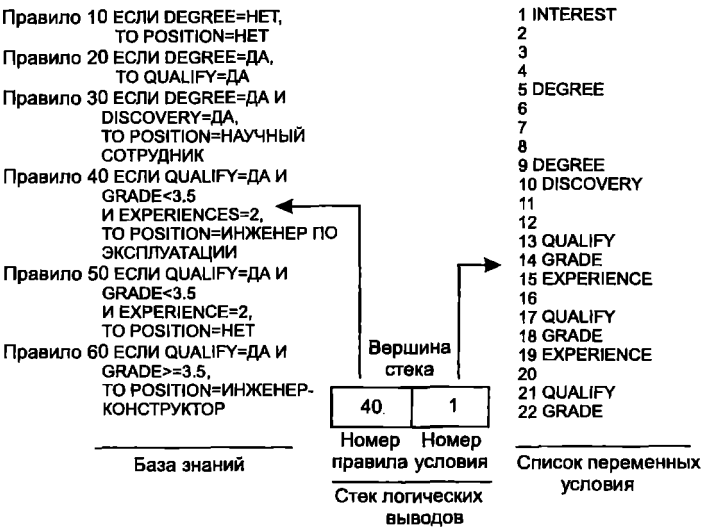


Рис. 6.11. Стек логических выводов

Видно, что в настоящий момент проверяется первое условие правила 40:

QUALIFY=ДА

Но значение переменной QUALIFY присваивается в правиле 20:

ЕСЛИ DEGREE=ДА (условие 1), ТО QUALIFY=ДА

Поэтому для определения значения переменной QUALIFY надо обратиться к правилу 20. Правило 20 помещается в стек (см. рис. 6.12). Номер условия указывает, что нужно посмотреть первое условие правила 20:

DEGREE=ДА

Значение переменной DEGREE (ДА или НЕТ) присваивается согласно данным, полученным от посетителя. Переменной QUALIFY можно присвоить значение ДА только, если DEGREE=ДА (таково

условие в части ЕСЛИ). Предположим, у посетителя есть ученое звание (DEGREE=ДА), тогда QUALIFY=ДА.

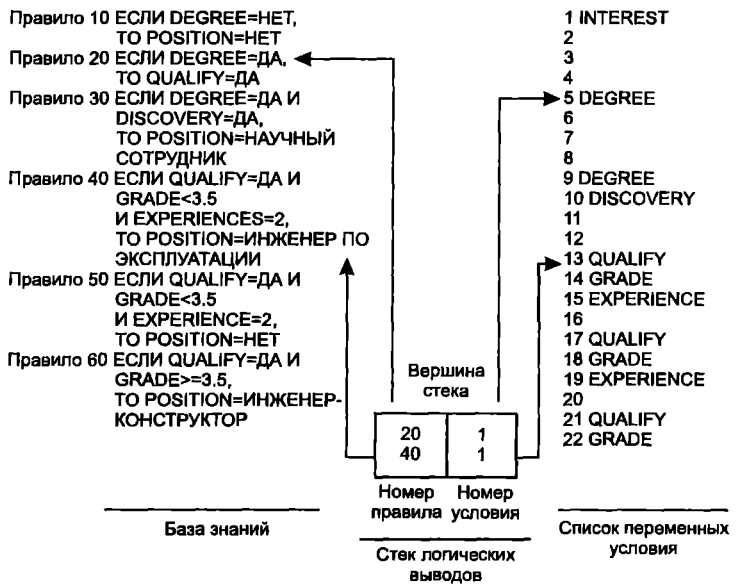


Рис. 6.12. Стек логических выводов при работе с дополнительным правилом

После этого правило 20 можно удалить из стека. Поскольку QUALIFY=ДА, можно перейти от первого ко второму условию правила 40 (рис. 6.13). Такое взаимодействие стека выводов с базой знаний продолжается на протяжении всей цепочки обратных рассуждений.

В заключение повторим, что стек логических выводов хранит информацию о том, какое условие какого правила проверяется в определенный момент времени.

Пример использования базы знаний

Для того чтобы свести воедино все вышесказанное, рассмотрим полный цикл обработки экспертной системой запроса пользо-

вателя. Работа начинается с ввода пользователем логического вывода в форме вопроса:

Будет ли посетитель принят на работу?

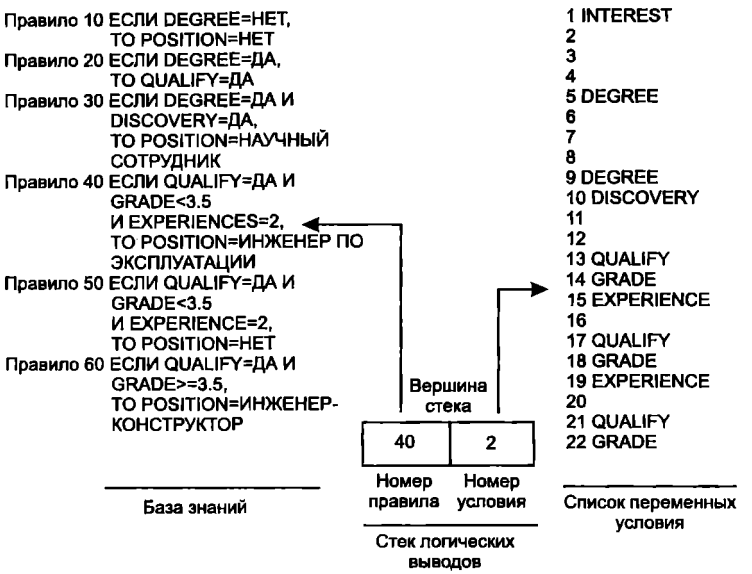


Рис. 6.13. Обновленный стек логических выводов

Система выбирает из таблицы переменных переменную логического вывода POSITION. Затем в стек помещается номер правила, в котором есть такая переменная логического вывода. Список логических выводов просматривается с самого начала (с правила 10), так как переменная POSITION встретилась в первый раз. Система сразу же находит переменную POSITION в правиле 10 и помещает в стек один элемент (см. рис. 6.14). В правиле 10 система находит переменную условия DEGREE и, просмотрев список переменных, обнаруживает, что она еще не проинициализирована. Переменной DEGREE нет и в списке переменных логического вывода, поскольку она не входит в часть ТО какого-либо правила. Система запросит информацию:

Посетитель имеет ученое звание?

Как только полученный ответ будет передан системе, переменной DEGREE присваивается значение, а в списке переменных для нее проставляется признак инициализации (рис. 6.15).

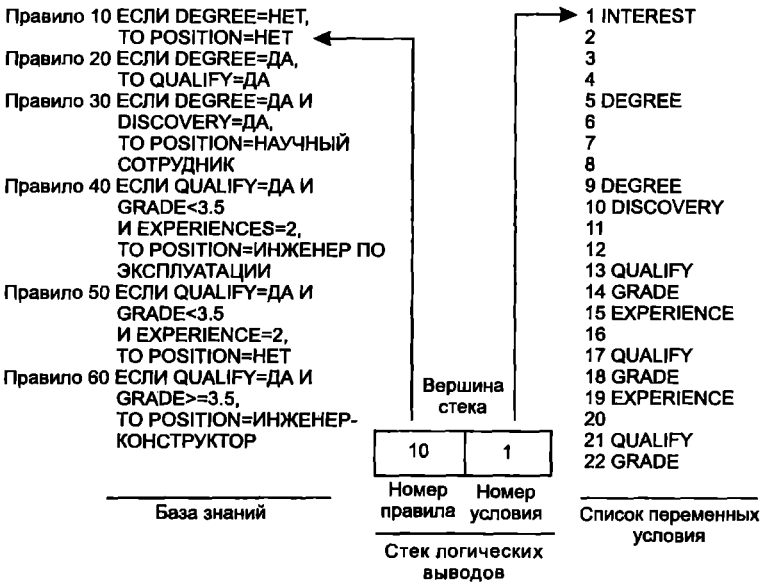


Рис. 6.14. Начальное состояние стека логических выводов

Пусть посетитель ответил на вопрос «ДА», тогда правило

10 ЕСЛИ DEGREE=НЕТ, ТО POSITION=НЕТ

из-за противоречия в условии исключается, а соответствующий ему логический вывод удаляется из стека. В поисках нового правила с переменной логического вывода POSITION просмотр списка логических выводов продолжается. Следующим система выбирает правило 30

30 ЕСЛИ DEGREE=ДА И DISCOVERY=ДА,
ТО POSITION=НАУЧНЫЙ СОТРУДНИК

и помещает его в стек (см. рис. 6.16). Теперь система попытается присвоить значения всем переменным условия, входящим в правило 30; список переменных условия приведен на рис. 6.17.

Правило 10 ЕСЛИ DEGREE=НЕТ,
ТО POSITION=НЕТ
Правило 20 ЕСЛИ DEGREE=ДА,
ТО QUALIFY=ДА
Правило 30 ЕСЛИ DEGREE=ДА И
DISCOVERY=ДА,
ТО POSITION=НАУЧНЫЙ
СОТРУДНИК
Правило 40 ЕСЛИ QUALIFY=ДА И
GRADE < 3.5
И EXPERIENCE=2,
ТО POSITION=ИНЖЕНЕР ПО
ЭКСПЛУАТАЦИИ
Правило 50 ЕСЛИ QUALIFY=ДА И
GRADE < 3.5
И EXPERIENCE=2,
ТО POSITION=НЕТ
Правило 60 ЕСЛИ QUALIFY=ДА И
GRADE >=3.5,
ТО POSITION=ИНЖЕНЕР-
КОНСТРУКТОР

База знаний

	Значение
DEGREE	ДА
DISCOVERY	NI
EXPERIENCE	NI
GRADE	NI

Список переменных

Рис. 6.15. Скорректированный список переменных

Номер условия, хранящийся в стеке для правила 30, указывает на переменную DEGREE, которой уже присвоено значение (в списке переменных она отмечена как I), поэтому для первого элемента стека номер условия будет увеличен на единицу. Второй в списке переменных условия для правила 30 стоит переменная DISCOVERY (рис. 6.18).

В списке переменных условия отмечено, что переменной DISCOVERY значение еще не присвоено, а так как DISCOVERY не входит в переменные логического вывода, система опять запросит информацию:

Посетитель сделал важное открытие?

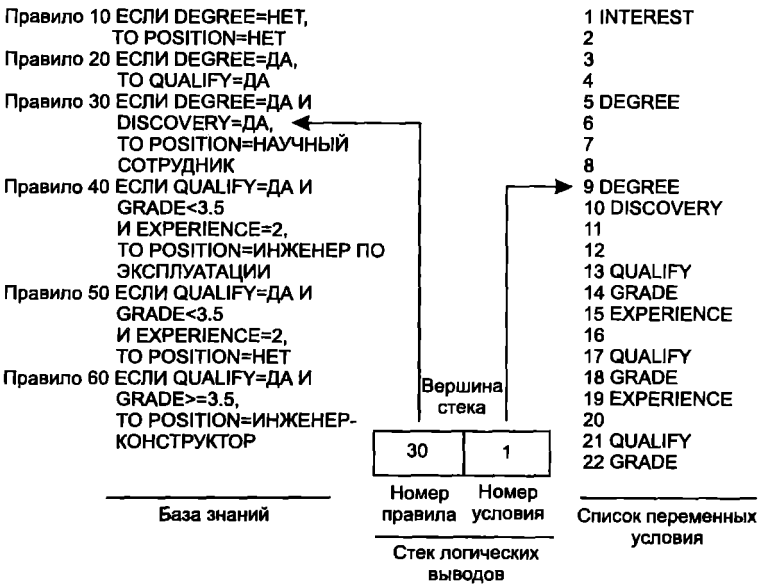


Рис. 6.16. Управление стеком логических выводов

Предположим, система получит ответ «Нет», тогда список переменных будет скорректирован (см. рис. 6.19). Поскольку первые две переменные уже проинициализированы, номер условия в стеке будет увеличен до 3. В правиле 30 третьей переменной условия нет, значит, можно приступить к его анализу:

30 ЕСЛИ DEGREE=ДА И DISCOVERY=ДА,
ТО POSITION=НАУЧНЫЙ СОТРУДНИК

Но второе условие правила не выполняется, и, следовательно, логический вывод опять удаляется из стека.

Правило 10 ЕСЛИ DEGREE=НЕТ,	1 INTEREST
ТО POSITION=НЕТ	2
Правило 20 ЕСЛИ DEGREE=ДА,	3
ТО QUALIFY=ДА	4
Правило 30 ЕСЛИ DEGREE=ДА И	5 DEGREE
DISCOVERY=ДА,	6
ТО POSITION=НАУЧНЫЙ	7
СОТРУДНИК	8

Правило 40 ЕСЛИ QUALIFY=ДА И
GRADE < 3.5
И EXPERIENCE=2,
ТО POSITION=ИНЖЕНЕР ПО
ЭКСПЛУАТАЦИИ
Правило 50 ЕСЛИ QUALIFY=ДА И
GRADE < 3.5
И EXPERIENCE=2,
ТО POSITION=НЕТ
Правило 60 ЕСЛИ QUALIFY=ДА И
GRADE >=3.5,
ТО POSITION=ИНЖЕНЕР-
КОНСТРУКТОР

База знаний

9 DEGREE
10 DISCOVERY
11
12
13 QUALIFY
14 GRADE
15 EXPERIENCE
16
17 QUALIFY
18 GRADE
19 EXPERIENCE
20
21 QUALIFY
22 GRADE

Список переменных условия

Рис. 6.17. Список переменных условия

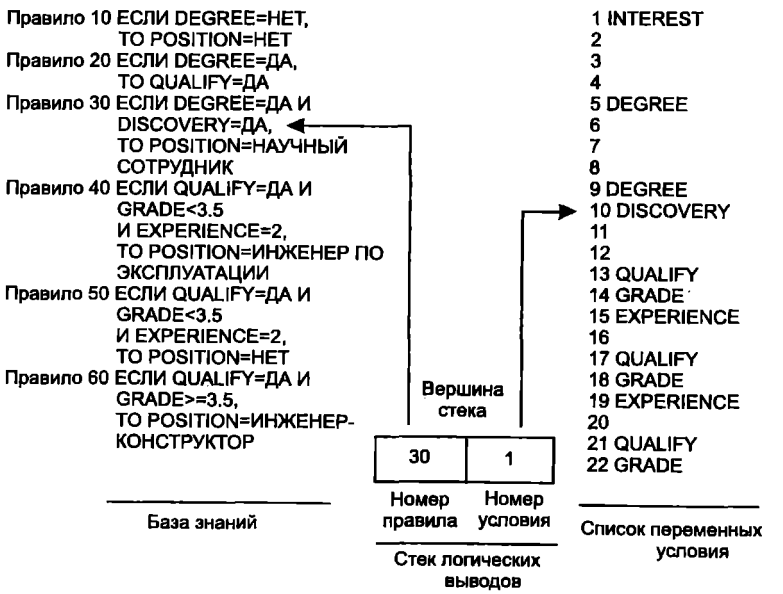


Рис. 6.18. Очередное изменение стека логических выводов

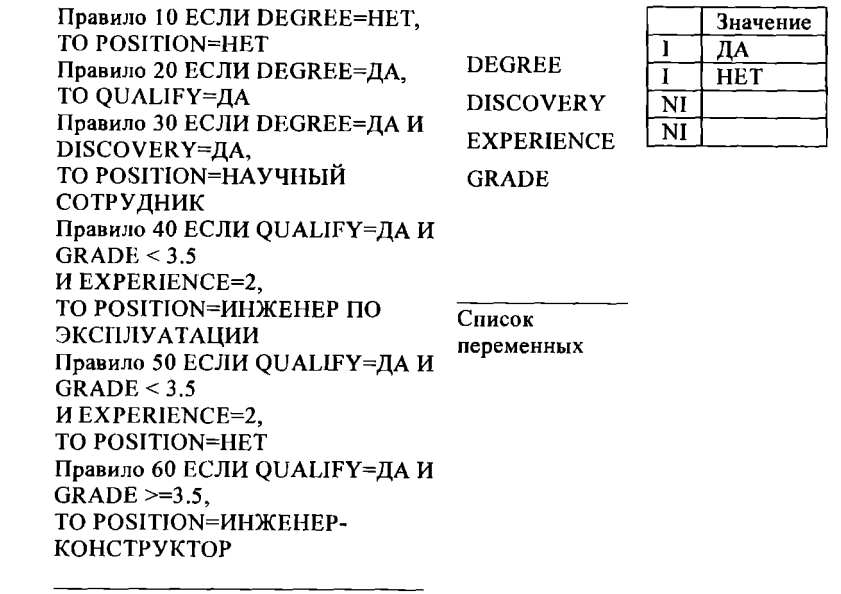


Рис. 6.19. Список переменных

Поиск правила с переменной логического вывода POSITION будет продолжен. Следующим система выбирает правило 40:

40 ЕСЛИ QUALIFY=ДА И GRADE < 3,5 И EXPERIENCE=2,
ТО POSITION=ИНЖЕНЕР ПО ЭКСПЛУАТАЦИИ

и помещает его в стек (см. рис. 6.20). Просмотрев условия правила и список переменных, система обнаруживает, что переменной QUALIFY значение еще не присвоено и что, кроме того, эта переменная есть в списке логического вывода правила 20. Система заносит в стек новый элемент (см. рис. 6.21). Правило 20 содержит только одну переменную условия — DEGREE и его можно выполнить, так как переменная DEGREE уже проинициализирована (DEGREE=ДА):

20 ЕСЛИ DEGREE=ДА, ТО QUALIFY=ДА

Согласно правилу 20 переменной QUALIFY присваивается значение ДА (QUALIFY=ДА), и отработанный логический вывод

удаляется из стека. Система вернется к правилу 40, увеличит на единицу номер условия (см. рис. 6.22) и приступит к работе с переменной GRADE. Переменной GRADE нет в списке выводов, и значение ей еще не присвоено. Система задаст новый вопрос:

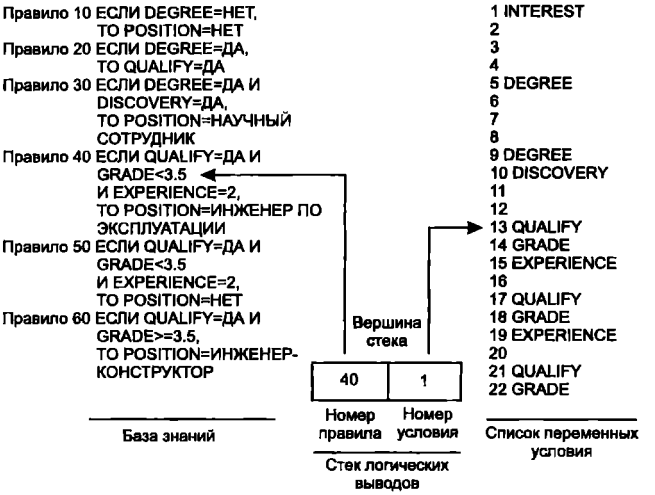


Рис. 6.20. Стек выводов после очередного изменения

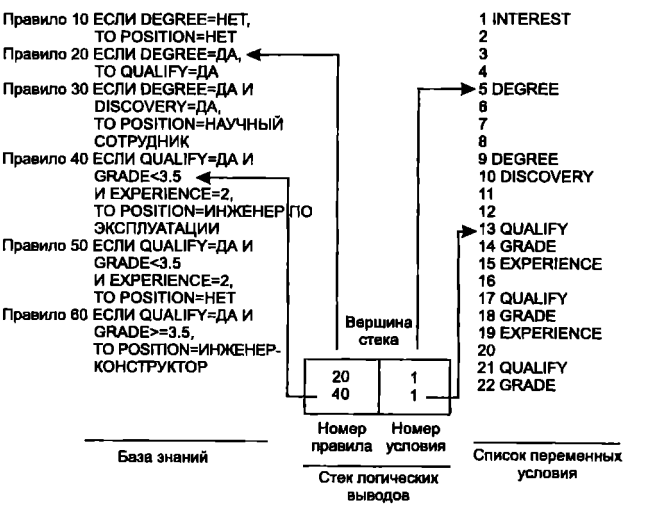


Рис. 6.21. Стек логических выводов с несколькими элементами

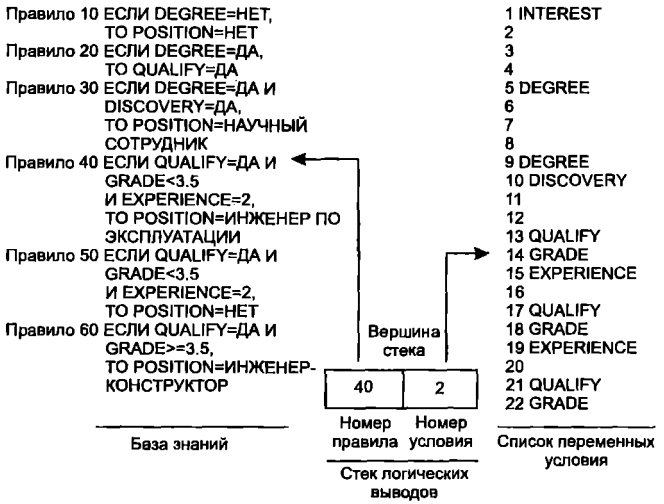


Рис. 6.22. Стек логических выводов после очередного изменения

Каков был средний балл посетителя в учебном заведении?

Ответ посетителя, скажем 3.0, и задает значение переменной GRADE. На рис. 6.23 показаны список переменных и стек логических выводов после того, как номер условия продвинут до трех.

Переменной EXPERIENCE нет в списке логических выводов, и она не проинициализирована, поэтому система опять спросит:

Сколько лет посетитель работал по специальности?

Пусть системе передан ответ посетителя: 4. Тогда переменной EXPERIENCE присваивается значение 4, признак ее инициализации изменяется на 1, а номер условия в стеке продвигается до 4. В правиле 40 переменных условия больше нет, и, значит, можно приступить к проверке условий. Все условия правила

40 ЕСЛИ QUALIFY=ДА И GRADE=3.0 И EXPERIENCE=2

истинны, так как

QUALIFY=ДА
GRADE=3.0
EXPERIENCE=2

и в соответствии с частью ТО

ТО POSITION=ИНЖЕНЕР ПО ЭКСПЛУАТАЦИИ

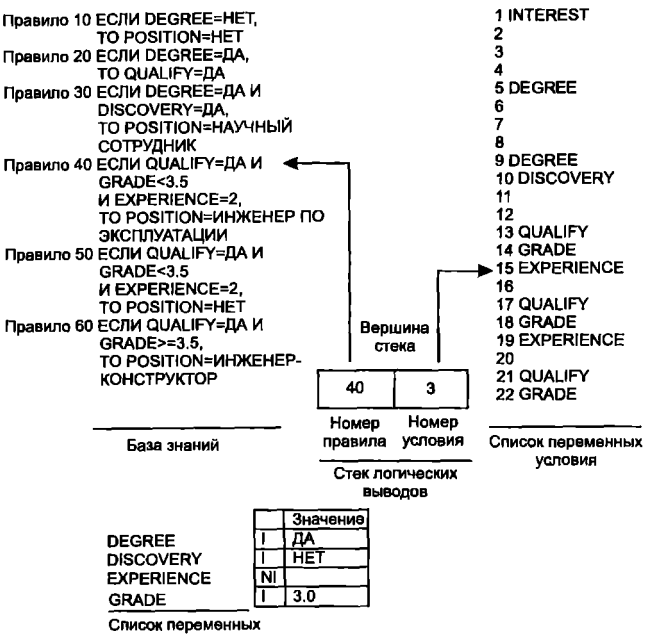


Рис. 6.23. Состояние задачи

переменной POSITION присваивается значение ИНЖЕНЕР ПО ЭКСПЛУАТАЦИИ. Цель достигнута, посетителю будет предложена должность в отделе эксплуатации оборудования.

Замечания по разработке системы

Система, реализующая обратную цепочку рассуждений, должна выполнять следующие шаги:

1. Определить переменную логического вывода.

2. В списке логических выводов искать первое вхождение этой переменной. Если переменная найдена, в стек логических выводов поместить номер соответствующего правила и установить номер условия равным 1. Если переменная не найдена, сообщить пользователю, что ответ найти невозможно.

3. Присвоить значения всем переменным условия из данного правила.

4. Если в списке переменных указано, что какой-либо переменной условия не присвоено значение и ее нет среди переменных логического вывода (ее нет в списке логических выводов), запросить ее значение у пользователя.

5. Если какая-либо переменная условия входит в переменные логического вывода, поместить в стек номер правила, в логический вывод которого она входит, и вернуться к шагу 3.

6. Если из правила нельзя определить значение переменной, удалить соответствующий ему элемент из стека и в списке логических выводов продолжить поиск правила с этой переменной логического вывода.

7. Если такое правило найдено, перейти к шагу 3.

8. Если переменная не найдена ни в одном из оставшихся правил в логическом выводе, правило для предыдущего вывода неверно. Если предыдущего вывода не существует, сообщить пользователю, что ответ получить невозможно. Если предыдущий вывод существует, вернуться к шагу 6.

9. Определить значение переменной из правила, расположенного в начале стека; правило из стека удалить. Если есть еще переменные логического вывода, увеличить значение номера условия и для проверки оставшихся переменных вернуться к шагу 3. Если больше нет переменных логического вывода, сообщить пользователю окончательный вывод.

Экспертная система

Приведенный выше алгоритм можно использовать для разработки программы на БЕЙСИКе, реализующей обратную цепочку рассуждений, которую можно включить в экспертную систему. База знаний должна состоять из правил. Для того чтобы разобраться в описанных выше правилах, нужно подробно изучить программу, приведенную на листинге 6.1.

Пример программы

Давайте рассмотрим, как можно настроить приведенную в настоящей главе программу, реализующую обратную цепочку рассуждений, на любую базу знаний. Программа разработана так, что пользователь может вставить в нее свои правила начиная

с предложения 1500 по 1680. Запустить программу можно с помощью команды RUN.

Пояснения к программе

В программе используется база знаний из примера с приемом на работу. Например, предложения программы 1500 и 1510 описывают правило

10 ЕСЛИ DEGREE=НЕТ, ТО POSITION=НЕТ,

а предложения 1560–1570 описывают правило:

ЕСЛИ QUALIFY=ДА И GRADE 3.5 И EXPERIENCE=2,
ТО POSITION=ИНЖЕНЕР ПО ЭКСПЛУАТАЦИИ

Все переменные, входящие в список переменных (V$), — переменные условия, а не логического вывода. Список переменных формируется в предложении 367. Система должна присвоить значения всем переменным из списка (предложения программы 1700–1715). Список переменных определен в предложении 367. В предложении 305 формируется список логических выводов, а в предложениях 407–409 — список переменных условия.

Листинг программы

Листинг программы сопровождается примерами ее выполнения. Обратите внимание на то, что ответы системы приводятся в сокращенном виде. Например, РО=НАУЧНЫЙ СОТРУДНИК (или ИНЖЕНЕР-КОНСТРУКТОР, или ИНЖЕНЕР ПО ЭКСПЛУАТАЦИИ) означает «Предложить посетителю должность в соответствующем отделе», а РО=НЕТ — «Посетителю отказать»; QUALIFY=ДА означает «Посетитель, возможно, будет принят на работу, но пока неизвестно на какую должность».

```
1 REM ***СИСТЕМА ОБРАТНЫХ РАССУЖДЕНИЙ***
10 REM НОВЫЕ УСЛОВИЯ ПРАВИЛ ЗАПИСЫВАЮТСЯ В ПРЕДЛОЖЕНИЯХ
11 REM 1500, 1520, ...
20 REM НАПРИМЕР, 1500 IF (A1=2) AND (A2=6) OR (A3$=«ДА») THEN S=1
```

```
22 REM 1502 RETURN
30 REM 1520 IF (J$=«НЕТ») THEN S=1
32 REM 1522 RETURN
40 REM ЧАСТЬ ТО ПРИВЕДЕННЫХ ВЫШЕ ПРАВИЛ ВСЕГДА
СОДЕРЖИТ
41 REM THEN S=1
50 REM ПРЕДЛОЖЕНИЯ 1510, 1530, ..., 1550 ПРОГРАММЫ
51 REM ЧАСТИ ТО ПРАВИЛ ПОЛЬЗОВАТЕЛЯ,
60 REM НАПРИМЕР, 1510 PO=«ДА»: RETURN
200 REM ***БЛОК ИНИЦИАЛИЗАЦИИ ПЕРЕМЕННЫХ***
210 REM СПИСОК ВЫВОДОВ (CL$), СПИСОК ПЕРЕМЕННЫХ
(V$), ПРИЗНАКИ
220 REM ИНИЦИАЛИЗАЦИИ (IN), СТЕК ПРАВИЛ (SS$)
221 REM И СТЕК УСЛОВИЙ (SS$),
230 REM СПИСОК ПЕРЕМЕННЫХ УСЛОВИЯ (CV$) И УКАЗА-
ТЕЛЬ СТЕКА (SP)
240 DIM GL$ (10), V$ (10), CV$ (40), IN (10), SS (10), CS (10)
250 SP=11: REM РАЗМЕР СТЕКА —10 ЭЛЕМЕНТОВ, SP=10+1
260 FOR I=1 TO 10
270 GL$ (I)=« »: V$ (I)=« »: IN (I)=0: SS (I)=0: CS (I)=0
280 NEXT I
290 REM ВВОД ПЕРЕМЕННЫХ ЛОГИЧЕСКОГО ВЫВОДА
295 REM ПЕРЕМЕННЫЕ ВВОДЯТСЯ В ПОРЯДКЕ ПОЯВЛЕНИЯ В
ПРАВИЛАХ
300 REM НАЖАТИЕ КЛАВИШИ ВОЗВРАТ ЗАВЕРШАЕТ ВВОД
305 GL$ (1)=«PO»: GL$ (2)=«QU»: GL$ (3)=«PO»: GL$ (4)=«PO»: GL$
(5)=«PO»
306 GL$ (6)=«PO»
307 PRINT«***СПИСОК ЛОГИЧЕСКИХ ВЫВОДОВ***»
310 FOR I=1 TO 10
320 PRINT «ЛОГИЧЕСКИЙ ВЫВОД»; 1;
325 PRINT GL$ (I)
335 NEXT I
338 PRINT «ДЛЯ ПРОДОЛЖЕНИЯ НАЖМИТЕ КЛАВИШУ
ВОЗВРАТ»; Z
340 REM ФОРМИРОВАНИЕ СПИСКА ПЕРЕМЕННЫХ
```

```
350 REM ИМЕНА ПЕРЕМЕННЫХ ВВОДЯТСЯ В ПОРЯДКЕ ИХ
ПОЯВЛЕНИЯ В
351 REM ПРАВИЛАХ, НЕ БОЛЕЕ 3 ДЛЯ ПРАВИЛА
360 REM ПЕРЕМЕННЫЕ ИМЕЮТ УНИКАЛЬНЫЕ ИМЕНА
365 REM НАЖАТИЕ КЛАВИШИ ВК ЗАВЕРШАЕТ ВВОД ПЕРЕ-
МЕННЫХ
366 INPUT «***СПИСОК ПЕРЕМЕННЫХ***»'
367 V$ (1)=«DE»: V$ (2)=«DI»: V$ (3)=«EX»: V$ (4)=«GR»
370 FOR I=1 TO 10
380 PRINT «ПЕРЕМЕННАЯ»; 1;
385 PRINT V$ (I)
395 NEXT I
396 INPUT «ДЛЯ ПРОДОЛЖЕНИЯ НАЖМИТЕ КЛАВИШУ
ВОЗВРАТ»; Z
400 REM ФОРМИРОВАНИЕ СПИСКА ПЕРЕМЕННЫХ УСЛОВИЯ
(ИМЕНА
401 REM ПЕРЕМЕННЫХ ЗАПИСЫВАЮТСЯ В ПОРЯДКЕ ИХ
ПОЯВЛЕНИЯ
402 REM В ЧАСТЯХ ЕСЛИ ПРАВИЛ, НЕ БОЛЕЕ 3 ДЛЯ ПРАВИЛА)
405 REM НАЖАТИЕ КЛАВИШИ ВК ЗАВЕРШАЕТ ВВОД
406 PRINT «***СПИСОК ПЕРЕМЕННЫХ УСЛОВИЯ***»
407 CV$ (1)=«DE»: CV$ (5)=«DE»: CV$ (9)=«DE»: CV$ (10)=«D1»:
CV$ (13)=«QU»
408 CV$ (14)=«GR»: CV$ (15)=«EX»: CV$ (17)=«QU»: CV$
(18)=«GR»:
409 CV$ (19)=«EX»: CV$ (21)=«QU»: CV$ (22)=«GR»
410 FOR I=1 TO 8: PRINT «***УСЛОВИЕ***»; I
420 FOR J=1 TO 4: PRINT «ПЕРЕМЕННАЯ»; J
423 K=4*(I-1)+J
424 PRINT CV$ (K)
430 NEXT J
432 IF I=4 THEN INPUT «ДЛЯ ПРОДОЛЖЕНИЯ НАЖМИТЕ ВК»
435 NEXT I
500 REM ***БЛОК ВЫВОДА***
505 INPUT «***ВВЕДИТЕ ИМЯ ПЕРЕМЕННОЙ ЛОГИЧЕСКОГО
ВЫВОДА***»; V$
```

```
510 REM ИЗ СПИСКА ЛОГИЧЕСКИХ ВЫВОДОВ (GL$) ПОЛУ-
ЧИТЬ НОМЕР
511 REM ПРАВИЛА, СОДЕРЖАЩЕГО ЭТОТ ВЫВОД (SN)
520 F=1: GOSUB 2200: REM НАЧАТЬ ПОИСК С ПЕРВОГО ПРАВИЛА
525 IF SN=0 THEN STOP: REM НЕТ ТАКОГО ЛОГИЧЕСКОГО
ВЫВОДА
530 REM В СТЕК ПОМЕЩАЕТСЯ НОМЕР ПРАВИЛА (SN) И
НОМЕР
535 REM УСЛОВИЯ=1, (СТЕК СОДЕРЖИТ НОМЕРА ПРАВИЛ (SS))
537 REM И НОМЕРА УСЛОВИЙ (CS)
540 GOSUB 2000
545 REM НАЙТИ УСЛОВИЕ В СПИСКЕ ПЕРЕМЕННЫХ УСЛОВИЯ
550 I=(SS (SP)–1)*4+CS (SP)
555 V$=CV$ (I): REM ПЕРЕМЕННАЯ УСЛОВИЯ
560 REM ПОСЛЕДНЕЕ УСЛОВИЕ ДАННОГО ПРАВИЛА?
565 IF V$=« » THEN GOTO 580: REM ПОСЛЕДНЕЕ УСЛОВИЕ:
566 REM — ВЫПОЛНИТЬ ПРАВИЛО
567 REM ПЕРЕМЕННАЯ УСЛОВИЯ ЯВЛЯЕТСЯ ВЫВОДОМ?
568 F=1: GOSUB 2200: IF SN 0 THEN GOTO 520
569 REM ПОМЕСТИТЬ ВЫВОД В СТЕК
570 GOSUB 2400:
571 REM ПРОИНИЦИАЛИЗИРОВАНА ЛИ ПЕРЕМЕННАЯ УСЛО-
ВИЯ?
575 CS (SP)=CS (SP+1): GOTO 545: REM ПРОВЕРИТЬ СЛЕДУЮ-
ЩЕЕ УСЛОВИЕ
580 REM В ЧАСТИ ЕСЛИ ПРАВИЛА БОЛЬШЕ НЕТ УСЛОВИЙ
582 SN=SS (SP)
585 S=0: ON SN GOSUB 1500, 1520, 1540, 1560, 1580, 1600
590 REM МОЖНО ЛИ ВЫЗВАТЬ ЧАСТЬ ТО, ТО ЕСТЬ S=1?
600 IF S=1 THEN GOTO 630: REM ВЫЗВАТЬ ЧАСТЬ ТО
605 REM НЕУДАЧА ... ПРОДОЛЖИТЬ ПОИСК В ОСТАВШИХСЯ
ПРАВИЛАХ
610 I=SS (SP): V$=GL$ (I): REM ПОЛУЧИТЬ ЛОГИЧЕСКИЙ ВЫВОД
615 F=SS (SP)+1: REM ПОИСК ВЫВОДА В ОСТАВШИХСЯ ПРА-
ВИЛАХ
620 GOSUB 2200
```

```
625 SP=SP+1: GOTO 525
626 REM ВЫБРАТЬ ИЗ СТЕКА СТАРЫЙ ВЫВОД И ПОМЕСТИТЬ
НОВЫЙ
630 REM ЧАСТЬ ЕСЛИ ИСТИННА, ВЫЗВАТЬ ЧАСТЬ ТО
635 ON SN GOSUB 1510, 1530, 1550, 1570, 1590, 1610
650 REM ВЫБРАТЬ ЭЛЕМЕНТ СТЕКА; ЕСЛИ БОЛЬШЕ НЕТ —
ОТВЕТ ПОЛУЧЕН
655 REM ЕСЛИ СТЕК НЕ ПУСТ — ПРОВЕРИТЬ ПРЕДЫДУЩИЙ
ВЫВОД
660 SP=SP+1: REM ВЫБРАТЬ ИЗ СТЕКА
670 IF SP 11 THEN GOTO 690: REM СТЕК НЕ ПУСТ
680 PRINT «***НОРМАЛЬНОЕ ЗАВЕРШЕНИЕ***»: REM КОНЕЦ
685 STOP
690 REM СТЕК ПУСТ — ВЫБРАТЬ СЛЕДУЮЩЕЕ УСЛОВИЕ И
ПРОДОЛЖИТЬ
695 CS (SP)=CS (SP)+1
700 GOTO 545
1490 REM ***ПРАВИЛА ЕСЛИ–ТО***
1492 REM ПРАВИЛА ВЗЯТЫ ИЗ БАЗЫ ЗНАНИЙ О ДОЛЖНОСТЯХ
1494 REM ИМЯ ПЕРЕМЕННОЙ СОСТОИТ ИЗ 2 БУКВ ДЛЯ СОВ-
МЕСТИМОСТИ
1496 REM С РАЗНЫМИ ВЕРСИЯМИ БЕЙСИКА (DEG СОКРАЩЕ-
НО ДО DE,
1498 REM A QUAL — ДО QU)
1499 REM ***ПРАВИЛО 1
1500 IF DE$=«НЕТ» THEN S=1
1502 RETURN
1510 PO$=«НЕТ»: PRINT «PO=НЕТ»: RETURN
1519 REM ***ПРАВИЛО 2
1520 IF DE$=«ДА» THEN S=1
1522 RETURN
1530 QU$=«ДА»: PRINT «QU=ДА»: RETURN
1539 REM ***ПРАВИЛО 3
1540 IF (DE$=«ДА») AND (DI$=«ДА») THEN S=1
1542 RETURN
1550 PO$=«ДА»: PRINT «PO=НАУЧНЫЙ СОТРУДНИК»: RENURN
```

```
1559 REM***ПРАВИЛО 4
1560 IF (QU$=«ДА») AND (GR 3.5) AND (EX=2) THEN S=1
1562 RETURN
1570 PO$=«ДА»: PRINT «PO=ИНЖЕНЕР ПО ЭКСПЛУАТАЦИИ»:
RETURN
1579 REM ***ПРАВИЛО 5
1580 IF (QU$=«ДА») AND (GR 3.5) AND (EX 2) THEN S=1
1582 RETURN
1590 PO$=«НЕТ»: PRINT «PO=НЕТ»: RETURN
1599 REM ***ПРАВИЛО 6
1600 IP (QU$=«ДА») AND (GR=3.5) THEN S=1
1602 RETURN
1610 PO$=«ДА»: PRINT «PO=ИНЖЕНЕР-КОНСТРУКТОР»: RETURN
1611 REM ***КОНЕЦ БАЗЫ ЗНАНИЙ***
1690 REM ***ВВОД ДАННЫХ***
1699 REM ВВОД ВЫПОЛНЯЕТСЯ ДЛЯ БАЗЫ ЗНАНИЙ «ДОЛЖ-
НОСТЬ»
1700 INPUT «ПОСЕТИТЕЛЬ ИМЕЕТ УЧЕНОЕ ЗВАНИЕ?»; DE$:
RETURN
1705 INPUT «ПОСЕТИТЕЛЬ СДЕЛАЛ ОТКРЫТИЕ?»; DI$: RETURN
1710 INPUT «СКОЛЬКО ЛЕТ ПОСЕТИТЕЛЬ РАБОТАЛ ПО СПЕЦИ-
АЛЬНОСТИ?»; EX: RETURN
1715 INPUT «КАКОВ СРЕДНИЙ БАЛЛ ПОСЕТИТЕЛЯ В УЧЕБ-
НОМ ЗАВЕДЕНИИ»; CP: RETURN
1750 REM***КОНЕЦ ВВОДА ДАННЫХ***
1999 REM***БЛОК ПОДПРОГРАММ***
2000 REM ПОДПРОГРАММА ЗАПИСИ НОМЕРА ПРАВИЛА (SN)
НОМЕРА
2002 REM УСЛОВИЯ=1 В СТЕК ВЫВОДОВ, СОСТОЯЩЕГО ИЗ
НОМЕРОВ
2003 REM ПРАВИЛ (SS) ИЗ НОМЕРОВ УСЛОВИЙ (CS)
2004 REM ДЛЯ ЭТОГО УКАЗАТЕЛЬ СТЕКА (SP) УМЕНЬШАЕТСЯ
НА ЕДИНИЦУ
2020 SP=SP–1: SS (SP)=SN: CS (SP)=1
2030 RETURN
2099 REM ***
```

```
2200 REM ПОДПРОГРАММА ИЩЕТ В СПИСКЕ ЛОГИЧЕСКИХ
ВЫВОДОВ
2201 REM ПЕРЕМЕННУЮ V$ (I), ЕСЛИ ПЕРЕМЕННАЯ ЕСТЬ В
СПИСКЕ,
2202 REM ТО ВОЗВРАЩАЕТСЯ SNO, НЕТ SN=0
2220 SN=0: REM SN СНАЧАЛА ОБНУЛЯЕТСЯ
2230 FOR I=F TO 8: REM В СПИСКЕ ВЫВОДОВ ИЩЕТСЯ F
2240 IF V$=GL$ (I) THEN GOTO 2270: REM ПРОВЕРКА РАБОЧЕЙ
ТАБЛИЦЫ
2250 NEXT I
2260 RETURN: REM НЕТ В СПИСКЕ
2270 SN=I: REM ЕСТЬ В СПИСКЕ
2280 RETURN
2299 REM ***
2400 REM ПОДПРОГРАММА ИНИЦИАЛИЗАЦИИ ПЕРЕМЕННОЙ V$
2402 REM IN=1 — ПЕРЕМЕННОЙ ПРОИНИЦИАЛИЗИРОВАНА,
IN=0 — НЕТ
2404 REM V$ ЕСТЬ В СПИСКЕ ПЕРЕМЕННЫХ
2420 FOR I=1 TO 10: REM ПОИСК ПЕРЕМЕННОЙ В СПИСКЕ
2430 IF V$=V$ (I) THEN GOTO 2445: REM ПЕРЕМЕННАЯ НАЙДЕНА
2440 NEXT I
2445 IF IN (I)=1 THEN RETURN: REM ПЕРЕМЕННАЯ ПРОИНИ-
ЦИАЛИЗИРОВАНА
2450 IF IN (I)=1 THEN RETURN: REM ПЕРЕМЕННАЯ ПРОИНИ-
ЦИАЛИЗИРОВАНА
2460 REM В ПРОГРАММЕ БЛОК ИНИЦИАЛИЗАЦИИ ПЕРЕМЕННОЙ
2462 REM НАЧИНАЕТСЯ С ПРЕДЛОЖЕНИЯ 1700
2464 REM ИХ ФОРМАТ: ПРЕДЛОЖЕНИЕ INPUT И ВК
2465 IN (I)=1: REM ВВЕСТИ ПРИЗНАК ИНИЦИАЛИЗАЦИИ
2470 ON IGOSUB 1700, 1705, 1710, 1715
2480 RETURN
```

Примеры выполнения программы

```
RUN
***СПИСОК ЛОГИЧЕСКИХ ВЫВОДОВ***
ЛОГИЧЕСКИЙ ВЫВОД 1 РО
```

ЛОГИЧЕСКИЙ ВЫВОД 2 QU
ЛОГИЧЕСКИЙ ВЫВОД 3 PO
ЛОГИЧЕСКИЙ ВЫВОД 4 PO
ЛОГИЧЕСКИЙ ВЫВОД 5 PO
ЛОГИЧЕСКИЙ ВЫВОД 6 PO
ЛОГИЧЕСКИЙ ВЫВОД 7
ЛОГИЧЕСКИЙ ВЫВОД 8
ЛОГИЧЕСКИЙ ВЫВОД 9
ЛОГИЧЕСКИЙ ВЫВОД 10
ДЛЯ ПРОДОЛЖЕНИЯ НАЖМИТЕ ВК

СПИСОК ПЕРЕМЕННЫХ
ПЕРЕМЕННАЯ 1DE
ПЕРЕМЕННАЯ 2 DI
ПЕРЕМЕННАЯ 3 EX
ПЕРЕМЕННАЯ 4 GR
ПЕРЕМЕННАЯ 5
ПЕРЕМЕННАЯ 6
ПЕРЕМЕННАЯ 7
ПЕРЕМЕННАЯ 8
ПЕРЕМЕННАЯ 9
ПЕРЕМЕННАЯ 10
ДЛЯ ПРОДОЛЖЕНИЯ НАЖМИТЕ ВК

СПИСОК ПЕРЕМЕННЫХ УСЛОВИЯ
***УСЛОВИЕ 1
ПЕРЕМЕННАЯ 1 DE
ПЕРЕМЕННАЯ 2
ПЕРЕМЕННАЯ 3
ПЕРЕМЕННАЯ 4
***УСЛОВИЕ 2
ПЕРЕМЕННАЯ 1 DE
ПЕРЕМЕННАЯ 2
ПЕРЕМЕННАЯ 3
ПЕРЕМЕННАЯ 4
***УСЛОВИЕ 3

ПЕРЕМЕННАЯ 1 DE
ПЕРЕМЕННАЯ 2 DI
ПЕРЕМЕННАЯ 3
ПЕРЕМЕННАЯ 4
***УСЛОВИЕ 4
ПЕРЕМЕННАЯ 1 QU
ПЕРЕМЕННАЯ 2 GR
ПЕРЕМЕННАЯ 3 EX
ПЕРЕМЕННАЯ 4
***УСЛОВИЕ 5
ПЕРЕМЕННАЯ 1 QU
ПЕРЕМЕННАЯ 2 GR
ПЕРЕМЕННАЯ 3 EX
ПЕРЕМЕННАЯ 4
***УСЛОВИЕ 6
ПЕРЕМЕННАЯ 1 QU
ПЕРЕМЕННАЯ 2 GR
ПЕРЕМЕННАЯ 3
ПЕРЕМЕННАЯ 4
***УСЛОВИЕ 7
ПЕРЕМЕННАЯ 1
ПЕРЕМЕННАЯ 2
ПЕРЕМЕННАЯ 3
ПЕРЕМЕННАЯ 4
***УСЛОВИЕ 8
ПЕРЕМЕННАЯ 1
ПЕРЕМЕННАЯ 2
ПЕРЕМЕННАЯ 3
ПЕРЕМЕННАЯ 4

Примеры диалогов с программой

Пример 1:

***ВВЕДИТЕ ИМЯ ПЕРЕМЕННОЙ ЛОГИЧЕСКОГО ВЫВОДА? РО
ПОСЕТИТЕЛЬ ИМЕЕТ ЛИ УЧЕНОЕ ЗВАНИЕ? НЕТ
РО=НЕТ
***НОРМАЛЬНОЕ ЗАВЕРШЕНИЕ

```
Break In 685
Ok
```

Пример 2:

```
***ВВЕДИТЕ ИМЯ ПЕРЕМЕННОЙ ЛОГИЧЕСКОГО ВЫВОДА? PO
ПОСЕТИТЕЛЬ ИМЕЕТ УЧЕНОЕ ЗВАНИЕ? ДА
ПОСЕТИТЕЛЬ СДЕЛАЛ ОТКРЫТИЕ? НЕТ
QU=ДА
КАКОВ СРЕДНИЙ БАЛЛ ПОСЕТИТЕЛЯ В УЧЕБНОМ ЗАВЕДЕ-
НИИ? 3.0
СКОЛЬКО ЛЕТ ПОСЕТИТЕЛЬ РАБОТАЛ ПО СПЕЦИАЛЬНОСТИ? 2.8
PO=ИНЖЕНЕР ПО ЭКСПЛУАТАЦИИ
***НОРМАЛЬНОЕ ЗАВЕРШЕНИЕ
Break In 685
Ok
```

Пример 3:

```
***ВВЕДИТЕ ИМЯ ПЕРЕМЕННОЙ ЛОГИЧЕСКОГО ВЫВОДА? PO
ПОСЕТИТЕЛЬ ИМЕЕТ ЛИ УЧЕНОЕ ЗВАНИЕ? ДА
ПОСЕТИТЕЛЬ СДЕЛАЛ ОТКРЫТИЕ? НЕТ
КАКОВ СРЕДНИЙ БАЛЛ ПОСЕТИТЕЛЯ В УЧЕБНОМ ЗАВЕДЕ-
НИИ? 2.0
СКОЛЬКО ЛЕТ ПОСЕТИТЕЛЬ РАБОТАЛ ПО СПЕЦИАЛЬНОСТИ? 4.0
PO=ИНЖЕНЕР ПО ЭКСПЛУАТАЦИИ
***НОРМАЛЬНОЕ ЗАВЕРШЕНИЕ
Break In 685
Ok
```

Пример 4:

```
***ВВЕДИТЕ ИМЯ ПЕРЕМЕННОЙ ЛОГИЧЕСКОГО ВЫВОДА? QU
ПОСЕТИТЕЛЬ ИМЕЕТ УЧЕНОЕ ЗВАНИЕ? НЕТ
Break In 525
Ok
```

В последнем примере для условия DE=НЕТ нет правила, в которое переменная QU входила бы в качестве переменной логического вывода.

Пример 5:

***ВВЕДИТЕ ИМЯ ПЕРЕМЕННОЙ ЛОГИЧЕСКОГО ВЫВОДА? QU
ПОСЕТИТЕЛЬ ИМЕЕТ УЧЕНОЕ ЗВАНИЕ? ДА
QU=ДА
***НОРМАЛЬНОЕ ЗАВЕРШЕНИЕ
Break In 685
Ok

Рабочие таблицы системы обратных рассуждений

Приведенная ниже таблица предназначена для упрощения процесса разработки новой базы знаний и замены уже существующей. Она позволит читателю проверить взаимосвязь базы знаний и структур данных экспертной системы. В дальнейшем можно разработать входной компилятор, который будет читать базу знаний и автоматически преобразовывать ее в структуры данных. Однако это рекомендуется делать после того, как будут полностью усвоены все изложенные здесь положения. Рабочая таблица поможет создать и преобразовать переменные (переменная 11, переменная 12 и т. д.) и значения (значение 11, значение 12 и т. д.) новой базы знаний в предложения программы. Сначала создаются правила, затем они преобразуются в рабочую таблицу.

IF	Переменная 11	=	Значение 11	AND
	Переменная 12	=	Значение 12	AND
	Переменная 13	=	Значение 13	
THEN	Переменная 14	=	Значение 14	
IF	Переменная 21	=	Значение 21	AND
	Переменная 22	=	Значение 22	AND
	Переменная 23	=	Значение 23	
THEN	Переменная 24	=	Значение 24	
IF	Переменная 31	=	Значение 31	AND
	Переменная 32	=	Значение 32	
	Переменная 33	=	Значение 33	

Следующая таблица содержит список переменных условия (предложения 407–409).

CV$(1)	Переменная 11
CV$(2)	Переменная 12
CV$(3)	Переменная 13
CV$(5)	Переменная 21
CV$(6)	Переменная 22
CV$(7)	Переменная 23
CV$(9)	Переменная 31
CV$(10)	Переменная 32
CV$(11)	Переменная 33
CV$(13)	Переменная 41
CV$(15)	Переменная 43
CV$(17)	Переменная 51
CV$(18)	Переменная 52
CV$(19)	Переменная 53

Следующую таблицу можно использовать для включения переменных в список переменных (предложение 367). Имя переменной может быть занесено в список только один раз.

V$(1) = Переменная
V$(2) = Переменная
V$(3) = Переменная
V$(4) = Переменная
V$(5) = Переменная

Следующая таблица предназначена для включения в программу правил (предложения 1490–1611).

Части IF правила:

1500 IF Переменная 11 $= "Значение 11" AND
Переменная 12 $= "Значение 12" AND
Переменная 13 $= "Значение 13" THEN S=1
1520 IF Переменная 21 $= "Значение 21" AND
Переменная 22 $= "Значение 22" AND
Переменная 23 $= "Значение 23" THEN S=1
1540 IF Переменная 31 $= "Значение 31" AND
Переменная 32 $= "Значение 32" AND
Переменная 33 $= "Значение 33" THEN S=1
1560 IF Переменная 41 $= "Значение 41" AND
Переменная 42 $= "Значение 42" AND
Переменная 43 $= "Значение 43" THEN S=1
1580 IF Переменная 51 $= "Значение 51" AND
Переменная 52 $= "Значение 52" AND
Переменная 53 $= "Значение 53" THEN S=1

Части THEN правила:

VII. ВВЕДЕНИЕ В ЯЗЫК ИСКУССТВЕННОГО ИНТЕЛЛЕКТА ПРОЛОГ

Лабораторная работа 1
Работа с простейшими программами в системе Турбо-Пролог

Цель работы:

1. Знакомство со структурой Пролог-программ, использующих внешние или внутренние цели.
2. Получение навыков работы в оболочке системы Турбо-Пролог.
3. Знакомство с методикой отладки и трассировки программ.
4. Разработка простейших программ с использованием стандартных предикатов Турбо-Пролога.

1. Введение

Разработка простейших программ и их выполнение в среде системы Турбо-Пролог требуют начальных сведений по основным конструкциям языка Пролог-программ, которые должны содержать как минимум две секции:

— predicates — секцию описания структур отношений в программе в виде предикатов;

— clauses — секцию определения предикатов в виде набора фактов и правил.

Если в программе описаны только эти две секции, то предполагается, что цель (цели), решаемые программой, будут формулироваться в интерактивном режиме работы внутри системы программирования.

Работа в системе программирования предполагает наличие у пользователя элементарных знаний по работе в Турбо-оболочках,

таких как загрузка, ввод и редактирование программных файлов, запуск их на выполнение, а также отладка программ и конфигурирование системы.

Построение программ, не зависящих от среды выполнения, требует включения в их структуру еще одной секции — goal, обеспечивающей описание цели, решаемой программой.

При этом цель формулируется в виде запроса к программе, который представляет собой конъюнкцию подцелей, создавая необходимое множество связанных переменных. Если же одна из подцелей ложна, Пролог возвратится назад и просмотрит все возможные альтернативные решения предыдущих подцелей, а затем вновь пойдет вперед, но с другими значениями переменных. Такой процесс называется «поиском с возвратом».

Каждая программа на Прологе представляет собой текстовый файл, который для запуска из системы Турбо-Пролог должен иметь расширение PRO.

Содержимое какого-либо текстового файла или программы на языке Пролог может быть включено в другую программу в режиме ее компиляции. Для такой текстовой подстановки режима компиляции используется директива компилятора include, которая имеет следующий синтаксис:

include «dos_file_name»,

где dos_file_name — имя текстового файла системы MS DOS, включаемого в текущий программный файл. Имя файла может включать путь доступа к нему.

Включаемые файлы могут быть использованы только в естественных границах программы. Таким образом, ключевое слово include может появиться только там, где допускается одно из ключевых слов domains, predicates, clauses или goal.

Если в программе использована эта директива, то при компиляции в данное место текущей программы будет вставлено содержимое текстового файла как части текущей программы. Это позволяет некоторый набор уже отлаженных предикатов хранить в отдельном текстовом файле и включать его в другие программы при использовании в них обращений на эти, уже отлаженные, предикаты.

Включаемый файл сам может содержать директивы include. Однако включаемые файлы не должны использоваться рекурсивно, то есть так, чтобы тот же самый файл включался более одного раза в процессе компиляции. Использование многих уровней включаемых файлов требует больше памяти в процессе компиляции, чем если бы те же самые файлы были включены непосредственно в главную программу.

2. Загрузка системы Турбо-Пролог, ввод и запуск программ

2.1. Загрузите систему Турбо-Пролог. Для удобства работы в Windows перейдите в оконный режим.

Далее:

— ознакомьтесь с опциями главного меню и изучите их назначение;

— установите режим компиляции в память компьютера;

— сконфигурируйте, если необходимо, размеры и цветовые палитры всех окон;

— определите пути доступа к файлам, установив нужные для вас каталоги;

— выполненные установки запишите в файл конфигурации.

2.2. Войдите в режим редактирования системы Турбо-Пролог, воспользовавшись командой «РЕД» главного меню, и введите программу 1.

```
/*programm 1_1*/
predicates
likes (string, string). /*Описание предиката*/
clauses
likes («Sam», «Iris»). /*Факт*/
likes («Bill», «Travelling»). /*Факт*/
likes («Sam», X) if likes («Bill», X). /*Правило*/
```

Сохраните ее в своей папке с именем LAB1_1.PRO. Закончив ввод, выйдите из редактора в главное меню системы Турбо-Пролог, нажав клавишу «Esc».

2.3. Запустите программу на выполнение, выбрав команду «ВЫП» главного меню.

В данной программе нет секции goal, то есть в программе отсутствует внутренняя цель, определяющая решение конкретной задачи. Такие программы могут использоваться только в среде системы Турбо-Пролога. Поэтому после ее запуска на выполнение системой активизируется окно «Диалог», и появляется приглашение на ввод внешней цели (GOAL:).

3. Работа с Пролог-программами в режиме диалога

3.1. Внешние цели — это запросы к программе, формируемые пользователем в окне «Диалог».

Введите запрос GOAL: likes (Who, «Travelling»)

Объясните, что обозначает данный запрос, к каким элементам языка Турбо-Пролога следует отнести такие объекты запроса, как «Travelling», Who и likes.

3.2. Активизируйте введенный Вам запрос. Для этого надо после окончания его набора нажать клавишу «Enter». До нажатия «Enter» запрос можно редактировать. В ответ на ваш запрос в окне «Диалог» должны появиться сообщения.

Who=Bill

Who=Sam

2

GOAL:

Объясните полученный результат и смысловое назначение выводимых в окне «Диалог» сообщений.

3.3. Система запоминает последний из введенных запросов. Для того чтобы вызвать повторно предыдущий запрос, следует нажать функциональную клавишу F8. Вызовите повторно предыдущую цель и отредактируйте ее так, чтобы она имела вид GOAL: likes (Who, «Travelling», «Iris»)

Запустите ее на выполнение и объясните полученный результат.

3.4. Аналогичные действия проделайте по вводу и запуску запроса вида: GOAL: likes («Sam», X)

Объясните полученный результат, внесите изменения в запрос, чтобы он удовлетворял синтаксису языка Турбо-Пролог, и повторно запустите запрос на выполнение.

3.5. Измените описание предиката так, чтобы было ясно, между какими объектами реального мира отношение likes устанавливается. В частности, для рассматриваемого примера отношения likes определяются между некоторым лицом (person) и некоторым другим лицом или вещью (thing). Для учета введенного дополнения измените в секции predicates описание предиката на новое likes (person, thing) и запустите программу на выполнение.

При этом система выдаст сообщение об ошибке, и в окне редактирования курсором будет отмечено то место, где транслятор обнаружил ошибку. Текст сообщения об ошибке «Необъявленный домен или ошибка в написании» дает подсказку о том, что, перейдя к использованию нестандартных (то есть определенных пользователем) доменов, мы забыли объявить их типы.

Так как областью изменения обоих вновь определяемых доменов являются символьные данные (точнее, данные типа строки символов), то в программу должна быть добавлена секция domains, где должны быть объявлены нестандартные домены и их типы. Для данной программы она может иметь один из двух возможных видов:

```
domains
person=string
thing=string
```

или

```
domains
person, thing=string
```

Введите эти добавления в программу, запустите ее на выполнение и задайте любой из ранее вводимых запросов. Результат должен соответствовать предыдущему. Сохраните эту программу с именем LAB1_2.PRO.

4. Трассировка программ в среде системы Турбо-Пролога

Познакомиться с тем, как Пролог-система осуществляет поиск ответов на запросы, а также отследить последовательность согласования фактов и правил Пролог-программы, можно, используя пошаговый режим ее выполнения. Для перехода в режим

пошагового выполнения программы (трассировки) необходимо в программе использовать директиву trace.

4.1. Вставьте первой строкой программы директиву trace, которая при выполнении программы обеспечит трассировку всех предикатов. Запустите программу на выполнение и задайте один из ранее выполнявшихся запросов. Помните о возможности вызова предыдущего запроса с помощью F8.

4.2. Осуществите с использованием клавиши F10 пошаговое выполнение программы, тщательно отслеживая все перемещения курсора в окне редактирования и регистрируя все вводимые в окне трассировки сообщения.

4.3. Разберитесь в последовательности доказательства цели Пролог-системой. Для этого надо записать все сообщения режима трассировки и дать им подробное толкование в отчете по лабораторной работе.

5. Работа с программами, содержашими внутреннюю цель

5.1. Измените программу таким образом, чтобы один из ранее использованных запросов явился бы внутренней целью программы. Для этого в программу надо добавить еще одну секцию goal, где должна быть описана основная цель, решаемая программой. Пусть для нашего примера она будет иметь вид:

```
goal
likes (Who, «Travelling»)
```

и аналогична запросу 3.1 данной лабораторной работы.

5.2. Модифицированную программу запустите на выполнение. Если при запуске на решение у вас появилось сообщение о синтаксической ошибке при мигающем курсоре в соответствующем месте экрана, то посмотрите, не забыли ли вы о том, что каждое предложение Турбо-Пролога должно заканчиваться точкой.

5.3. Если в программе отсутствуют синтаксические ошибки, то после запуска ее на выполнение в окне диалога появится сообщение «Нажмите ПРОБЕЛ», которое свидетельствует о том, что программа отработала и запрос выполнен. Но тогда возникает вопрос: а где же результат запроса? Все дело в том, что в сформированной цели дается запрос о согласовании переменной Who

с предложениями программы, а об отображении или выводе полученных результатов ничего не говорится.

5.4. В отличие от диалогового режима работы, когда внешняя цель формулируется в виде запроса к программе, а Турбо-система сама управляет процессом поиска и отображения результатов в некотором стандартном виде, при формировании внутренней цели все эти функции возлагаются на пользователя.

Формирование внутренней цели программы требует от пользователя не только постановки запроса, но и обеспечения отображения его результатов на экране. Поэтому изменим цель, добавив еще одну подцель, обеспечивающую отображение терма на экране дисплея:

```
goal
likes (Who, «Travelling»), write (Who).
```

Запустим на выполнение программу, в которой цель представляет собой конъюнкцию уже двух подцелей. В результате выполнения этой программы в окне диалога выдается сообщение об одном из любителей путешествий.

```
Диалог
Bill
Нажмите ПРОБЕЛ.
```

А после нажатия клавиши «Пробел» происходит остановка программы и выход в главное меню системы. Сохраните эту дополненную программу с именем LAB1_2.PRO.

Связано это с тем, что при значении Who=«Bill» каждая из подцелей принимает значение «истина» и вся цель становится истинной, что приводит к окончанию процесса вывода. Таким образом, мы обнаружили еще одно существенное отличие в формулировке одного и того же запроса в виде внешней или внутренней цели. Если при внешней цели система сама, управляя поиском, ищет все удовлетворяющие запрос ответы, то при использовании внутренней цели ищется только первый из них.

5.5. Если цель решения задачи должна полностью соответствовать запросу и ее требуется включить в тело программы, а домены отношения для большей определенности должны быть переименованы, то в этом случае программа может быть представлена в виде:

```
/*programm 1_3*/
domains
person, thing=string
predicates
likes (person, thing)
goal
likes (Who, «Travelling»), write (Who), nl, fail. /*цель*/
clauses
likes («Sam», «Iris»). /*факт*/
likes («Bill», «Travelling»). /*факт*/
likes («Sam», X): — likes («Bill», X). /*правило*/
```

В этой программе домены отношения likes имеют имена person и thing, которые имеют тип строки символов. Цель решения состоит из четырех подцелей, соединенных запятыми. Запятая в предложениях Пролога равносильна логической функции «И» (and), то есть цель решения задачи представляется конъюнкцией подцелей. Цель будет достигнута, то есть примет значение «истина» (true), если каждая из подцелей будет истиной. Подцели данной программы содержат: один определенный пользователем предикат likes и три встроенных стандартных предиката Турбо-Пролога:

write (term) — выводит терм на дисплей;

nl — обеспечивает переход на новую строку;

fail — вызывает состояние неудачи при доказательстве целевого утверждения.

Включение в подцель дополнительных подцелей связано с тем, что задание цели внутри программы требует от пользователя не только формулирования запроса, но и обеспечения отображения его результатов на экране, а также обеспечения поиска всех удовлетворяющих запросу значений.

Так, если в цели отсутствует 4-я подцель, то будет найден только один любитель путешествий, а именно Билл. Поэтому добавление в цель предиката fail вызывает состояние неудачи при доказательстве целевого утверждения и переход к повторному его доказательству при иных значениях.

Отредактируйте вашу программу до состояния программы 3 и запустите ее на выполнение. Если цель достигнута, сохраните

программу в вашем рабочем каталоге на диске под именем LAB1_3.PRO.

6. Простейшая программа ввода-вывода данных

Рассмотрим еще один пример программы с внутренним описанием цели, которая демонстрирует простейшие возможности Турбо-Пролога по организациям интерфейса с пользователем на основе использования стандартных предикатов Турбо-Пролога. В систему Турбо-Пролог входит более 200 встроенных стандартных предикатов и более 12 стандартных доменов. В случае их использования нет необходимости объявлять их в программе.

6.1. Загрузить программу 1_4

```
/*programm 1_4*/
predicates
hello
goal
hello
clauses
hello:-
makewindow (1, 7, 7, «Dialog programm», 4, 54, 10, 22), nl,
write («Please, type your password»),
cursor (4, 5), readln (Password), nl,
write («Welcome», Password).
```

Разберитесь в ее структуре и запустите на выполнение. Ознакомьтесь с синтаксисом, семантикой и назначением стандартных предикатов (как, например, write, makewindow, cursor и т. п.), используемых в данной программе.

Установив режим трассировки, ознакомьтесь с последовательностью выполнения программы и действием стандартных предикатов.

6.2. Модифицируйте программу таким образом, чтобы окно создавалось в середине экрана и в другой цветовой палитре. Для этого необходимо изучить описание предиката makewindow и вычислить параметры цветовой палитры.

6.3. Исключить из программы внутреннюю цель, а описание и определение предиката hello изменить таким образом, чтобы можно было использовать цель в режиме диалога, например hello («Tom») или hello (« »).

6.4. Отлаженную по п. 6.3 программу записать на диск LAB1_4.PRO.

7. Построение простейшего интерфейса для вывода результатов запросов

Если вы правильно сформировали и хорошо отладили предикат hello (person) в программе LAB1_4.PRO, то с его помощью можно выводить в окно любые определенные в нем термы, что позволяет использовать его в качестве интерфейса для ранее разработанной программы LAB1_1.PRO. Возможно два варианта использования предикатов из этих двух программ:

— режим переносов в исходный модуль описаний и определений предиката путем копирования из другого файла;

— режим текстовой подстановки в исходный модуль файла, содержащего описание и определение требуемых предикатов.

В рамках данной лабораторной работы следует изучить оба этих варианта и составить две различные программы, реализующие поставленную цель.

7.1. Вариант 1. Загрузить программу LAB1_1.PRO. Войти в режим редактирования и, используя режим «копия извне», вызываемый нажатием клавиши F9, перенести в исходный файл описание и определение предиката hello из файла LAB1_4.PRO. Исходная цель приобретает качество первой подцели, например:

```
goal
hello («Who likes travelling: \n»), ..., ...,
```

где последовательность символов \n — это стандартная константа «перевод строки».

Изменив текст программы, отладить ее и записать в файл LAB1_4.PRO.

7.2. Вариант 2. Загрузить программу LAB1_1.PRO. Войти в режим редактирования и в первой строке программы ввести директиву:

include «LAB1_4.PRO»

В исходную внутреннюю цель в качестве первой подцели вставить предикат, аналогичный п. 7.1 или любой другой. Для совместной отладки основного и подгружаемого программных модулей использовать двухоконный режим работы редактора (F5 — вход в дополнительное окно редактирования, F10 — выход в меню).

7.3. Исследовать возможность применения пользовательского предиката hello ( ) вместо стандартного предиката write ( ) в разработанных программах.

8. Содержание отчета по лабораторной работе

Отчет по лабораторной работе должен содержать:

1. Результаты исследований по п. 3.1–3.5.
2. Протокол трассировки программы по п. 4 и пояснения к ней.
3. Пояснения по всем видам заданий, предлагаемых в п. 5, 6, 7.
4. Тексты программ LAB1_1.PRO, LAB1_2.PRO, LAB1_3.PRO, LAB1_4.PRO.

Лабораторная работа 2
Пролог-программы как простейшие базы данных и знаний

Цель работы:

1. Знакомство с организацией баз данных как совокупностью фактов.
2. Получение навыков организации явных и неявных баз данных.
3. Изучение способов построения универсальных запросов к базам.
4. Знакомство с представлением знаний в виде правил и процедур.

1. Введение

Система понятий для представления знаний несколько отличается от понятий для представления данных. Вместе с тем база

знаний (БЗ) способна хранить данные и как простую разновидность знаний в виде базы данных (БД). Запросы, которые формирует пользователь к базе, реализуются одним из двух возможных способов:

— сообщения, являющиеся ответом на запрос, хранятся в явном виде в БД, и процесс получения ответа представляет собой выделение подмножества значений из БД, удовлетворяющих запросу;

— ответ в явном виде в БД не существует и формируется в процессе логического вывода на основании имеющихся данных.

Последний случай принципиально отличается от технологий использования БД и рассматривается в рамках представления знаний, то есть информации, необходимой в процессе вывода новых фактов. В Пролог-программах вывод новых фактов возможен на основании набора правил, включаемых в программу и представляющих собой упрощенный вариант БЗ. Представление знаний в виде набора правил имеет следующие преимущества:

— простота создания и понимания отдельных правил;

— простота механизма логического вывода.

К недостаткам этого способа организации БЗ относится его отличие от человеческой структуры знаний.

2. Запросы к базе данных

Простейшая Пролог-программа представляет собой множество фактов, которое неформально называют базой данных.

Рассмотрим пример.

Пусть для хранения информации о студентах и их группах необходимо создать БД со структурой отношения УЧИТСЯ (ИМЯ, ГРУППА). При этом атрибут ИМЯ описывает домен данных типа строки символов, а атрибут ГРУППА — домен целочисленных данных.

```
/*programm 2_1*/
domains
name=string
group_num=integer
predicates
```

```
study (name, group_num)
clauses
study («Orlov», 101).
study («Pavlov», 211).
study («Sokolov», 101).
```

Для решения поставленной задачи в Пролог-программе:

— исходное отношение описывается предикатом аналогичной структуры;

— в секции domains задаются области изменения каждого аргумента предиката;

— каждый кортеж данного отношения представляется в секции clauses в виде факта.

Одним из примеров программы, реализующих данную задачу, может быть приведенная здесь программа 2_1.

После того как данная программа введена в систему Турбо-Пролог с помощью текстового редактора или загружена с диска, а затем запущена на выполнение командой «ВЫП» главного меню системы, активизируется окно диалога (появляется сообщение GOAL:), и система готова к приему запросов. Синонимом слова запрос является слово цель. В переводе с английского goal обозначает цель.

2.1. Простые запросы

Простой запрос состоит из имени предиката, за которым располагается список аргументов.

Если в запрос входят только константы (то есть атомы и числа), то такой запрос называется запросом с константами, и на него система выдает только один из двух ответов — True или False. Ответ True свидетельствует о том, что система доказала истинность запроса в соответствии с множеством фактов, загруженных в нее в данный момент. Ответ False — это невозможность системы доказать истинность запроса. Приведенные в окне диалога запросы соответствуют вопросам:

— «Учится ли Павлов в 211-й группе?» Ответ на него можно трактовать так: «Да, Павлов учится в 211-й группе».

— «Учится ли Павлов в 101-й группе?», ответ на который: «Нет, Павлов не учится в 101-й группе».

Однако применение запросов с константами весьма ограничено, поэтому наиболее часто применяются запросы, которые используют переменные — запросы с переменными.

Переменная — это вид терма, начинающийся с заглавной буквы. В запросе, содержащем переменную, неявно спрашивается о том, существует ли хотя бы одно значение переменной, при котором запрос будет истинным.

То есть переменные в запросах квантифицированы экзистенциально. Если это иметь в виду, то приведенный в окне диалога запрос можно прочесть так: «Существует хотя бы один студент, который учится в 101-й группе?» Запрос будет истинным, если такое лицо будет найдено в текущей базе.

Система пытается унифицировать (то есть согласовать) аргумент запроса с аргументами фактов, входящих в базу данных «study». Запрос окажется успешным при его сопоставлении с первым же фактом, поскольку атом «101» в запросе унифицируется с атомом «101» первого факта, а переменная «Who» унифицируется с атомом «Орлов», входящим в этот факт. В результате данного процесса переменная «Who» примет значение атома «Орлов», сообщение о чем выводится в окне диалога.

Далее происходит сопоставление запроса с другими фактами БД, и обо всех успешных унификациях и их количестве выдается сообщение в окне диалога. Говорят, что переменная конкретизируется, когда при выполнении запроса она унифицируется с некоторым значением.

2.2. Составные запросы

Составные запросы образуются из простых, соединенных между собой запятыми. Каждый простой запрос называется подцелью. Для того чтобы составной запрос оказался истинным, необходимо, чтобы каждая из его подцелей была истинной. Введем составной запрос, который будет соответствовать вопросу: «Есть ли такая группа, где вместе учатся Орлов и Соколов?»

```
Goal: study («Orlov», X), study («Sokolov», X)
X=101
1
Goal:
```

Переменная X входит в обе подцели, то есть для истинности всего запроса требуется, чтобы вторые аргументы в каждой из подцелей принимали одни и те же значения.

Использование констант в аргументах запроса эквивалентно заданию входных параметров в программе. Использование переменных эквивалентно требованию получить от программы выходные данные.

2.3. Запросы с анонимными переменными

Символ подчеркивания (_) выступает в качестве анонимной переменной, которая предписывает системе проигнорировать значение аргумента. Эта переменная унифицируется с чем угодно, но не обеспечивает выдачу на экран данных. Каждая анонимная переменная, входящая в запрос, отличается от других переменных этого запроса.

Посмотрим, что будет, если ввести запрос study (Student, _). При выполнении этого запроса будут выданы все возможные значения первого аргумента предиката study ( ), вне зависимости от того, какие значения будет принимать второй аргумент. Таким образом, данный запрос соответствует вопросу:

«Существует ли хотя бы один студент, который учится в какой-либо из групп?»

Сформированный к программе запрос аналогичен желанию получить данные обо всех студентах, которые учатся во всех группах института.

```
Goal: study (Student, _)
Student=«Orlov»
Student=«Pavlov»
Student=«Sokolov»
3
Goal:
```

3. Статические и динамические базы данных

Как уже отмечалось, множество фактов Пролог-программы можно рассматривать как базу данных, к которой могут формулироваться любые произвольные запросы. Однако данные такой БД жестко связаны с самой Пролог-программой. Любое

манипулирование БД требует изменения или добавления того или иного факта в текст программы. В связи с этим такие базы данных называют статическими.

Так, для ввода информации о зачислении в институт в 101-ю группу Воробьева, нам потребуется войти в режим редактирования программы и добавить новый факт.

```
/*programm 2_2*/
domains
name=string
group_num=integer
predicates
study (name, group_num)
clauses
study («Orlov», 101).
study («Pavlov», 211).
study («Sokolov», 101).
study («Vorobev», 101). /*Новый факт в базе данных*/
```

Выполните следующие задания:

1. Повторите запрос: Кто является студентом (учащимся) с учетом того, что добавился новый факт в базе данных. Это запрос с использованием анонимной переменной

```
Goal: study (Student, _)
Student=«Orlov»
Student=«Pavlov»
Student=«Sokolov»
Student=«Vorobev»
4
Goal:
```

2. Поменяйте местами два последних факта в базе данных (Соколова и Воробьева), повторите запрос и поясните, что изменится при выводе результата.

Если затем в режиме выполнения мы зададим запрос о студентах всех групп, то получим информацию уже о четырех студентах. Причем обратите внимание, что в ответе на запрос данные об Орлове будут выводиться в той по номеру строке, каким по счету в программе является факт об учебе Орлова в 101-й группе.

Попробуйте, войдя в режим редактирования, переставить факты местами. Затем, войдя в режим выполнения, сформулировать запрос к базе. Заметьте, что система находит ответы в БД в том порядке, в каком они введены в программе. Это еще одна особенность работы со статическими БД.

Динамические базы данных — это БД, в которых факты могут модифицироваться во время выполнения программы или выбираться из файла. Для работы с такими БД в тексте программы в секции database должны быть декларированы предикаты для описания структуры динамической БД. Работа с такими БД будет описана ниже.

4. Явные и неявные базы данных. Правила логического вывода

Рассмотренная выше база данных об учебе студентов в группах является явной БД, так как она составлена из фактов, аргументы которых константы. При работе даже с такой простой БД у человека хватает интеллекта установить между имеющимися данными и совсем иные отношения, кроме учебы студента в конкретной группе.

Так, человеку хватает знаний на то, чтобы назвать двух студентов коллегами, если они учатся в одной и той же группе. Вместе с тем, работая с БД study ( ), для определения коллег Орлова пользователю системы необходимо:

— определить номер группы (N), в которой учится Орлов;

— найти всех студентов, которые учатся в группе с номером N;

— исключить из полученного списка самого Орлова;

— сформировать все сочетания Орлова с лицами из полученного списка.

Таким образом, у пользователя появляется возможность свои представления о понятии «коллега» сформировать в виде составного запроса к явной БД и получить ответ на интересующий его вопрос. И такая ситуация будет повторяться при каждом использовании понятия «коллега» при формировании запросов, таких как «Являются ли Орлов и Соколов коллегами?», «Кто коллега Соколова и Воробьева?» и т. д. Ясно, что такая процедура длительна, требует от пользователя системы должной квалификации.

Вместе с тем эти несложные познания по преобразованию отношений и свои представления о понятии «коллега» пользователь может передать Пролог-программе, описав свои знания набором декларативных правил.

Для этого можно ввести новое отношение КОЛЛЕГА (СТУДЕНТ 1, СТУДЕНТ 2), определив его как пару разных студентов, которые учатся в одной и той же группе. Новое отношение следует описать в секции predicates двуместным предикатом со структурой, аналогичной отношению КОЛЛЕГА ( ).

Аргументы предиката должны принимать значения из области символьных данных, а сам предикат в секции clauses определяется на основании логического правила вывода:

Любые два студента СТУДЕНТ 1 и СТУДЕНТ 2 являются коллегами

ЕСЛИ

существует такая группа N_групп, что учится СТУДЕНТ 1 в группе N_групп

И

учится СТУДЕНТ 2 в группе N_групп

И

студенты СТУДЕНТ 1 и СТУДЕНТ 2 различны.

Выполнив все необходимые действия по модификации исходной программы, мы получим программу, которая содержит не только факты, но и знания в виде правил манипулирования этими фактами. Это позволяет к правилам применять те же возможные типы запросов, что и к фактам.

Если нас интересуют все коллеги студента Орлова, то запрос надо задавать в виде:

Goal: colleague («Orlov», Who),

а если интересуют общие коллеги и Орлова и Соколова, то запрос будет иметь вид:

Goal: colleague («Orlov», X), colleague («Sokolov», X)

и система найдет только один ответ о студенте по фамилии Воробьев, так как именно он учится в одной группе со студентами Орловым и Соколовым.

/*programm 2_3*/

```
domains
name=string
group_num=integer
predicates
study (name, group_num)
colleague (name, name)
clauses
colleague (Stud1, Stud2):- study (Stud1, N), study (Stud2, N),
Stud1<>Stud2.
study («Orlov», 101).
study («Pavlov», 211).
study («Sokolov», 101).
study («Vorobev», 101).
```

Добавлено новое правило, которое представляет собой неявную базу данных, а явная база данных остается без изменения.

Выполните следующие задания:

1. Выполните запрос: кто является коллегами (одногруппниками) Орлова?

2. Выполните запрос: кто является коллегами Орлова и Воробьева?

Из изложенного следует, что если БД study ( ) — это явная база данных, так как составлена из фактов, аргументы которых константы, то база colleague ( ) — это неявная база данных, поскольку правило для ее формирования определено с использованием переменных, значения которых зависят от подцелей этого правила. С точки зрения пользователя, формирующего запрос, не имеет значения, является база явной или неявной.

5. Использование структур в качестве доменов отношений

Турбо-Пролог позволяет создавать объекты, компонентами которых являются другие объекты. Причем сложные объекты рассматриваются и обрабатываются так же, как и простые. Это сильно упрощает составление программ и организацию баз данных.

В предыдущем разделе мы сформировали неявную базу данных о коллегах на основе наших представлений о понятии «коллега». При этом, формулируя правила для представления этого

понятия, мы ограничились только представлением о коллегах как о людях, которых объединяет совместная учеба, то есть как об одногруппниках.

Более точно определить понятие «коллеги» можно как пару лиц, объединенных единой целью, интересом, предметом деятельности и т. д. При такой формулировке понятия «коллеги» мы можем для его реализации сформировать отношение:

ОБЪЕДИНЯЕТ (ЛИЦО1, ЛИЦО2, ПРЕДМЕТ)

или в синтаксисе языка Турбо-Пролога

unite (name, name, object).

И тогда мы могли бы, например, называть людей коллегами, если:

— объединяет Сэма и Билла работа;

— объединяет Соколова и Орлова общее хобби, которым является музыка;

— объединяет Орлова и Тима проект по новым системам для IBM;

— объединяет Козлова и Соколова совместная работа.

У нас не возникнет никаких сложностей при представлении первого предложения в виде факта на Прологе, который будет иметь вид

unite («Sam», «Bill», labour).

Но если второе предложение записать в аналогичной форме, то есть

unite («Sokolov», «Orlov», «hobby music»),

так как hobby является некоторым свойством объекта совместной деятельности, а sport является конкретным значением этого свойства, то hobby — это атрибут объекта object, а sport — конкретный экземпляр этого атрибута. При таком подходе единственный вариант записи второго предложения будет

unite («Sokolov», «Orlov», hobby (music)),

где hobby (music) — это составной терм или структура Турбо-Пролога. Тогда по аналогии можно записать Пролог-факты для второго и третьего предложений

unite («Orlov», «Tim», project («New system», ibm)),

unite («Kozlov», «Sokolov», labour),

Таким образом, можно сделать вывод о том, что во введенном для определения понятия «коллеги» отношении unite первые два

домена являются простыми объектами, а третий — это сложный объект, атрибуты которого сами являются объектами.

Описание данного отношения на Прологе в виде предиката и определение областей изменения его аргументов будет иметь вид:

```
domains
name, firm=symbol
object=labour; hobby (name); project (name, firm)
predicates
unite (name, name, object)
```

где символ «;» (точка с запятой) эквивалентен логической операции «ИЛИ» и в данном описании использован для того, чтобы показать, что домен object может иметь одну из возможных структур, описанных для него в области domains.

Задание 1

1. Откорректируйте программу 2.3 и сохраните ее как программу 2.4, включив в нее описание предиката unite и определив его для четырех фраз, приведенных в данном разделе.

```
/*programm 2_4*/
domains
name, firm=symbol
group_num=integer
object=labour; hobby (name); project (name, firm)
predicates
study (name, group_num)
colleague (name, name)
unite (name, name, object)
clauses
colleague (Stud1, Stud2):- study (Stud1, N), study (Stud2, N),
Stud1<>Stud2.
study («Orlov», 101).
study («Pavlov», 211).
study («Sokolov», 101).
study («Vorobev», 101).
unite («Sam», «Bill», labour).
unite («Sokolov», «Orlov», hobby (music)).
unite («Orlov», «Tim», project («New system», ibm)).
unite («Kozlov», «Sokolov», labour).
```

2. Сформируйте запросы, соответствующие вопросам:
— «Кого объединяет совместный труд?»,
— «Есть ли пара любителей шахмат?»,
— «Кто является коллегой Сэма?»,
— «Для кого (какой фирмы) и с кем Тим делает проект?»

3. Сформулируйте самостоятельно еще три составных запроса к базе и введите в программу.

4. Все вопросы, соответствующие им запросы и результаты вывода представьте в отчете по лабораторной работе.

6. Процедуры как элемент представления знаний

Смысл предложений Пролог-программ может быть понят либо с позиций декларативного подхода, либо с позиций процедурного подхода. Декларативный смысл подчеркивает статическое существование отношений. Порядок следования подцелей в правиле не влияет на декларативный смысл этого правила.

При процедурной трактовке программы подчеркивается последовательность шагов, которые выполняются при обработке запроса. В этом случае приобретает значение порядок следования подцелей в правиле.

Множество предложений, имеющих одно и то же имя предиката с одинаковым количеством аргументов, называют процедурой. Когда обрабатывается запрос к процедуре, то он анализирует фразы, образующие процедуру, в том порядке, как они в ней представлены. Считается, что между правилами процедуры неявно присутствует соединитель «ИЛИ».

В предыдущих разделах мы сформировали два подхода к представлению наших знаний о понятии «коллега». С одной стороны, коллегами являются сослуживцы, то есть любая пара лиц, которые работают вместе, с другой — любая пара лиц, которая объединена объектом общей деятельности. Мы подошли к тому, чтобы два наших знания о понятии «коллега» объединить в одно.

Представьте себе, что одну из формулировок какого-либо понятия (в нашем случае это коллега) мы получили от одного эксперта, имеющего свои знания этой проблемы, а вторую — от эксперта, имеющего иные знания этой же проблемы. Объединяя в Про-

лог-программе два знания одной и той же проблемы, мы получим систему, которая знает больше каждого отдельного эксперта.

Понятие коллеги, удовлетворяющее обеим точкам зрения, может быть описано отношением вида:

ПОЛНЫЙ_КОЛЛЕГА (ЛИЦО1, ЛИЦО2,
ПРЕДМЕТ ОБЩЕЙ _ДЕЯТЕЛЬНОСТИ),

которое на Прологе будет описано предикатом all_colleague, структура которого будет полностью аналогична структуре предиката unite, а определить его можно в виде процедуры, содержащей три декларации предиката all_colleague.

```
predicates
all_colleague (name, name, object)
clauses
all_colleague (X, Y, Z):- colleague (X, Y), Z=labour.
all_colleague (X, Y, Z):- unite (X, Y, Z).
all_colleague (X, Y, Z):- unite (Y, X, Z).
```

С декларативной точки зрения описание процедуры «полный коллега» можно прочитать так:

Для любых двух лиц X и Y и любой общей деятельности Z
X и Y являются коллегами по общей деятельности Z
ЕСЛИ
X и Y являются сослуживцами
И
общая их деятельность Z — это труд
ИЛИ
X объединяет с Y общая деятельность Z
ИЛИ
Y объединяет с X общая деятельность Z

Последнее правило устраняет асимметрию отношения unite по отношению к лицам, объединенным общей деятельностью. Действительно, если Козлов является коллегой Соколова по работе, то, очевидно, что и Соколов является коллегой Козлова по работе.

Задание 2

1. Измените программу, добавив в нее описание предиката all_colleague и процедуру для его определения.

2. Определите состав явных и неявных баз, используемых в данной программе, и опишите их структуру.

3. Введите запрос «Кто является коллегой Тима?» Третье правило процедуры заключите в /*...*/ и повторите запрос. Объясните, почему разные ответы.

4. В чем заключается разница в выполнении запросов unite («Sokolov», Who, X) и all_colleague («Sokolov», Who, X)? Введите еще ряд произвольных запросов.

5. Определите, кто является коллегой Воробьева и кто коллегой Павлова, а кто — коллега Козлова.

6. Все вопросы, запросы и результаты вывода привести в отчете по работе.

7. Целостность и непротиворечивость баз данных и знаний

Этими двумя сложными понятиями, одними из основных при построении баз данных и знаний, мы постоянно будем оперировать дальше. Здесь же остановимся лишь на одном небольшом примере, иллюстрирующем их важность.

При выполнении п. 5 задания 2 мы определили, что у Козлова только один коллега — Соколов, связанный с ним совместным трудом. Вместе с тем у Соколова, кроме Козлова, есть еще два коллеги, которые связаны с ним совместной учебой.

Но из этих двух посылок и наших представлений о понятии коллеги любому человеку ясно, что Козлов учится в той же группе, что и Соколов. А если это так, то у него имеется больше одного коллеги, в отличие от ответа системы. То есть у нашей системы не хватает интеллекта на такой вывод.

А на запрос study («Козлов», institut) система вообще даст отрицательный ответ. Налицо противоречивость данных. Частично исправить ситуацию можно, если доопределить предикат study в виде

study (Stud1, N):- unite (Stud1, Stud2, labour), study (Stud2, N).

Тогда на запрос о номере группы у Козлова и его коллегах система будет давать более точные ответы. Но ведь в базе study ( ) отсутствуют данные о Козлове в виде фактов, то есть в явном виде. Стало быть, после нашего доопределения эта база стала не совсем

явной, так как часть данных хранится в явном виде, а часть выводима из других на основе правил. В первом приближении — это уже прообраз базы знаний. Текст программы 2_5 со всеми добавлениями, введенными по ходу работы, имеет вид:

```
/*Программа 2_5*/
domains
name, firm=symbol
group_num=integer
object=labour; hobby (name); project (name, firm)
predicates
study (name, group_num)
colleague (name, name)
unite (name, name, object)
all_colleague (name, name, object)
clauses
colleague (Stud1, Stud2):- study (Stud1, N), study (Stud2, N),
Stud1<>Stud2.
all_colleague (X, Y, Z):- colleague (X, Y), Z=labour.
all_colleague (X, Y, Z):- unite (X, Y, Z).
all_colleague (X, Y, Z):- unite (Y, X, Z).
unite («Sam», «Bill», labour).
unite («Sokolov», «Orlov», hobby (music)).
unite («Orlov», «Tim», project («New system», ibm)).
unite («Kozlov», «Sokolov», labour).
study («Orlov», 101).
study («Pavlov», 211).
study («Sokolov», 101).
study («Vorobev», 101).
study (Stud1, N):- unite (Stud1, Stud2, labour), study (Stud2, N).
```

8. Содержание отчета по лабораторной работе

1. Текст программы с набором фактов, который применяется при запросах.

2. Запросы, сформированные самостоятельно при изучении п. 2 и 3 данной лабораторной работы, а также результаты их выполнения.

3. Результаты выполнения задания 1 и 2 данной лабораторной работы.

4. Результаты выполнения индивидуального задания, выданного преподавателем.

Лабораторная работа 3
Управление ходом выполнения программ в системе Турбо-Пролог

Цель работы:

1. Познакомиться с процессами унификации и поиска с возвратом.
2. Изучить правила унификации термов.
3. Изучить работу системы Турбо-Пролог при выполнении запроса.
4. Ознакомиться с методом отката после неудачи.

1. Работа системы Турбо-Пролог при выполнении запросов

Запрос к системе — это всегда последовательность, состоящая из одной или нескольких целей. Для ответов на запрос система пытается достичь всех целей, то есть показать, что утверждения, содержащиеся в запросе, истинны в предположении, что все отношения программы истинны. Другими словами, достичь цель — это показать, что она логически следует из фактов и правил программы, а если в запросе есть переменные, то еще и конкретизировать их, то есть найти те конкретные объекты, которые, будучи подставленными вместо переменных, обеспечат достижение цели.

Для этого система опускается в структуру программы так глубоко, как это необходимо, чтобы найти факты, требующиеся для доказательства истинности запроса. Затем система вернется в исходное состояние, доказав или оказавшись не в состоянии доказать истинность запроса. В основе работы системы лежит рекурсивный процесс унификации (то есть сопоставления с образцом) и доказательства подцелей.

При задании запроса система просматривает всю программу в поисках первого предложения, заголовок которого будет унифицироваться с запросом. Для того чтобы запрос и заголовок предложения унифицировались между собой, необходимо:

— совпадение у них имени предиката;

— совпадение количества аргументов предиката;

— возможность унификации всех аргументов предиката (правила унификации термов приведены ниже).

Если предложение, которое унифицируется с запросом, будет обнаружено, то оно начинает обрабатываться:

— если тело предложения является пустым (то есть это факт), то запрос сразу оказывается успешным, переменные запроса конкретизируются объектами факта, и это решение помечается указателем возврата.

— если тело предложения содержит подцели, то каждая из них последовательно обрабатывается слева направо, точно так же, как исходный запрос.

Если система в тексте программы не находит предложения, унифицирующегося с запросом, то она:

— возвратится к последней успешно доказанной подцели;

— ликвидирует конкретизацию всех переменных, которая явилась результатом успешной обработки этой под цели, то есть делает переменные свободными;

— приступает к поиску во множестве предложений текущей программы; заголовка другого предложения, унифицирующегося с данной подцелью.

Такая процедура обработки запроса получила название поиск с возвратом. Применяя ее, при успешном выполнении запроса система выдает:

— либо значение всех переменных, входящих в состав запроса, которые были конкретизированы в результате обработки;

— либо слова «да» или «нет», если переменные в запросе отсутствовали.

Если запрос был сформирован как внешняя цель решения задачи, то система после найденного первого ответа вернется в точку, помеченную указателем возврата, и попытается найти иной ответ.

2. Унификация термов

Наиболее важной операцией над термами является унификация, при которой осуществляется: конкретизация переменных, доступ к структурам данных через общий механизм согласования и определенные виды тестов на равенство. Процесс унификации соответствует передаче параметров в других языках программирования. Унификация термов регулируется следующими правилами:

1. Свободная переменная унифицируется с константой или структурой. В результате этого переменная становится конкретизированной, то есть принимает значение этой константы или структуры.

Goal: X=Sam

X=Sam

2. Переменная унифицируется с переменной, при этом обе они становятся одной и той же переменной.

Goal: X=Y

X=1

Y=1

3. Анонимная переменная унифицируется с чем угодно.

? _=Sam

4. Константа может быть унифицирована с другой константой, если они идентичны, или со свободной переменой соответствующего типа.

Goal: Sam=Sam

True

5. Структура унифицируется с другой структурой при условии, что их функторы одинаковы, а аргументы могут попарно унифицироваться.

friend (X)=friend (Sam)

X=Sam

Для того чтобы познакомиться с процессом унификации различных классов объектов Пролога, рассмотрим выполнение приведенной ниже программой нескольких запросов.

```
/*программа 3_1*/
domains
```

```
title, author=symbol
pages=integer
publications=book (title, pages)
predicates
written_by (author, publications)
long_novel (publications)
clauses
written_by («В. Петров», book («Информационные системы», 687)).
written_by («Д. Нидерст», book («Web-мастеринг для профессионалов», 569)).
long_novel (book (Title, Length)): — written _by (_, book (Title, Length)), Length < 600.
```

Задание 1

Отладить программу 3_1 с тремя запросами, которые будут приведены далее. Сохранить программу с именем LAB3_1.PRO.

а) Рассмотрим запрос вида: written_by (X, Y). При решении задачи система должна поочередно согласовать цель с предложениями программы, пытаясь достичь соответствия между параметрами X и Y, с одной стороны, и параметрами предложений программы — с другой, выполняя операцию унификации.

Так как в данном запросе переменные X и Y являются свободными и могут согласовываться с любой константой, то самое первое предложение для предиката written_by дает желаемое соответствие

```
written_by (X, Y)
written_by («В. Петров», book («Информационные системы», 687)),
```

то есть X конкретизируется константой «В. Петров», а Y принимает значение структуры book («Информационные системы», 687).

Система Турбо-Пролога помечает эту точку указателем возврата и выдает на экран сообщение.

```
X=«В. Петров» Y=book («Информационные системы», 687)
```

Так как мы задавали запрос как внешнюю цель, система возвращается в точку, помеченную указателем возврата, и продолжает начиная с этой точки процесс унификации и находит второе

предложение, которое также может быть согласовано с запросом. После унификации переменных система выдает

X=«Д. Нидерст» Y=book («Web-мастеринг для профессионалов», 569)

2

и заканчивает процесс унификации.

б) Если введем запрос written_by (X, book («Web-мастеринг для профессионалов», Y)), то попытка унификации переменных с первым предложением программы будет выглядеть так:

written_by (X, book («Web-мастеринг для профессионалов», Y)

written_by («В. Петров», book («Информационные системы», 687)).

Так как X свободна, она примет значение константы «В. Петров» и сделает попытку установить соответствие между двумя структурами. Но составной объект согласуется с другим объектом при условии, что они оба имеют один и тот же функтор, одинаковое количество аргументов и все аргументы могут быть попарно унифицированы. Однако константа «Web-мастеринг для профессионалов» может быть унифицирована только со свободной переменной или сама с собой. Так как между первыми двумя компонентами структуры book соответствие невозможно, то формируется признак неудачи и система пытается согласовать цель со следующим предложением программы, переходя к проверке соответствия между:

written_by (X, book («Web-мастеринг для профессионалов», Y))

written_by («Д. Нидерст», book («Web-мастеринг для профессионалов», 569)).

Свободная переменная унифицируется с константой «Д. Нидерст». Обе структуры имеют один и тот же функтор book, содержат равное число компонентов, и первые компоненты обеих структур — одинаковые константы. То есть эти структуры могут быть унифицированы, и при этом константа 569 унифицируется с переменной Y, то есть цель достигнута, и Турбо-Пролог выводит сообщение:

X=«Д. Нидерст» Y=569

в) Наконец, рассмотрим выполнение запроса: long_novel (X). Прежде всего система пытается отыскать предложения, заголовки которых согласуются с запросом:

long_novel (X)

long_novel (Title):- written_by (_, book (Title, Length)), Length<600

После этого согласовываются левая и правая части правила. Переменные X и Title согласуются, так как они свободны и становятся одной и той же переменной. Затем Турбо-Пролог объявляет первое предложение указанного выше правила подзадачей и делает попытку ее унификации:

written_by (_book (Title, Length))

written_by («В. Петров», book («Информационные системы», 687)).

Так как анонимная переменная согласуется с любым объектом и структуры book также согласуются, эти два предиката могут быть унифицированы. В результате этого переменная принимает значение «Информационные системы», а переменная Length становится равной 687.

После этого делается попытка согласовать вторую подзадачу тела правила, а именно: Length < 600. Перед попыткой унификации связанная переменная Length заменяется своим численным значением 687. Так как выражение 687<600 ложно, Турбо-Пролог делает возврат назад к уже доказанной подцели и пытается его предсказать. То есть снова пытается унифицировать первую подзадачу

written_by (_book (Title, Length)),

используя следующий из имеющихся фактов:

written_by (_, book (Title, Length))

written_by («Д. Нидерст», «Web-мастеринг для профессионалов», 569)),

который связывает Title с «Web-мастеринг для профессионалов» и Length с 569. В данном случае Length <600, то есть вторая подзадача также становится истиной. Правило полностью согласовано при полученных значениях переменных, задача решена, о чем выдается сообщение

X=book («Web-мастеринг для профессионалов», 569)

1

Исходная цель является полностью доказанной, и функционирование системы Турбо-Пролог по выдаче ответа на введенный запрос заканчивается. Система готова к приему новых запросов.

3. Поиск с возвратом при выполнении Пролог-программ

Значение метода поиска с возвратом, реализованном в Пролог-системе, позволит вам правильно составлять Пролог-программы и управлять поиском решений. Более подробно рассмотрим этот процесс на примере программы 2_5 из лабораторной работы 2.

Задание 2

1. Загрузите в систему Турбо-Пролог программу 2_5 (файл LAB2_5.PRO) из лабораторной работы 2, войдите в режим редактирования и введите директиву пошагового выполнения программы. Для сокращения пространства поиска последнее правило в базе данных study ( ) сделайте комментарием.

2. Сформируйте запрос о поиске всех коллег Соколова и в режиме пошаговой трассировки отследите процесс поиска решений.

3. Сохраните программу с именем LAB3_2.PRO.

Решая любую задачу, Пролог строит дерево подзадач, отмечает, какие подзадачи решены, а какие еще нет. Так, при решении поставленной в задании задачи дерево подзадач будет иметь вид:

colleague (« », Y)

study (Stud1, N)), study (Stud2, N)) Stud1<> Stud2

При решении любых подзадач Турбо-Пролог использует два основных правила:

— поиск решений подзадач производится справа налево;

— выбор предикатов для рассмотрения при решении каждой подзадачи осуществляется в том порядке, в котором они появляются в программе.

При этом если какая-либо из подзадач не может быть успешно согласована, то выполняется возврат (или откат), чтобы найти другие возможные пути ее вычисления.

Рассмотрим процессы поиска с возвратом и унификации, реализуемые Пролог-системой при выполнении запроса colleague (« », Y). На рис. 2 приведена схема работы Пролог-системы при выводе ответа на поставленный запрос. Решение всей задачи представляет собой последовательность четко определенных этапов (шагов) выполнения задачи:

1. После ввода запрос помещается в вершину стека активных запросов, система сразу приступает к поиску предложений с тем же самым именем предиката, что и у запроса.

2. Анализируя найденное предложение, система унифицирует каждый аргумент запроса с соответствующим аргументом предложения (правила). После унификации запроса с заголовком система переходит к телу предложения и унифицирует все его переменные в соответствии заголовком правила.

3. Тело правила составное, поэтому первая подцель помещается в стек запросов, становится активным запросом и начинает обрабатываться как запрос. На данном этапе стек имеет вид:

colleague («Соколов», Y)

study («Соколов», N) ← активный запрос

Новый активный запрос приступает к поиску предложений с тем же самым именем предиката, что и активного запроса.

Если попытка унификации запроса с заголовком фразы заканчивается неудачей, то система переходит к анализу следующего предложения. Этот процесс продолжается до тех пор, пока не обнаружится предложение, которое будет унифицироваться с запросом. Если его не будет, то запрос завершится неудачей.

Если унифицированное предложение не найдено и оно остается фактом, то подцель сразу же оказывается успешной, переменные унифицируются с фактом и в стек запросов заносится подцель, которая становится активной:

colleague («Соколов», Y)

study («Соколов», 101)*(истина)

study (Y, 101) ← активный запрос

Данные шаги аналогичны п. 5–6 с той лишь разницей, что если в первом случае поиск в базе study ( ) осуществляется по фамилии, то во втором поиск в той же базе выполняется по номеру группы. Первое предложение базы study ( ), являющееся фактом, делает активный запрос успешным. Его система помечает маркером возврата, унифицирует переменную Y с константой «Орлов», и третья подцель загружается в стек запросов:

colleague («Соколов», «Орлов»)

study («Соколов», 101)*(истина)

study («Орлов», 101)*(истина)

«Соколов»<>«Орлов» ← активный запрос

Проверка запроса показывает, что последняя подцель является истиной, так как значения истины приняли ранее и первые

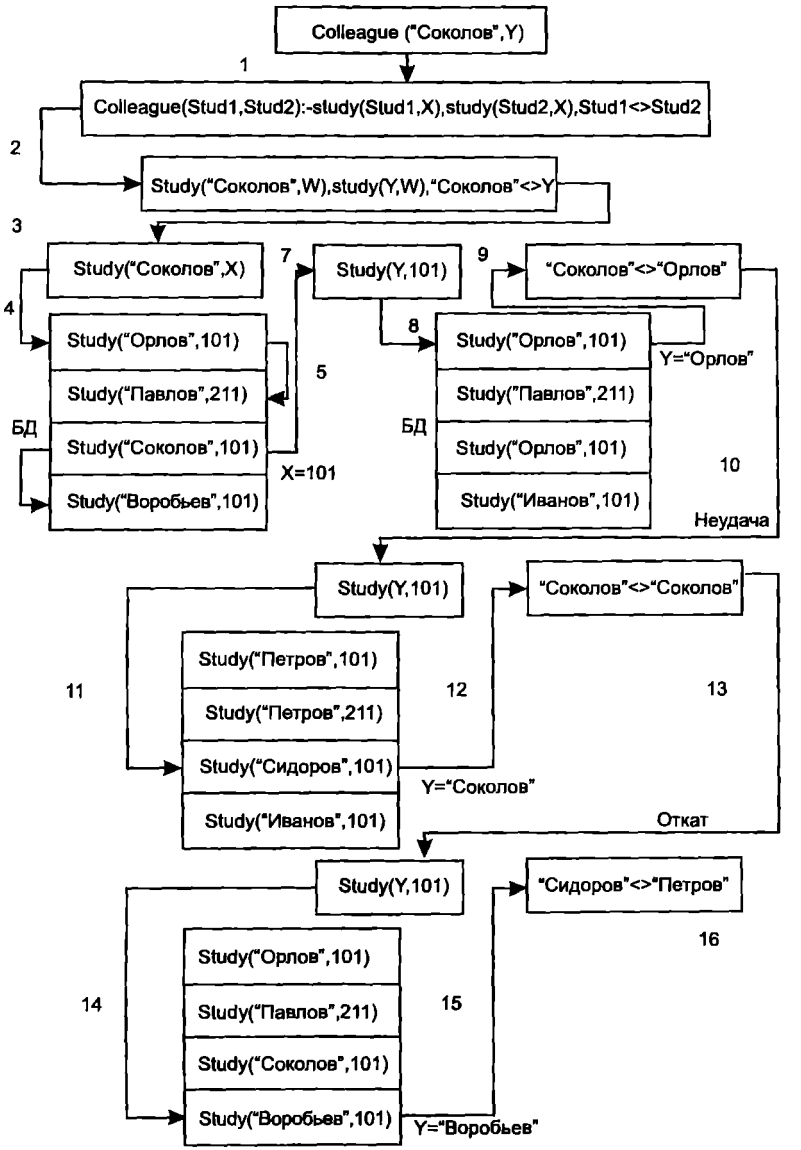


Рис. 2. Схема работы Пролог-системы при выводе ответа на запрос

две подцели, то и вся цель является истинной при значении переменной Y=«Орлов», о чем выдается сообщение на дисплей, то есть задача решена.

Но так как цель является внешней, то после поиска первого решения и вывода его на экран система искусственно генерирует состояние неудачи и делает откат к повторному доказательству предыдущей подцели с освобождением переменных. При этом новое состояние стека запросов будет иметь вид:

colleague («Соколов», Y)
study («Соколов», 101)*(истина)
study (Y, 101) ← активный запрос

Повторяется процесс, аналогичный п. 6. Существенное отличие в том, что поиск осуществляется не сначала, а с той позиции, которая отмечена маркером возврата, что сокращает пространство поиска.

После согласования активного запроса с базой и унификации переменной Y и константой «Соколов» система помечает эту позицию базы маркером возврата, и стек запросов принимает вид:

colleague («Соколов», «Соколов»)
study («Соколов», 101)*(истина)
study («Соколов», 101)*(истина)
«Соколов»<>«Соколов» ← активный запрос

Сопоставление в активном запросе дает ложное значение, что приводит к тому, что и весь запрос является ложным. Пролог-система автоматически пытается его передоказать, вызывая ситуацию отката (возврата) к последней успешной подцели, освобождая конкретизированные в последнем запросе переменные. Стек запросов аналогичен п. 9.

Поиск в базе, начиная с позиции, помеченной маркером возврата, согласование с фактом, унификация переменной Y с новым значением и переход к третьей подцели приведут к тому, что состояние стека запросов будет иметь вид:

colleague («Соколов», «Воробьев»)
work («Соколов», 101)*(истина)
work («Воробьев», 101)*(истина)
«Соколов»<>»Воробьев» ← активный запрос

Проверка в активном запросе показывает, что последняя подцель является истинной, а так как значения истины приняли ранее и первые две подцели, то и вся цель является истинной при значении переменной Y=«Воробьев», о чем выдается сообщение на дисплей, то есть задача решена. А так как маркер возврата стоит на конце базы данных, то у оболочки Турбо-Пролога отсутствует возможность искусственного вызова состояния неудачи, что приводит к окончанию задачи.

Задание 3

Выполните трассировку программы и составьте упрощенную схему работы Пролог-системы для случая поиска всех коллег.

Задание 4

Повторите задание 3 для случая выполнения запроса о поиске всех коллег Козлова all_colleague («Козлов», X, Y)

Откат или возврат автоматически инициируется системой Турбо-Пролога, если не используются специальные средства управления им. Для управления процессом отката в Прологе предусмотрены два предиката: fail (неудача) cut (отсечение).

Использование внешней цели побуждает переменные получать все возможные значения одно вслед за другим. При этом все их значения в неотформатированной форме выдаются в окне диалога системы. Если их число велико, то текст в окне будет быстро меняться и прочитать его будет довольно сложно и даже невозможно. Поэтому в этом случае встает задача обеспечения интерфейса вывода.

4. Использование отката после неудачи при использовании внутренней цели для организации простейшего интерфейса вывода

Если простейший интерфейс вывода будет реализован как внутренняя цель программы, то внутренние унификационные процессы Турбо-Пролога закончат поиск решения сразу после первого успешного вычисления целей. Для поиска всех значений Пролог-программа должна заставить повторно работать внутренние унификационные средства Турбо-Пролога.

Одним из способов реализации данной задачи является использование метода отката после неудачи (ОПН), использующего предикат fail.

Пример использования этого предиката демонстрирует программа 3_3.

```
/*программа 3_3*/
include «LAB2_5.PRO»
predicates
query
do_answer (name)
goal
query.
clauses
query:-
makewindow (2, 7, 15, «Запрос о студентах», 18, 0, 6, 50), cursor (1, 10),
write («Введите фамилию:»),
readln (Who),
makewindow (1, 7, 15, «Коллеги», 1, 50, 22, 29),
do_answer (Who).
do_answer (X):- colleague (X, Y), write (« », X, « », Y), nl, fail.
```

Два предиката этой программы позволяют формировать запрос и получать на него ответ. Текстовая подстановка файла программы 5 из лабораторной работы 2 обеспечивает доступ ко всем ее предикатам и базам данных.

Таким образом, эта программа являет собой пример интерфейса для ввода-вывода данных программы 2_5.

Программа содержит внутреннюю цель в виде предиката query, который создает на экране окно ввода данных, выдает подсказку, обеспечивает ввод и означивание переменной Who, а также формирует окно вывода данных.

Предикат do_answer обеспечивает запрос на поиск коллег введенного студента. Действия, выполняемые системой по данному запросу, аналогичны первым 11 шагам, приведенным на рис 2.

Однако на 11-м шаге система не будет осуществлять принудительный возврат и решение задачи не остановится.

Чтобы этого не случилось, в программу введен предикат fail, который всегда вызывает состояние неудачи, что заставляет систему после неудачи выполнять откат к предыдущей подцели и фактическое продолжение функционирования программы в режиме, аналогичном приведенному на рис. 2.

5. Содержание отчета по лабораторной работе

1. Результаты трассировки.
2. Результаты выполнения заданий 1, 2, 3, 4 данной лабораторной работы.
3. Результаты исследования программы 3_3 (сохранить как LAB3_3.PRO) при наличии и отсутствии предиката fail в программе.
4. Модифицированный вариант программы 3_3 (сохранить как LAB3_3.PRO) для случая, когда окно вывода занимает целый экран, а окно вывода присутствует на экране только на момент ввода.
5. Список, назначение и описание простейших предикатов Турбо-Пролога для ввода и вывода данных разного типа.

Лабораторная работа 4
Управление ходом выполнения Пролог-программ

Цель работы:

1. Ознакомление с действиями предикатов неудачи и отсечения.
2. Изучение методов организации повторного выполнения группы задач.
3. Знакомство со способом построения программ меню.
4. Управление ограничением пространства поиска с использованием отсечения.

1. Организация повторяющихся процессов

При выполнении запросов программа последовательно обращается к фактам и правилам. При этом правила часто требуют, чтобы задачи типа поиска элементов в базе, ввода и вывода данных выполнялись многократно. Однако в Прологе отсутствует возможность непосредственного задания итерационного процес-

са, то есть не реализованы синтаксические конструкции типа FOR, WHILE или REPEAT.

Существует два способа построения правил, выполняющих одну и ту же задачу несколько раз. Это повторение, использующее возврат, и рекурсия, использующая вызов процедуры самой себя.

Повторение реализуется с использованием метода отката после неудачи, когда осуществляется возврат к последней, имеющей альтернативное решение, подцели, реализация альтернативы и очередной возврат. Поэтому поиск с возвратом можно использовать для выполнения повторяющихся процессов. Примером построения правила, использующего повторение, является предикат do_answer ( ) в программе 3_3.

Но если этот подход применить к предикату query той же программы, то есть искусственно вызвать состояние неудачи, добавив предикат fail в виде последней подцели правила, то никакого повторного запроса на ввод мы не получим.

Задание 1

Вызовите программу 3_3, добавьте fail в конец описания query и, запустив программу на выполнение, убедитесь, что процесс не повторяется.

Объясняется это тем, что предикаты формирования окна и вывода не являются альтернативами, то есть не имеют альтернативных решений, а предикат do_answer ( ) все альтернативные решения получил, то есть цель query является истинной и решение задачи заканчивается.

Для того чтобы обеспечить многократный ввод данных, необходимо, чтобы откат выполнялся к некоторому предикату, имеющему альтернативное решение, с одной стороны, и чтобы это решение было постоянно истинным — с другой. При невыполнении последнего условия откат может произойти к еще более ранним подцелям и не обеспечит повторения нужной нам группы действий.

Используя простейшую рекурсию, процедура задания предиката для правила повтора, определяемого пользователем, может иметь вид:

```
Repeat.
Repeat:- repeat.
```

Первая строка — это факт, который всегда успешен и объявляет предикат repeat истинным. Однако так как есть для этого предиката еще одно правило, то указатель отката устанавливается на первый repeat. Вторая строка — это правило, которое использует самовызов. Правило (второй repeat) вызывает подцель (третий repeat), и этот вызов вычисляется успешно, так как факт (первый repeat) удовлетворяет подцели. Следовательно, правило также всегда успешно. Предикат repeat будет успешно вычисляться при каждой новой попытке его вызвать после отката. Таким образом, repeat — это рекурсивное правило, которое никогда не бывает неуспешным.

Задание 2

Введите в программу описание предиката repeat. Вставьте обращение к этому предикату в качестве первой подцели правила query. Запустите на выполнение программу. Теперь у вас появилась возможность повторного ввода, но нет признака окончания и единственный вариант выхода из программы — это Ctrl+Break.

Но для того чтобы сформировать признак окончания повторений, давайте разберемся, как работает программа и кто инициирует повтор. Первым выполняется repeat, который ничего не делает, далее — не имеющие альтернатив стандартные предикаты и последним — предикат do_answer, который и обеспечивает откат после неудачи к предикату repeat за счет присутствия в нем предиката fail.

Задание 3

Откомментируйте в правиле do_answer предикат fail и запустите программу на выполнение. Обратите внимание, что наличие в программе предиката repeat еще не обеспечивает повторов, что для организации повторов необходим возврат к предикату repeat. Снимите комментарий.

Но если откат к repeat вызывает do_answer, то он должен обеспечить, при определенных условиях, окончание этого процесса. Если за условие выхода будет принят, например, ввод слова «stop» вместо фамилии, то можно доопределить предикат do_answer еще одним правилом:

```
do_answer (X):- X=«stop»,
```

write («good bye»)

которое будет истинным при согласовании и которое, ввиду отсутствия признака неудачи, не вызовет отката к предикату repeat.

Задание 4

Добавьте новое правило в последнюю строку процедуры. Запустите программу на выполнение сначала в обычном режиме, а затем в режиме трассировки, но только предиката do_answer. Переставьте новое правило на первую строку процедуры и выполните трассировку. Какой получился результат? Что будет, если правило для условия выхода записать в виде:

do_answer («stop»):- write («good bye»)?

Обобщая изложенное, можно сделать вывод о том, что условие выхода из цикла может определяться любым предикатом, одно из множества альтернативных описаний которого должно содержать предикат fail или вызывать поиск с возвратом, то есть обеспечивать откат.

Из анализа даже простейших случаев организации повторяющихся процессов, как реализуемых самой Пролог-системой, так и определяемых пользователем, можно сделать вывод о необходимости управления этими процессами со стороны пользователя.

2. Управление поиском с возвратом

Один из вариантов управления поиском с возвратом можно иллюстрировать на примере предиката do_answer, записанного в несколько другой форме.

do_answer (X):- colleague (Z, Y), X=Z, write (« », X, «→», Y), nl, fail.

Альтернативная форма записи не изменяет сути данного правила, но дает возможность показать, что введением новых подцелей в правило можно управлять поиском с возвратом. Из этого примера видно, что вторая подцель может оказаться неуспешной из-за несоответствия служащего, унифицированного по первой подцели, со служащим, унифицированным через заголовок правила, то есть введенного с клавиатуры. Неуспех унификации второй подцели приведет к тому, что откат возникнет до выдачи информации на экран и предикат fail не потребуется. Включен

предикат fail в правило, чтобы вызвать откат, если условия правила будут выполнены и все правило окажется успешным.

```
Show_menu:- makewindow (1, 7, 15, «Меню», 1, 50, 10, 20),
repeat, clearwindow, write («1 — процесс 1»), nl, write («2 — процесс 2»), nl, write («0 — выход»), nl, nl, write («Ваш выбор →»),
readint (Menu), Menu<3.
process (Menu):-
stop_menu (Menu).
stop_menu (0).
stop_menu (_):- fail.
```

Еще одним примером по управлению поиском с возвратом является процедура построения меню show_menu.

В ней repeat используется так, что после выхода из любого модуля, вызываемого предикатом process ( ), идет возврат в меню.

Исключением является выбор нуля, что вызывает окончание программы.

В данном правиле управление откатом используется дважды: в виде предикатов Menu<3 и stop_menu (Menu).

Подцель Menu<3 проверяет значение, введенное с клавиатуры. Если введено значение большее или равное трем, то выполнение подцели закончится неуспехом и произойдет откат. Если введено значение меньшее, чем 3, то подцель закончится успехом и будет сделана попытка выполнения следующей подцели process ( ).

Если процедура process ( ) завершится успехом, то система делает попытку выполнить процедуру stop_menu ( ), которая при любых, кроме 0, значениях выбора завершается неудачей, что вызывает откат к предикату repeat. При значении выбора, равном 0, она является истинным фактом, и программа завершается.

Задание 5

Сохраните на диске ваш рабочий файл с именем LAB4_1.PRO. Он нам скоро понадобится. Создайте новый файл LAB4_2.PRO, в котором, используя приведенные выше предикаты, напишите программу организации выполнения двух произвольных процессов с выбором их через меню. Процессы должны быть описаны соответствующими предикатами и иметь простейший вид. Например, открытие окна и выдача сообщения.

Запустите программу на выполнение. Затем, используя трассировку, внимательно изучите ход решения задачи, откаты и повторы, выполняемые программой.

3. Управление ходом выполнения программ с использованием отсечения

Турбо-Пролог содержит средство, препятствующее поиску с возвратом в определенных условиях. Эта операция называется отсечением и выполняется предикатом cat, который в программах обозначается восклицательным знаком (!). Воздействие этого средства просто сводится к прекращению поиска. Отсечение используется в двух случаях:

1. Для ограничения пространства поиска в случаях, когда заранее известно, что некоторые возможные пути не приведут к интересующим вас решениям, то есть обработка их приведет к ненужной потере времени. С использованием отсечения программа решается быстрее и требует меньшего объема памяти.

2. Когда отсечение требуется по логике программы для:

— недопущения возврата к предыдущей подцели правила при откате.

Пусть какое-либо правило имеет вид:

R1 (X, Y, Z) if a (X), b (Y), !, c (X, Y, Z).

Правило, записанное в такой форме, указывает на то, что система пройдет через предикат cat только в том случае, если и подцель a (X), и подцель b (Y) будут успешными. После того как предикат cut будет обработан, система не сможет вернуться назад для повторного рассмотрения подцелей a (X) и b (Y), если подцель c (X, Y, Z) потерпит неудачу при текущих значениях переменных X, Y и Z.

— предотвращения перехода к следующему предложению процедуры.

Пусть процедура описания предиката r состоит из трех правил. Обозначим через r1, r2 и r3 записи одного и того же предиката r в каждом из трех предложений процедуры. Тогда два варианта записи этой процедуры в виде:

а) r1 (X, Y) if 1, a (X), b (Y), c (X, Y).

r2 (X, Y) if !, d (X, Y)
r3 (X, Y) if e (X, Y)
б) r1 (X, Y) if a (X), b (Y), c (X, Y), !.
r2 (X, Y) if d (X, Y), !.
r3 (X, Y) if e (X, Y)

соответствуют тому, что в первом случае при обработке предиката r будет использовано лишь одно из правил r1, r2, r3, а во втором случае истинность какого-либо одного из правил приводит к окончанию процедуры и исключению из рассмотрения всех записанных ниже.

Пример

Процедуру поиска максимального из двух чисел можно записать в виде двух правил предиката max. Но эти правила взаимоисключающие. Если неудачу терпит первое, то второе будет выполняться. Поэтому с использованием отсечения возможна значительно более короткая формулировка процедуры.

Max (X, Y, X):- X>=Y
Max (X, Y, Y):- X<Y

Max (X, Y, X):- X>=Y, !
Max (X, Y, X)

Таким образом, если предикат fail инициирует бектрекинг (возврат к перебору очередных альтернатив после первой найденной), то предикат cut его завершает.

Рассмотрим на простых примерах использование различных вариантов отсечения для управления ходом выполнения Пролог-программ.

4. Использование метода отката и отсечения

Использование этого метода начнем с рассмотрения задачи о коллегах, когда возникают требования по выдаче ответов на более широкий круг вопросов, чем определение коллег, только одного конкретного коллеги, задаваемого с клавиатуры (как это было в предыдущем примере). Предположим, что перечень возможных для обращения к системе запросов включает требования:

— по поиску всех коллег;

— поиску всех коллег конкретного коллеги;

— выяснению наличия коллег у конкретного коллеги;

— выдаче списка всех коллег, ограниченного снизу некоторым коллегой.

Один из вариантов реализации требований по запросам к системе может иметь вид расширенной процедуры do_answer ( )

```
/*правило 1*/
do_answer («stop»):- write («good bye»).
/*правило 2*/
do_answer (X):- X=«all», colleague (Z, Y), write (« », Z, «→», Y), nl, fail, I.
/*правило 3*/
do_answer (X):- frontchar (X, «I», Z), colleague (Z, Y), I, write (Z, «имеет коллег»), nl, fail.
/*правило 4*/
do_answer (X):- frontchar (X, «<», Z), colleague (Q, Y), write (« », «Q», «→», Y), nl, Q=Z, 1, fail.
/*правило 5*/
do_answer (X):- colleague (X, Y), write (« », «X», «→», Y), nl, fail.
```

Процедура do_answer ( ) состоит из пяти правил, два из них, первое и пятое, уже были рассмотрены ранее.

Задание 6

Загрузите с диска ваш файл Lab4_1.PRO и дополните процедуру do_answer ( ) недостающими правилами. Изучите их назначение, способ организации и исполнения системой Турбо-Пролог.

Правило 1

Обеспечивает прекращение повторов запросов за счет того, что его истинность при согласовании не вызывает отката к предикату repeat.

Правило 5

Реализует второе требование на запросы к системе и выдаче всех коллег конкретного коллеги. Рассмотрим подробнее остальные правила.

Правило 2

Обеспечивает поиск и выдачу всех коллег при условии, что в ответ на запрос о фамилии вводится слово all (то есть все). Если

с клавиатуры введено любое другое значение, первая подцель этого правила не согласуется и выполняется откат, который приводит к дальнейшему поиску согласующих предложений среди заголовков правил процедуры do_answer ( ).

Если первая подцель согласуется, то выполнение второй подцели обеспечит поиск первой пары коллег, а третья подцель выводит их на экран. Предикат fail, вызывая неудачу, обеспечивает откат ко второй подцели для поиска всех альтернативных решений, которые затем выдаются на экран.

Разработчику программы заранее известно, что для обработки слова all никаких других правил в процедуре do_answer ( ) не предусмотрено, но это неизвестно системе, которая будет просматривать все правила процедуры и проводить их унификацию.

В этом случае имеет смысл ограничить время и пространство поиска, закончив выполнение всей процедуры do_answer после обработки слова all. С этой целью в данное правило последним введен предикат отсечения, который приводит к прекращению процедуры do_answer, то есть исключению из дальнейшего согласования правил 3, 4 и 5.

Правило 3

Реализует третье требование к запросам при условии, что в ответ на приглашение к вводу фамилия вводится начиная с восклицательного знака. Выбор начального символа абсолютно произволен. Здесь этот символ выбран, как еще одно напоминание об отсечении.

Пусть, например, с клавиатуры введено «!Иванов». При согласовании заголовка правила 3 переменная X унифицируется со строковой константой «!Иванов» и начинается обработка тела правила.

Первая подцель вызывает предикат frontchar (X, !, Z) обработки символьных строк. Если первый символ X совпадает с «!», то переменная Z принимает значение остатка строки X, начиная со второго символа (Z=«Иванов»). Если первый символ в X отличный от «!», то первая подцель заканчивается неудачей и происходит откат к обработке следующего правила процедуры.

Вторая подцель будет успешно решена, если согласуется предикат colleague (Z, Y).

При каком-либо значении Z использование анонимной переменной обусловлено характером запроса, в котором требуется определить только наличие коллег, а не их фамилии. Успешное согласование второй подцели вызовет переход к предикату отсечения, который установит признак запрета на откат в этом месте правила.

После этого согласование правила продолжается в порядке слева направо, и осуществляется вывод на экран дисплея сообщения «Иванов имеет сослуживцев».

Последняя подцель вызывает состояние неуспеха в доказательстве правила, что ведет к откату, но так как предикат write ( ) альтернатив не имеет, то происходит откат к предикату отсечения, которым уже установлен запрет на дальнейший откат.

Это приводит к тому, что, в отличие от предыдущих примеров, повторное согласование предиката colleague ( ) выполняться не будет.

На этом завершается согласование данного правила, но не только его, но и всей процедуры do_answer ( ). Причем завершение процедуры неудачей ведет к тому, что неудачей завершится и последняя подцель в query и, как следствие, произойдет откат к предикату repeat и повтору ввода данных.

Задание 7

Что будет, если из этого правила исключить предикат fail? Попробуйте это сделать. Изучите процесс выполнения программы в этом случае. Восстановите программу. В отчете приведите описание процесса выполнения программы.

Правило 4

Реализует требование на ограничение выводимого списка некоторым условием. В качестве условия выступает фамилия коллеги, которым должен оканчиваться список коллег. При этом фамилия этого коллеги вводится начиная с символа «<». Выбор этого символа произволен и никакой смысловой нагрузки, с точки зрения языка Пролог, не имеет.

Первая подцель даст истинный результат, если первым символом X будет «<». Переменная Z в этом случае примет значение остатка строки X начиная со второго символа. В иных случаях

подцель закончится неудачей, вызывая неудачу и всего правила. Фактически эта подцель является условием входа на обработку правила.

Следующие две подцели согласуют и выводят на экран первую пару коллег, а четвертая — обеспечивает возврат к поиску следующей пары, если фамилия первого из коллег не совпадает с введенной с клавиатуры. Как только эти фамилии совпадут, то есть Q=Z станет истиной, выполнится отсечение, которое не позволит предикату fail обеспечить откат к поиску новой пары коллег, что вызовет окончание выполнения данного правила.

Наличие в правиле предиката отсечения не только ограничивает область поиска в базе данных заданной фамилией, но и обеспечивает завершение всей процедуры do_answer ( ). Завершение процедуры приводит к возврату в query.

Следует особо обратить внимание на то, что откат к поиску альтернативных решений подцели colleague (Q, Y) в данном правиле, в отличие от правил 2 или 5, выполняет не fail, а подцель Q=Z. Тогда встает вопрос, а нужен ли в этом правиле вообще предикат fail, и если нужен, то зачем?

Все заключается в том, что процедура do_answer ( ) выполняется не сама по себе, а вызывается как последняя цель правила query. Поэтому, кроме функций по поиску данных в базе, на нее возложена функция управления откатом к предикату repeat, то есть организация повторений по вводу.

Присутствие в правиле 4 предиката fail приводит к тому, что, несмотря на успешно выполненный запрос к базе данных, правило заканчивается неудачей. Неудачей завершается и запрос из query к процедуре do_answer ( ), что вызывает откат к repeat и повторение ввода данных.

Если в правиле 4 не будет предиката fail, то после выполнения поиска в базе правило будет успешно выполненным. Успешным будет и обращение из query к процедуре do_answer ( ), что ведет к тому, что и правило query станет истинным. Это вызовет его окончание, а вместе с этим и окончание ввода запросов.

Но так как признаком окончания ввода должен быть ввод слова stop, окончание работы программы по другому требованию

явилось бы ошибкой, что и вызывает необходимость обязательного включения предиката fail в правило 4.

Задание 8

Выполните программу в пошаговом режиме для правила 4. Изучите процесс выполнения отсечения. Повторите то же самое без fail. В отчете приведите схему выполнения программы для этих случаев.

Рассмотренные выше подходы к организации откатов и отсечений могут аналогичным способом использоваться и при формировании внешних целей в виде составных запросов.

Задание 9

Сформируйте внешние цели для запросов, аналогичных реализованным в процедуре do_answer ( ), и выполните их.

5. Откат и отсечение при реализации отношений вида «один-ко-многим»

Учет ассоциаций (связей) между объектами отношений служит средством оптимизации запросов в Прологе. Рассмотрим это на примере простого отношения

ОТЕЦ (ИМЯ, РЕБЕНОК),

которое в программе определяется набором некоторых фактов. Между объектами данного отношения существует ассоциация «один-ко-многим». Запрос к данной базе может заключаться:

— либо в поиске родителя конкретного ребенка (в этом случае имеет место простая связь);

— либо в поиске всех детей конкретного родителя (в этом случае имеет место множественная связь).

Самый примитивный способ реализации простой связи заключается в написании правила, которое должно выполнить поиск факта, а затем пройти через предикат отсечения. Для реализации сложной связи необходим поиск всех альтернативных решений в базе данных. Использование новой процедуры parent (родитель), которая учитывает введенные замечания, обеспечит интерфейс работы с БД father ( ).

Причем этот интерфейс учитывает имеющуюся связь между объектами отношения БД father ( ) и оптимизирует выполнение запроса Прологом.

Предикат bound (C) успешен в случае, если переменная C однозначна, а предикат free (C) успешен, если переменная C свободна.

Следует отметить, что два правила процедуры являются взаимоисключающими.

А если это так, то проверку во втором правиле состояния переменной C можно не выполнять, так как если она будет несвободной, то будет обработана первым правилом. Однако если эти правила поменять местами, то этого делать нельзя.

```
/*программа 4_3*/
domains
name, child=symbol
predicates
father (name, child)
parent (name, child)
clauses
father («Иван», «Петр»).
father («Иван», «Павел»).
father («Петр», «Олег»).
father («Олег», «Борис»).
/*Поиск родителя (отца) конкретного ребенка*/
parent (F, C):- bound (C), father (F, C), I.
/*поиск всех детей конкретного родителя*/
parent (F, C):- free (C), father (F, C).
```

Задание 10

Модифицируйте программу 2_5 таким образом, чтобы интерфейс по работе с БД study ( ) учитывал ассоциации между объектами этого отношения и позволял оптимизировать выполнение запросов Прологом за счет использования нового отношения

КОЛЛЕГА (ИМЯ, ГРУППА).

С целью тестирования программы выполните различные варианты запросов, используя внешние цели. Результаты их выполнения вместе с текстом модифицированной программы приведите в отчете по лабораторной работе.

Используя режим пошагового выполнения, выясните, как сказывается отсечение на режиме выполнения программы и поиске по базам данных.

Задание 11

Используя результаты выполнения задания 10, на основе программы формирования меню LAB4_2.PRO сформируйте запросную систему, которая должна выдавать ответы на вопросы: «В какой группе учится конкретный студент?» (процесс 1) и «Кто учится в конкретной группе?» (процесс 2).

6. Ступенчатые функции и отсечение

Часто в экономических расчетах приходится использовать ступенчатые функции для вычисления различных коэффициентов, зависящих от диапазонов изменения объемов производства, прибыли или совокупного годового дохода. Для примера рассмотрим двухступенчатую функцию, аналогичную вычислению процента подоходного налога, считая, что D — совокупный доход, а N — величина налога.

На Прологе данную функцию выражают с помощью бинарного отношения F (N, D), которое можно определить набором фактов вида

```
F (D, 0):- D, 10.
F (D, 12):- 10=<D, D<15.
F (D, 15):- 15=<D.
```

Три правила, входящие в отношение F ( ), являются взаимоисключающими, поэтому успех возможен самое большое в одном из них. Следовательно, мы (но не Пролог) знаем, что как только успех наступил в одном из них, нет смысла проверять иные, поскольку они все равно обречены на неудачу. Для предотвращения ненужного перебора следует воспользоваться отсечением. Предикат отсечения предотвращает, возвратив из тех точек программ, где он поставлен. С учетом этого новая процедура

```
F (D, 0):- D<10, I.
F (D, 12):- D<15, I.
F (D, 15)
```

дает тот же результат, что и исходная, но она значительно более эффективна при реализации в программе на Прологе.

7. Содержание отчета по лабораторной работе

1. Результаты выполнения задания 4 данной лабораторной работы.

2. В соответствии с заданием 5 текст отлаженной программы формирования меню, которая в ответ на выбор опции меню открывает некоторое окно, а после нажатия клавиши возвращается в основное меню, закрывая окно.

3. Полный текст программы запросной системы по коллегам и результаты ее исследований в соответствии с заданиями 6, 7, 8.

4. Список внешних целей, аналогичных реализованным в запросной системе, и результаты их выполнения (задание 9).

5. Результаты выполнения задания 10.

6. Текст отлаженной программы запросной системы по коллегам и группам с использованием меню и ее описание.

7. В качестве дополнительного задания — программа полной запросной системы.

VIII. ВВЕДЕНИЕ В ЯЗЫК ИСКУССТВЕННОГО ИНТЕЛЛЕКТА ЛИСП

Язык ЛИСП — один из наиболее распространенных базовых языков искусственного интеллекта. После появления ЛИСПа различными авторами был предложен ряд других алгоритмических языков, ориентированных на решение задач в области искусственного интеллекта, среди которых можно отметить Плэнер, Снобол, Рефал, Пролог, Смолтолк. Однако это не помешало ЛИСПу остаться наиболее популярным языком для решения таких задач. Более того, предполагается, что ЛИСП наряду с Прологом будет одним из основных языков компьютеров пятого поколения. Язык ЛИСП имеет множество диалектов, однако многие специалисты склонны отдавать предпочтение диалекту Common Lisp в качестве будущего стандарта языка ЛИСП.

Следует отметить, что наибольшую популярность ЛИСП получил в США. Популярность ЛИСПа объясняется следующими причинами:

1. ЛИСП ориентирован на работу с символьной информацией, а процесс решения большинства задач искусственного интеллекта сводится к обработке такой информации. (Английское название LISP, являющееся аббревиатурой выражения LISt Processing (обработка списков), хорошо подчеркивает основную область его применения.)

2. ЛИСП представляет собой интерпретирующую систему, а это позволяет значительно облегчить и ускорить процесс создания сложных программных комплексов в интерактивном режиме.

3. Идеология ЛИСПа крайне проста: данные и программы представляются в нем в одной и той же форме. Благодаря такой

унификации представления данные могут интерпретироваться как программа, а любая программа может быть использована как данные любой другой программой.

4. Язык ЛИСП является языком функционального программирования. Применение функциональных языков открывает широкие перспективы, позволяя пользователю описывать скорее природу своих задач, чем способ их решения.

Язык ЛИСП может служить основой для обучения методам искусственного интеллекта, исследованиям и практическому применению в этой области.

Данные методические рекомендации предназначены для выполнения лабораторных работ по программированию на языке ЛИСП, причем тематика работ ориентирована на приложения в области искусственного интеллекта.

В предлагаемом учебно-методическом пособии 3 лабораторные работы. В зависимости от количества отведенных часов и степени подготовленности студентов лабораторные работы могут быть рассчитаны на два или четыре часа. В большинстве лабораторных работ варианты заданий приводятся в порядке возрастания их сложности и должны выдаваться дифференцированно в зависимости от уровня подготовленности студента. Задания на следующую лабораторную работу получают на текущей лабораторной работе, с тем чтобы студент мог дома изучить необходимые функции, методы и алгоритмы и написать черновой вариант ЛИСП-программы.

Порядок выполнения лабораторной работы включает: получение у преподавателя задания, составление, тестирование и отладку ЛИСП-программы, оформление отчета. По каждой лабораторной работе составляется отчет, который должен содержать: название и цель работы, задание, программу на языке ЛИСП и ее описание, тестовые примеры и результаты выполнения программы, выводы.

Лабораторная работа 1
Работа со списками

Цель работы:

Изучение функций для работы со списками, изучение методов обработки списков с использованием рекурсии, приобретение навыков написания функций для работы со списками.

Постановка задачи

Списки являются основными типами данных языка ЛИСП. Под списком понимается конечная последовательность элементов списка, заключенная в круглые скобки. Элементом списка может быть либо атом, либо список. Первый элемент списка называется головой списка, остаток списка (без первого элемента) — хвостом списка. Атомом называется произвольная последовательность букв и цифр, заключенная между двумя ограничителями языка ЛИСП. Литеральный атом часто называется символом. В языке ЛИСП определены два стандартных атома T и NIL, играющие роль понятий ИСТИНА и ЛОЖЬ. Строго говоря, числа в ЛИСПе также являются атомами.

Пример списка: (A (B (C))). Данный список состоит из двух элементов: атома A и списка (B (C)), который в свою очередь состоит из атома A и списка (C).

Пустой список — это список, не содержащий ни одного элемента. Он обозначается ( ) или атомом NIL. Пустой список в ЛИСПе относится к атомам.

Список представляет собой ссылочную структуру. Основная ссылочная структура языка — так называемая точечная пара, которая состоит из указателей на первый и второй элементы пары. Так, например, точечная пара с указателями на атомы A и B изображается в символьной нотации как (A. B). Если точечная пара первым указателем ссылается на атом A, а вторым — на другую точечную пару (B. C), то это изображается в символьной нотации как (A. (B. C)).

Список (A B C D) в точечной нотации будет иметь следующий вид:

(A. (B. (C. (D. NIL)))).

Под S-выражением в языке ЛИСП понимается либо атом, либо список. В языке ЛИСП с точки зрения синтаксиса программы и данные не различаются. Любое S-выражение ЛИСПа можно интерпретировать как программу. Интерпретацию S-выражений выполняет функция EVAL. Однако не любое S-выражение может быть удачно интерпретировано из-за семантических соображений. Можно выделить некоторые общие правила интерпретации S-выражений:

1) если S-выражение является числом или атомом T или NIL, то EVAL возвращает это S-выражение без изменений;

2) если S-выражение является литеральным атомом, то функция EVAL возвращает последнее значение, которое было присвоено этому атому, в противном случае возвращается сообщение об ошибке;

3) если S-выражение представляет собой список вида (f arg1 arg2 ... argN), то функция EVAL пытается интерпретировать его следующим образом: первый элемент списка интерпретируется как имя функции, которую необходимо выполнить, взяв в качестве аргументов оставшиеся элементы списка arg1, arg2, ..., argN. В случае удачи функция EVAL возвращает S-выражение, которое является результатом выполнения функции f. Функция f может быть встроенной или определенной пользователем.

В некоторых случаях не надо вычислять значение выражения, а нужно само выражение. Чтобы предотвратить вычисление значения выражения, нужно перед этим выражением поставить апостроф ‘. Апостроф перед выражением — это на самом деле сокращение ЛИСПовской формы (QUOTE), где QUOTE — специальная функция, которая возвращает невычисленный аргумент.

```
> ‘ (+ 2 3)
(+ 2 3)
> (+ 2 3)
5
```

Здесь и ниже в примерах знак «>» означает приглашение интерпретатора Xlisp. После данного знака набирается интерпретируемое выражение. Ответ интерпретатора представляется на второй строке.

Базисными функциями обработки S-выражений являются: CAR, CDR, CONS, ATOM и EQ.

Вызов функции CAR: (CAR). Функция CAR возвращает в качестве значения первый (головной) элемент списка-аргумента. Тип результата — S-выражение.

```
> (CAR " ((1 2) 3))
(1 2)
```

Вызов функции CDR: (CDR). Функция CDR возвращает в качестве значения хвостовую часть списка-аргумента. Тип результата — список.

```
> (CDR " ((1 2) 3))
(3)
> (CDR " A)
NIL
```

Функции CAR и CDR позволяют добраться до любого элемента любого списка. Например, до второго элемента списка можно добраться с использованием следующего S-выражения: (CAR (CDR)).

```
> (CAR (CDR " (1 2 3 4)))
2
```

Для композиции функций CAR и CDR введены специальные обозначения. В Xlisp это — CxxR, CxxxR, CxxxxR, где x — это буква A или D. Например, вызов (CADR X) эквивалентен вызову (CAR (CDR X)).

```
> (CADR " (1 2 3 4))
2
```

Функция CONS (construct) строит новый список из переданных ей в качестве аргументов головы (S-выражение) и хвоста (списка):

```
(CONS).
> (CONS "A " (B C))
(A B C)
> (CONS " (A B) " (C D))
((A B) C D)
```

Предикат ATOM проверяет, является ли аргумент атомом. Аргументом может быть любое S-выражение. Значение предиката

равно T, если значение аргумента — атом, и NIL, когда значение аргумента — не атом.

> (ATOM " X)

T

Предикат EQ проверяет тождественность двух символов. Предикат EQ принимает значение T, если символы идентичны, и значение NIL во всех других случаях, включая случаи, когда аргументами являются не символы.

> (EQ (CAR " (A A)) (CAR " (A B C D)))

T

Более общим по сравнению с EQ является предикат EQL, который дополнительно позволяет сравнивать однотипные числа и строки. Обобщением EQL является предикат EQUAL, позволяющий проверять одинаковость двух списков:

> (EQUAL " (X Y Z) " (X Y Z))

T

Основными функциями, изменяющими физическую структуру списков, являются RPLACA (replace CAR) и RPLACD (replace CDR):

(RPLACA)

(RPLACD)

Функция RPLACA заменяет первый элемент списка, а функция RPLACD — остаток списка значением выражения. Обе функции возвращают список после его изменения.

Условные выражения в ЛИСПе обычно строятся с использованием функции COND. Структура условного выражения такова:

(COND (p1 e1) (p2 e2)... (pN eN)),

где p1, p2, ..., pN — предикаты, а e1, e2, ..., eN — произвольные результирующие выражения.

Значение функции COND определяется следующим образом:

1. Вычисляются последовательно слева направо значения выражений pi до тех пор, пока не встретится выражение, значение которого отлично от NIL, что интерпретируется как ИСТИНА.

2. Вычисляется результирующее выражение, соответствующее этому предикату, и полученное значение возвращается в качестве значения функции COND.

3. Если истинного предиката нет, то значением COND будет NIL. В условном выражении может отсутствовать результирующее выражение pi или на его месте часто может быть последовательность выражений: (COND (p1 e11)... (pi)... (pk ek2... ekn)...).

Если условию не ставится в соответствие результирующее выражение, то в качестве результата предложения COND при истинности предиката выдается само значение предиката. Если же условию соответствует несколько выражений, то при его истинности выражения вычисляются последовательно слева направо и результатом функции COND будет значение последнего выражения последовательности.

Символы в языке ЛИСП могут иметь значения. Изначально у символов нет какого-либо значения. Значения символам присваиваются при помощи псевдофункций (иначе, функций с побочным эффектом, причем основным действием и является этот побочный эффект) SET, SETQ, SETF.

Вызов функции SETQ имеет следующий вид: (SETQ), где символ (не вычисляется) — выражение, описывающее новое значение символа. Функция возвращает значение символа.

Функция SET аналогична функции SETQ, но, в отличие от последней, ее аргумент не вычисляется.

```
> (SETQ A " B)
B
> (SET A 7)
7
> B
7
```

Наряду со значением любой символ в языке ЛИСП может иметь специальный присоединенный список, называемый списком свойств. Структура этого списка:

(p1 v1 p2 v2 ... pn vn),

где p1, p2, ..., pn — атомы, обозначающие имена свойств (называемые иногда индикаторами), v1, v2, ..., vn — S-выражения, соответствующие значениям этих свойств. Выяснить значение свойства, связанного с символом, можно с помощью функции GET: (GET).

Присваивание нового свойства или изменение значения существующего свойства осуществляется псевдофункцией PUTPROP:

(PUTPROP).

Удаление свойства и его значения осуществляется псевдофункцией REMPROP: (REMPROP).

Ассоциативным списком (а-списком) в ЛИСПе называется список точечных пар вида: ((k1. v1) (k2. v2)... (kn. vn)). Первый элемент пары называется ключом, а второй — связанными с ключом данными. Функция ASSOC ищет пару, соответствующую ключу в а-списке:

(ASSOC). Поиск ведется от первой пары списка к последней. Функция ACONS добавляет новую пару в начало списка.

Определить новую функцию в языке ЛИСП можно с помощью функции DEFUN:

DEFUN (p1 v1 p2 v2 ... pn vn),

где DEFUN — имя определяемой функции, p1, p2, pn — атомы, используемые для указания формальных аргументов функции, v1, v2, vn — любые S-выражения, составляющие тело функции. Функция возвращает символ. Для определения неименованных функций в ЛИСПе используются лямбда-выражения.

Ввод в интерпретатор определений функций и их вызовов может осуществляться как с командной строки интерпретатора, так и из файла. Для последнего случая предусмотрена системная функция LOAD:

(LOAD).

Читаемые из файла выражения выполняются так, как будто они были введены пользователем.

В ЛИСПе под именем файла подразумевается путь к файлу, описанный средствами MS DOS, причем символ «/» в пути удваивается.

Функция LOAD возвращает T в случае, если файл удачно загружен, и NIL — в противном случае:

```
> (LOAD «c://xlisp//trace.lsp»).
T
```

Функция является рекурсивной, если в ее определении содержится вызов самой этой функции. Рекурсия является простой, если вызов функции встречается в некоторой ветви лишь один раз. Простой рекурсии в процедурном программировании соответствует обыкновенный цикл. Соответствующие вложенным циклам многократные повторения осуществляются обычно с помощью двух и более функций, каждая из которых соответствует простому циклу. В рекурсивном определении существенен порядок следования условий. Правильную последовательность можно получить путем логичного рассуждения на основе учета возможных ситуаций. При определении порядка следования условий основным моментом является то, что сначала проверяются все возможные условия окончания, а затем ситуации, требующие продолжения вычислений. Отсутствие проверки, ошибочное условие или неверный их порядок могут привести к бесконечной рекурсии.

В качестве примера использования рекурсии рассмотрим функции, реализующие пересечение двух множеств, представленных списками.

```
(defun belongs (elem list)
(cond ((null list) NIL)
((eq elem (car list)) T)
(T (belongs elem (cdr list)))))
(defun intersect (list1 list2)
(cond ((null list2) NIL)
((belongs (car list2) list1)
(cons (car list2) (intersect list1 (cdr list2))))
(T (intersect list1 (cdr list2)))))
```

Первая из приведенных функций (belongs) используется для определения, принадлежит ли элемент elem списку list. Функция возвращает T — если принадлежит, и NIL — если нет. Таким образом, данная функция является предикатом.

```
> (BELONGS " B " (A B C D))
T
```

Тело функции belongs состоит из условного предложения, имеющего три ветви:

1) Если список list пустой (определяется предикатом null), то возвращается значение NIL. Данный список может быть пустым либо с самого начала, либо потому что просмотр списка окончен.

2) Если первым элементом списка list является искомый элемент elem, то возвращается T.

3) Ни одно из предыдущих утверждений не верно. В таком случае либо элемент содержится в хвосте списка, либо вовсе не входит в список. Для продолжения решения задачи применяем тот же предикат belongs к хвосту списка. Таким образом, задача свелась к прежней, но на шаг более короткой.

Вторая функция (intersect) формирует список-результат из тех элементов, которые входят в список list1 и list2 одновременно.

Функция intersect состоит из условного предложения, имеющего три ветви:

1) Если список list2 пуст, то возвращается NIL.

2) Если первый элемент списка list2 принадлежит списку list1, то этот элемент включается как голова результирующего списка. Хвост данного списка формируется как результат пересечения списка list1 и хвоста списка list2.

3) Ни одно из предыдущих условий не верно. В этом случае результирующий список получается как результат пересечения списка list1 и хвоста списка list2.

```
> (INTERSECT " (A B C D) " (X D E C))
(C D)
```

Для отладки ЛИСП-программы следует использовать средства трассировки. Трассировка чего-либо в Xlisp включается при помощи функции TRACE: (TRACE ").

```
> (TRACE " INTERSECT)
TRACE
```

После ввода директивы TRACE интерпретатор будет распечатывать имя функции, значения аргументов каждого вызова трассируемой функции и полученный результат после окончания вычисления каждого вызова. Трассировку можно отключить аналогичной по форме директивой UNTRACE.

Варианты заданий на выполнение

Ниже приводятся спецификации функций, которые необходимо реализовать на языке ЛИСП. Вариант, помеченный символом [*], включает два подварианта:

подвариант а) — действия функции распространяются только на список верхнего уровня;

подвариант б) — рассматриваются подсписки всех уровней (замечание: во втором случае использовать рекурсию как по хвостам, так и по головам списков). Приведенные функции для удобства разбиты по темам.

Операции над списками

1) Функция COPY — создает в списочной памяти второй экземпляр произвольного S-выражения.

2) Функция LENGTH — возвращает в качестве значения длину списка [*].

3) Функция APPEND — соединяет два списка в один новый список.

4) Функция REMOVE — удаляет из списка все совпадающие с данным атомом элементы и возвращает в качестве значения список из всех оставшихся элементов [*].

5) Функция REMOVEF — удаляет из списка первые вхождения данного элемента [*].

6) Функция REMOVEL — удаляет из списка последний элемент [*].

7) Функция SUBSTITUE — заменяет все вхождения данного элемента в списке на новый элемент [*].

8) Функция REVERSE — изменяет порядок элементов в списке на обратный [*].

9) Функция FIRST-ATOM — результатом функции является первый атом списка (в учет принимаются списки всех уровней).

10) Функция LAST — возвращает последний элемент списка.

11) Функция ADDIFNONE — проверяет, содержится ли заданный элемент (значение первого аргумента) в заданном списке (значение второго аргумента), и если нет, то добавляет этот элемент к списку [*].

12) Функция COLLECT — перегруппирует элементы заданного списка так, чтобы одинаковые элементы, если они есть в списке, стояли подряд [*].

13) Функция FLATTEN — устраняет в произвольном S-выражении все внутренние скобки, а в точечных выражениях — и точки, превращая его в список атомов. Количество и относительный порядок атомов в выражении сохраняются.

14) Функция REVL — обращает список и разбивает его на уровни. Пример: исходный список — (a b c), результирующий список — (((c) b) a).

15) Функция DEVLEV1 — разбивает список на уровни. Пример: исходный список — (a b c), результирующий список — (a (b (c))).

16) Функция DEVLEV2 — разбивает список на уровни. Пример: исходный список — (a b c), результирующий список — (((a) b) c).

17) Функция DESTLEV1 — убирает уровни в списке. Пример: исходный список — (a (b (c))), результирующий список — (a b c).

18) Функция DESTLEV2 — убирает уровни в списке. Пример: исходный список — (((a) b) c), результирующий список — (a b c).

19) Функция REMSEC — удаляет из списка каждый второй элемент [*].

20) Функция DEVPAIR — разбивает список на пары. Пример: исходный список — (a b c d ...), результирующий список — ((a b) (c d)...).

21) Функция MIX — чередует элементы двух списков-аргументов и образует новый список. Пример: исходные списки — (a b ...) и (1 2 ...), результирующий список — (a 1 b 2 ...).

22) Функция DEPTH — вычисляет глубину списка (самой глубокой ветви).

Функции с побочным эффектом

1) Функция ATTACH — вырабатывает то же значение, что и CONS, но, в отличие от CONS, она заставляет обладать этим значением свой второй аргумент.

2) Функция DREVERSE — вырабатывает то же значение, что и REVERSE, но разрушает свой аргумент.

3) Функция NCONC — вырабатывает то же значение, что и APPEND, но одновременно она заставляет обладать этим значением свой первый аргумент.

4) Функция TCONC — помещает значение своего первого аргумента в конец очереди, представленной вторым аргументом. Если эта очередь пуста, то формируется очередь, состоящая из одного элемента.

Под очередью понимается списочная структура, состоящая из некоторого списка и ЛИСПовской ячейки, содержащей указатели на первый и последний элементы этого списка.

5) Функция EFFACE — значением второго аргумента функции должен быть список. Если этот список содержит хотя бы один элемент, совпадающий со значением первого аргумента, то первый по порядку из этих элементов исключается из списка, в противном случае список не меняется. Значением функции является преобразованный список. Если выброшенный элемент не был первым в списке, то значением второго аргумента также становится преобразованный список.

6) Функция DREMOVE — в отличие от EFFACE, выбрасывает из списка, являющегося значением ее второго аргумента, все элементы, совпадающие со значением первого аргумента функции.

7) Предикат LCYCLEP — вырабатывает значение T, если в значении аргумента есть цикл по цепочке d-указателей, и NIL в противном случае, то есть если в результате многократного применения функции CDR к значению аргумента можно получить атом.

8) Предикат CYCLEP — вырабатывает значение T, если значение аргумента — циклическая списочная структура, и значение NIL, если в значении аргумента нет циклов.

Предикаты

Предикаты FORALL, FORSOME, FORODD — у этих предикатов по два аргумента. Значением первого аргумента должен быть некоторый список L, а второго (функционального) аргумента — наименование или определяющее выражение функции P.

1) Предикат FORALL — принимает значение T лишь в том случае, если функция P принимает значение ИСТИНА (то есть не NIL) на всех элементах списка L.

2) Предикат FORSOME — принимает значение T, если функция P принимает значение ИСТИНА хотя бы на одном элементе списка L.

3) Предикат FORODD — принимает значение T, если число элементов списка L, на которых функция P принимает значение ИСТИНА, нечетно.

4) Предикат ATOMLIST — проверяет, является ли его аргумент списком (возможно, пустым), составленным лишь из атомов.

5) Предикат LISTP — принимает значение NIL, если заданное выражение является атомом, отличным от NIL, или выражением, которое может быть записано только в точечных обозначениях. В противном случае заданное выражение является списком, который можно записать, не прибегая к точечным обозначениям ни на одном из уровней, и предикат принимает значение T.

6) Предикат ONELEVEL — проверяет, является ли аргумент одноуровневым списком.

Порядок и упорядочение списков

1) Предикат ORDER — проверяет, в каком порядке два заданных элемента встречаются в данном списке. Список (упорядочивающая последовательность) задается в качестве значения третьего аргумента функции ORDER. Если при просмотре элементов этого списка слева направо встречается элемент, совпадающий со значением первого аргумента, а ни один из ранее просмотренных элементов не совпал со значением второго аргумента, то значение предиката равно T, во всех остальных случаях оно равно NIL.

2) Предикат ORDER1 — вычисляется так же, как и ORDER, однако если ни один из заданных элементов не содержится в данном списке, то в качестве значения предиката выдается атом ORDERUNDEF.

3) Предикат LEXORDER — сравнивает (элемент за элементом) два списка, заданные как значения первого и второго аргументов. Если в какой-либо позиции обнаруживаются различные элементы, то они сравниваются между собой с помощью пре-

диката ORDER1, причем в качестве третьего аргумента (упорядочивающей последовательности) указывается третий аргумент обращения к LEXORDER. Результат сравнения (T, NIL, ORDERUNDEF) выдается в качестве значения предиката LEXORDER. Если раньше, чем встретятся различные элементы, исчерпается первый список, то выдается результат T, если первым исчерпается второй список, то в качестве результата выдается NIL.

4) Предикат LEXORDER1 — отличается от LEXORDER тем, что значением третьего аргумента для него должен быть список, каждая позиция которого в свою очередь является списком, применяемым в случае необходимости для сравнения соответствующих позиций списков, заданных в качестве значений первых двух аргументов.

5) Функция FIRST — среди элементов списка, заданного в качестве значения первого аргумента, выбирает тот, который раньше встречается в списке, заданном в качестве значения второго аргумента. Если ни один из элементов первого списка не содержится во втором, то выбирается первый элемент первого списка.

6) Функция RANK — упорядочивает список, заданный в качестве ее первого аргумента, переставляя его элементы в той последовательности, в какой они встречаются в списке, являющемся значением второго аргумента.

Поиск в списках

В этом разделе собраны функции, выбирающие из списка элемент или элементы, обладающие заданным свойством. Список задается в виде значения аргумента, соответствующего связанной переменной L, свойство характеризуется предикатом, наименование или определяющее выражение которого дано в качестве значения аргумента, соответствующего связанной (функциональной) переменной P.

1) Функция POSSESSING — образует список из всех элементов данного списка, обладающих заданным свойством.

2) Функция SUCHTHAT — выбирает из заданного списка первый элемент, обладающий заданным свойством. Если такого элемента нет, то вырабатывается значение NIL.

3) Функция SUCHTHAT1 — проверяет, содержится ли в данном списке хотя бы один элемент с заданным свойством. Если да, то в момент обнаружения такого элемента в качестве результата принимается значение четвертого аргумента функции SUCHTHAT1. Если нет, то результатом является значение третьего аргумента.

4) Функция SUCHTHAT2 — проверяет, содержится ли в данном списке хотя бы один элемент с заданным свойством. Если да, то к хвосту заданного списка начиная с найденного элемента применяется функция, наименование или определяющее выражение которой задано в качестве значения третьего аргумента (функционального). Если нет, то вырабатывается значение NIL.

5) Функция FIRST-COIN — возвращает первый элемент, входящий в оба списка X и Y, в противном случае — NIL.

Операции над множествами

Списки рассматриваются как множества своих элементов, но порядку элементов в списке не придается значения, а два или более одинаковых элемента списка рассматриваются как один элемент множества.

1) Функция SETOF — для каждого повторяющегося элемента исключает из списка все вхождения, кроме одного.

2) Функция MAKESET — делает то же, что SETOF, но описана через PROG. Порядок элементов в результирующем списке оказывается другим.

3) Функция DIFLIST — вычисляет разность множеств X/Y. Иначе говоря, она исключает из списка, заданного в качестве значения первого аргумента функции, все элементы, встречающиеся в списке, представленном значением второго аргумента.

4) Функция SUBSET — вычисляет предикат «множество X является подмножеством множества Y». Иначе говоря, она вырабатывает значение T, если каждый элемент списка, заданного в качестве первого аргумента функции, содержится в списке, представленном значением второго аргумента.

5) Функция UNION — вычисляет объединение двух множеств. Значение функции представляет собой список всех выражений, являющихся элементами хотя бы одного из заданных списков. Если каждый из заданных списков не содержал повторяющихся

элементов, то в результирующий список каждый элемент войдет лишь один раз.

6) Функция LUNION — объединяет множества, заданные в качестве элементов списка, являющегося значением аргумента функции.

7) Функция INTERSECTION — вычисляет пересечение двух множеств. Значением функции является список всех выражений, входящих элементами в оба заданных списка. Если каждый из заданных списков не содержит повторяющихся элементов, то в результирующемся списке элементы не будут повторяться.

8) Предикат EQUALSET — проверяет, равны ли множества, представленные двумя заданными списками.

9) Функция CART — образует декартово произведение двух заданных множеств. Точнее говоря, она формирует лексикографически упорядоченный список, элементами которого являются всевозможные списки, содержащие по два элемента каждый, причем первый элемент берется из первого, а второй — из второго заданного списка.

10) Предикат SETP — проверяет, является ли список множеством, то есть входит ли каждый элемент в список лишь один раз.

11) Функция SIMDIFF — формирует множество из элементов, не входящих в оба множества (симметрическая разность множеств).

Ассоциативные списки

1) Функция PAIR — объединяет элементы двух заданных списков в ассоциативный список, вырабатываемый в качестве значения функции. Функция не определена, если второй из заданных списков короче первого.

2) Функция DPAIR — действует аналогично PAIR, но значение первого аргумента заменяется значением функции.

3) Функция ASSOC — вырабатывает в качестве значения первую по порядку пару из заданного ассоциативного списка, у которой первый элемент совпадает с заданным выражением. Если такой пары нет, то вырабатывается значение NIL.

4) Функция PAIRLIS — подобна функции PAIR и отличается от нее лишь тем, что она не создает ассоциативный список, а добавляет новые пары к существующему списку.

5) Функция SUBST — подставляет в заданное выражение z (в значение ее третьего аргумента) выражение x (значение первого аргумента) вместо всех подвыражений, совпадающих со значением y второго аргумента (на какой бы глубине они ни находились). Результат подстановки выдается в качестве значения функции.

6) Функция SUBLIS — в заданном выражении y (значение ее второго аргумента) заменяет все входящие в него атомы, которым в заданном ассоциативном списке (значение первого аргумента) поставлены в соответствия некоторые выражения, этими выражениями. Преобразованное выражение выдается в качестве значения функции. Функция SUBLIS обращается к функции SUB2, которая для заданного атома (значения второго аргумента) выдает в качестве значения либо выражение, поставленное ему в соответствие в заданном ассоциативном списке, либо сам этот атом, если такого выражения нет.

7) Функция SASSOC — подобна функции ASSOC и отличается от нее тем, что, когда в заданном ассоциативном списке не найдено никакого соответствия для заданного выражения, в качестве значения выдается результат обращения к функции (без аргументов), наименование или определяющее выражение которой задано в качестве третьего аргумента функции SASSOC.

Функционалы

1) Функция APL-APPLY — применяет каждую функцию fi списка f=(f1 f2 fN), являющегося первым аргументом функции, к соответствующему элементу xi списка x=(x1 x2 xN), являющегося вторым аргументом функции, и возвращает список, сформированный из результатов.

2) Функция MAPLIST — применяет функцию, заданную в качестве ее второго аргумента (функционального), последовательно ко всему списку, заданному в качестве значения первого аргумента, и ко всем спискам, поочередно получаемым отбрасыванием первого элемента от предыдущего списка. Значением функции MAPLIST является список полученных результатов.

3) Функция MAPCAR — подобна функции MAPLIST и отличается от нее тем, что заданная функция применяется к первым

элементам тех списков, к которым она применялась бы в случае обращения к MAPLIST.

4) Функция MAP — имеет смысл, если функция F, заданная в качестве ее второго аргумента, обладает каким-либо побочным эффектом. Эта функция последовательно применяется к тем же аргументам, что и в случае функции MAPLIST, но выработанные значения никак не используются и не сохраняются. Значение функции MAP равно NIL (а если очередной аргумент, подготовленный для обращения к функции F, окажется атомом, то этому атому).

Списки свойств

1) Функция PUTPROP — помещает в список свойств атома, указанного в качестве ее первого аргумента, свойство, представленное значением третьего аргумента, с индикатором, заданным в виде значения второго аргумента. В качестве результата функция PUTPROP вырабатывает атом, список свойств которого подвергся изменению. Если в списке свойств этого атома уже было свойство с данным индикатором, то это свойство замещается новым, а индикатор не дублируется.

2) Функция GETPROP — извлекает из списка свойств данного атома (значения первого аргумента) свойство с данным индикатором (значением второго аргумента). Если такого индикатора в списке свойств нет, то вырабатывается значение NIL.

3) Функция PROP — отличается от функции GETPROP тем, что она выдает в качестве результата не свойство, связанное с данным индикатором (если оно есть в списке), а весь хвост списка свойств, следующий за найденным индикатором. Если же индикатор в списке свойств не обнаруживается, то производится обращение к функции без аргументов, наименование или определяющее выражение которой должно быть задано в качестве третьего аргумента PROP. Результат этого обращения выдается в этом случае в качестве результата обращения к PROP.

4) Функция REMPROP — удаляет из списка свойств данного атома (значения первого аргумента в обращении к ней) свойство, снабженное данным индикатором (значение второго аргумента),

вместе с этим индикатором. Если такого индикатора не было, то список свойств не меняется. Значение функции совпадает со значением первого аргумента.

5) Функция REMPROPS — удаляет все свойства символа.

6) Предикат HASPROP — проверяет, обладает ли символ (первый аргумент) данным индикатором (второй аргумент).

Работа с упорядоченными бинарными деревьями

Упорядоченное бинарное дерево состоит из узлов вида:

( ).

В каждом узле выполнено следующее условие: все элементы из узлов как левого, так и правого поддерева в некотором упорядочении (например по числовой величине или в алфавитном порядке) предшествуют определенному элементу бинарного дерева. Пример:

(5 (3 (1 NIL NIL)
(4 NIL NIL)
(7 (6 NIL NIL)
(13 (11 NIL NIL)
(15 NIL NIL))))

1) Функция FINDBT — ищет в дереве данный элемент.

2) Функция ADDBT — добавляет в дерево данный элемент. (Замечание: копируйте дерево по пути поиска и исправляйте нужное поддерево.)

3) Функция PREDBT — выделяет в отдельное (упорядоченное) дерево из первоначального дерева все узлы, предшествующие данному элементу.

4) Функция POSTBT — выделяет в отдельное (упорядоченное) дерево из первоначального дерева все узлы, следующие за данным элементом.

5) Функция UNIONBT — объединяет два упорядоченных дерева в одно общее упорядоченное дерево. (Замечание: используйте функции PREDBT и POSTB.)

Разные процедуры формирования списков

1) Функция BULB (ЛУКОВИЦА) — строит N-уровневый вложенный список, элементом которого на самом глубоком уровне является N.

2) Функция FACT — результатом является выражение, являющееся факториалом числа, в котором числа (сомножители) упорядочены в порядке возрастания.

3) Функция FIBON — порождает последовательность чисел Фибоначчи. Числа Фибоначчи определяются следующим рекуррентным соотношением: F (n+1)=F (n)+F (n–1), при F (0)=1, F (1)=2.

4) Функция INTERPR — преобразует инфиксную запись операций выражения в префиксную и возвращает значение выражения.

Контрольные вопросы

1. Чем отличаются символы от атомов в языке ЛИСП?

2. В каких случаях выдается ошибка при попытке интерпретировать S-выражение?

3. Перечислите базовые функции языка ЛИСП. Каковы типы их аргументов и какие значения они возвращают в качестве результата?

4. Вычислите значения следующих выражений:

а) “ (CAR “ (A B C))

б) (EVAL “ (CAR “ (A B C)))

с) (EVAL (CAR “ (A B C)))

д) (EVAL (QUOTE (QUOTE QUOTE)))

е) (QUOTE (EVAL (QUOTE (QUOTE QUOTE))))

5. Каково будет значение атома A после следующих вызовов:

а) (SET (SETQ A “ A) “ B)

б) (SET (SETQ B “ A) (SETQ A “ C))

(SET B A)

6. Какое значение возвращает функция COND?

7. Какое условие должно стоять первым в определении рекурсивной функции?

8. Назовите функции с побочными эффектами. В чем заключается этот эффект?

9. Определите средства, порядок тестирования и отладки функций, реализующих пересечение двух множеств, описанных выше.

Лабораторная работа 2
Сопоставление с образцом

Цель работы:

Изучение метода сопоставления с образцом, изучение алгоритмов сопоставления, приобретение навыков написания программ для сопоставления с образцом.

Постановка задачи

Под сопоставлением с образцом (pattern matching) понимается процедура, при которой с известной символьной структурой, или образцом (pattern), сопоставляется некоторая другая структура, или образ, с целью выявления единообразия или подобия структур или для выявления условий этого. В дальнейшем будем рассматривать только образцы в виде списков. Синтаксически образец — это выражение-шаблон, которое накладывается на другое анализируемое выражение, чтобы определить, имеет ли оно нужную структуру. Возможны два условия сопоставления: удача и неудача.

Образцы могут представляться как одноуровневыми, так и многоуровневыми списками. В образцах могут использоваться как символы — элементы анализируемого образа (представленного также в виде списка), так и символы-заменители, переменные и предикаты, определяющие общую процедуру сопоставления.

Можно выделить следующие символы-заменители:

«?» — произвольный символ образа;

«+» — непустая последовательность символов образа (сегмент);

«*» — сегмент или пустая последовательность;

«-» — символ или пустая последовательность.

Простым образцом будем считать образец, представленный в виде одноуровневого списка, при построении которого использовались приведенные выше заменители и символы (константы), проверяемые на вхождение в образ. Обозначим через PATTERN — образец, а через TEST — образ. Алгоритм сопоставления можно сформулировать в виде следующего правила:

PATTERN сопоставим с TEST,

если TEST=NIL и PATTERN=NIL
или если (TEST не NIL) и (PATTERN не NIL), то
если CAR PATTERN=CAR TEST, то
CDR PATTERN сопоставим с CDR TEST
иначе если CAR PATTERN=?, то
PATTERN «?» — сопоставим с TEST
иначе если CAR PATTERN=+, то
PATTERN «+» — сопоставим с TEST
иначе если CAR PATTERN=*, то
PATTERN «*» — сопоставим с TEST
иначе если CAR PATTERN=-, то
PATTERN «-» — сопоставим с TEST.

Правило сопоставления для заменителя «?»:

PATTERN «?» — сопоставим с TEST, если CDR PATTERN сопоставим с CDR TEST.

Правило сопоставления для заменителя «+»:

PATTERN «+» — сопоставим с TEST, если CDR PATTERN сопоставим с TEST или PATTERN сопоставим с CDR TEST.

Правило сопоставления для заменителя «»:*

PATTERN «*» — сопоставим с TEST, если CDR PATTERN сопоставим с CDR TEST или если CDR PATTERN сопоставим с TEST или PATTERN сопоставим с CDR TEST.

Правило сопоставления для заменителя «-»:

PATTERN «-» — сопоставим с TEST, если CDR PATTERN сопоставим с CDR TEST или если CDR PATTERN сопоставим с TEST.

Основная идея определения функции сопоставления MATCH состоит в том, что образец и образ рассматриваются символ за символом и проверяется выполнение условий. Когда встречается заменитель, то, используя методы управляемого данными программирования, применяется соответствующая функция сопоставления. С каждым символом заменителя связывается список свойств, состоящий из единственного элемента вида: (сопоставитель), где первый элемент — имя свойства, второй элемент — лямбда-выражение функции, реализующий алгоритм сопоставления для заданного заменителя. Для определения соответствующих заменителям функций сопоставления можно составить

соответствующий макрос. Данный макрос по входным параметрам: имени заменителя, параметрам функции сопоставления (PATTERN, TEST), телу функции строит список свойств символа-заменителя. Набор заменителей может быть легко расширен.

Для того чтобы иметь возможность указывать, что на определенных местах образа должны стоять произвольные, но одинаковые выражения, в образец вводятся переменные (параметры). Переменной в процессе отождествления с образцом может быть сопоставлен любой, но один и тот же для всех ее вхождений в образец элемент из отождествляемого списка (образа). Переменные могут определенным образом типизироваться, что разнообразит сопоставление. Например, можно использовать переменные, соответствующие типам ?, +, *, -. Для представления переменных можно пойти двумя путями:

1) представлять переменные в виде двухэлементного списка, первый элемент которого отображает тип сопоставления, а второй — имя переменной, например (? X), (+ Y).

2) представлять переменные в виде символов с определенными префиксами, например ?X, +Y. При этом потребуются символьные функции для работы с атомами. Для Xlisp — это функции SYMBOL-NAME, CHAR, SUBSTR, INTERN.

Если в процессе сопоставления отождествляемый элемент образца есть переменная, то возможно два случая. Первый случай — эта переменная встретилась в первый раз. Такую переменную будем называть свободной. В этом случае следует запомнить значение переменной (то есть сопоставляемый с ней элемент отождествляемого списка) и перейти к отождествлению остатков образца и проверяемого списка. Второй случай — переменная встретилась раньше и поэтому уже имеет значение. Следует заменить в образце переменную на ее значение и повторить процесс сопоставления. Значения переменных удобно хранить в виде ассоциативного списка (назовем его VARVAL). В таком случае интерфейс функции MATCH для сопоставления может быть следующим:

(DEFUN MATCH (PATTERN, TEST, & OPTIONAL VARVAL)).

При успешном сопоставлении данная функция в качестве результата будет возвращать список пар, отображающий значения,

связанные с переменными. Если сопоставление неуспешно, то возвращается NIL.

> (MATCH " ((? X) OR (+ Y)) " (BEFORE OR AFTER NOON))
((X. BEFORE) (Y AFTER NOON))

Для того чтобы легче было различать первый и второй случаи использования переменной, можно ввести соответствующие признаки.

Например, знак «>» в списке переменной соответствует первому случаю, а знак «<» — второму.

> (MATCH " (? (?> X) (*> Y) (< X)) " (SYMBOL SAME MUST BE SAME))
((X. SAME) (Y MUST BE))

Рассмотрим случай, когда образец является многоуровневым списком. В случае, если ни элемент образца, ни элемент сопоставляемого списка не являются атомами, с помощью функции MATCH следует произвести отождествление этих элементов, используя текущее состояние ассоциативного списка значений переменных. Если это отождествление окажется удачным (то есть значение функции MATCH не есть NIL), то следует продолжить сравнение остатка образца и остатка отождествляемого списка с вычисленным функцией MATCH новым списком значений параметров VARVAL.

Пример сопоставления при использовании многоуровневых списков:

> (MATCH " ((+> A) (*(< A)*)), " (Y (X C Y D)))
((A. Y))

В образцах наряду с заменителями и переменными применяется предикатный образец. Предикатное условие можно задать в образце, например, в следующем виде: (p? предикат). Здесь предикат — это функция с одним аргументом, которая получает в качестве аргумента текущий элемент образца. Если предикат истинен, то считается, что сопоставление выполнено.

Пример использования предиката в образце:

> (MATCH " (?? (p? NUMBERP) +) " (A B 5 C))
T

В данном примере проверяется, является ли третий по порядку элемент числом.

Варианты заданий на выполнение

В приведенных вариантах используются следующие обозначения: {?, +, -, *} — заменители, {?X, +X, -X, *X} — переменные соответствующих типов, где X — имя переменной, numberp, symbolp, boundp, minusp, zerop, plusp, evenp, oddp — предикаты языка.

Варианты:

1) (?, +, *); 2) (?, +, -); 3) (?, *, -); 4) (+, *, -); 5) (?X, +X, *X); 6) (?X, +X, -X); 7) (?X, *X, -X); 8) (+X, *X, -X); 9) вариант 1 и (numberp, symbolp); 10) вариант 2 и (evenp, oddp); 11) вариант 3 и (minusp, zerop, plusp); 12) вариант 4 и (symbolp, boundp).

Варианты 13–24 образуются из вариантов 1–12 для случая многоуровневых образцов.

Контрольные вопросы

1. Что понимается под сопоставлением с образцом?
2. Какие заменители используются в образцах?
3. Для чего используются переменные в образцах?
4. Сколько существует случаев сопоставления переменной с образцом?
5. Для чего используется ассоциативный список при сопоставлении с образцом, в котором используются переменные?
6. Назовите типы переменных, используемых в образцах.
7. Каким образом предикатный образец ограничивает сопоставимость?
8. Определите результаты сопоставления:

а) образца (?? K?) и образа (F A C T);

б) образца (+ C T) и образа (O B J E C T);

в) образца (*T R*A C T) и образа (A N T R A C T);

г) образца (? C*) и образа ((A B) C D ((E)));

д) образца (? OR*) и образа (BEFORE OR AFTER NOON);

е) образца (A*D) и образа (A X C Y D);

ж) образца (A ((?> X) C) (< X)) и образа (A (D C) D);

з) образца (*(+> X)*) и образа ((X C Y D) Y);

и) образца (? (p? numberp) (p? symbolp) (p? oddp)*) и образа (AT 5 OR 6 MORNING)

Лабораторная работа 3
Изучение интерпретатора языка Пролог

Цель работы:

Изучение алгоритмов работы интерпретатора языка Пролог, изучение ЛИСП-программы интерпретатора языка Пролог, получение практических навыков построения и модификации интерпретатора языка Пролог на языке ЛИСП.

Постановка задачи

Язык Пролог — язык логического программирования. Теоретической основой Пролога является исчисление предикатов первого порядка. Пролог относится к декларативным языкам. Программирование на языке Пролог состоит из следующих этапов:

1) объявления некоторых фактов об объектах и отношениях между ними;

2) определения некоторых правил об объектах и отношениях между ними;

3) формулировка вопросов об объектах и отношениях.

Правила, факты и цели представляют собой хорновские дизъюнкты (клозы). Правила представляются в виде:

$$B{:-}\ A1, A2, ..., An\ (n>=1),$$

где Ai, i=1, 2, ..., n — условия, B — заключение. Подобная запись правила равносильна импликации вида:

$$A1\ \&\ A2\ \&...\ \&\ An \rightarrow B.$$

Факты представляются в виде клозов, в которых присутствуют только заключения:

$$B{:-}$$

Клозы, в которых присутствуют только условия, вида:

$$A1, A2, ..., An$$

интерпретируются как целевые утверждения (цели), представленные в форме отрицания.

В Прологе имеется мощный встроенный механизм логического вывода с поиском и возвратом. Для доказательства целевых утверждений в Прологе используется семантическая линейная резолюция. Основой построения интерпретатора Пролога

является процедурная интерпретация правил. Учитывая это обстоятельство вместо термина «логический вывод» используется термин «вычисление». Вместо термина «доказательство цели» иногда используется термин «согласование целевого утверждения с базой данных» [9]. Назовем конъюнкцию целей, которую следует доказать на рассматриваемом шаге выполнения программы, резольвентой. Вначале резольвента совпадает с начальной целью (вопросом). Если после удачной унификации очередной цели эту цель заменить телом выбранного правила, то получим резольвенту для очередного шага выполнения программы. Операцию, связанную с заменой цели G телом того же правила из программы P, заголовок которого совпадает с данной целью, называют редукцией. При этом заменяемая цель при редукции называется снятой, а добавляемые цели — производными.

Терм A называется примером (или конкретизацией) терма B, если существует подстановка #, такая, что A=#[B]. Терм #[B] получается одновременной заменой всех вхождений переменных в B на их образы относительно #. Множество термов {T1, T2, ..., Tn} унифицируемо, если существует подстановка # такая, что #[T1]=#[T2]=...=#[Tn]. В этом случае # называют унификатором для {T1, T2, ..., Tn}, так как при ее применении это множество «склеивается» в один элемент. Наиболее общий (или простейший) унификатор (НОУ) L для множества {Ti} обладает тем свойством, что если # — есть любой унификатор для {Ei}, дающий #[Ei], то существует некоторая подстановка #“ такая, что #[Ei]=#“*L[Ei], где*— операция композиции подстановок. Алгоритм унификации будет рассмотрен ниже при изучении рекурсивной функции унификации unify, написанной на языке ЛИСП.

Чтобы избежать путаницы при использовании переменных с одинаковыми именами в разных правилах, условимся переименовывать переменные всякий раз, когда предложение выбирается для выполнения редукции. Среди новых имен не должно быть имен, ранее использованных при вычислении.

Алгоритм работы интерпретатора в первом приближении можно описать так.

Исходные данные. Логическая программа P и вопрос G.

Результат. Выражение Q [G], если этот пример выводится из P, или сообщение о неудаче, если вывод невозможен.

Алгоритм.

1. В качестве начальной резольвенты принять входную цель G.
2. Если резольвента не пуста, то взять в качестве текущей самую левую цель G из резольвенты. Выполнить п. 4.
3. Если резольвента пуста, то полученный пример является ответом на вопрос. Выполнить п. 16.
4. Найти в программе правило, у которого функтор заголовка совпадает с функтором текущей цели.
5. Если правило в программе найдено, то выполнить п. 10.
6. Если правило в программе отсутствует, то выполнить п. 15.
7. Найти в программе очередное альтернативное правило.
8. Если такое правило есть, то выполнить п. 10.
9. Если альтернативных правил больше нет, то выполнить п. 13.
10. Выполнить унификацию выбранного правила и текущей цели.
11. Если унификация закончилась успешно, то осуществить редукцию текущей цели. Выполнить п. 2.
12. Если унификация закончилась неудачей, то выполнить п. 7.
13. Если цель предыдущего шага является начальной целью, то выполнить п. 15.
14. Если цель предыдущего шага не является начальной целью, то заменить текущую цель в резольвенте целью предыдущего шага и выполнить п. 7.
15. Вывести сообщение о неудаче.
16. Закончить выполнение алгоритма.

Интерпретатор, реализующий приведенный алгоритм, должен остановиться после обнаружения первого примера заданной цели. В реальных интерпретаторах предусматривается режим повторного запуска, обеспечивающий получение других примеров цели, если они существуют.

Ниже рассматривается интерпретатор языка Пролог, написанный на языке Xlisp [11]. Взаимосвязи функций интерпретатора представлены, причем стрелки могут интерпретироваться

как отношение «использует» или «вызывает». Функции rename_variables, print-bindings, try-each, unify, value являются рекурсивными, поскольку в определении их содержится вызов самих функций. Кроме того, функции prove и try-each являются взаиморекурсивными, так как они вызывают друг друга.

Запуск интерпретатора осуществляется следующим образом:

(prolog db),

где db — база данных (database). База данных представляется как список, состоящий из фактов и правил. Факты и правила в свою очередь представляются в списочной форме. Переменные в правилах представляются в виде двухэлементного списка вида:

(?).

Порядок следования головы и составляющих тела правила в списке сохраняются. Например, факт, определяемый на Прологе как

father (madelga, ernest)

представляется в виде списка (father madelga ernest), а правило

grandparent (X, Y):- parent (X, Z), parent (Z, Y) —

в виде списка

((grandparent (? X) (? Y))
(parent (? X) (? Z)) (parent (? Z) (? Y))).

Среди прочих основных структур данных можно выделить ассоциативный список environment, в котором сохраняются связи переменных с их значениями (иначе — подстановка). С помощью численного параметра level отличаются связи одноименных переменных из различных правил.

Основной цикл интерпретатора Пролог реализуется функцией верхнего уровня prolog, которой в качестве параметра database передается в Пролог-программу. На языке ЛИСП функция определена следующим образом:

```
(defun prolog (database & aux goal)
(do ( ) ((not (progn (princ «Query?») (setq goal (read)))))
(prove (list (rename-variables goal " (0)))
" ((bottom-of-environment))
database
1)))
```

Функция выводит строку «Query?» и ожидает ввода пользователем целевого утверждения goal. Если ввести пустой список, то интерпретатор завершает свою работу. Введенное целевое утверждение передается функции prove для доказательства. Перед этим, однако, переменные в целевом утверждении переименовываются и им присваивается уровень 0. Список (bottom-of-environment) выступает в качестве маркера дна стека связей переменных.

Функция prove определяется на языке Xlisp следующим образом:

```
(defun prove (list-of-goals environment database level)
(cond ((null list-of-goals)
(print-bindings environment environment)
(not (y-or-n-p «More?»)))
(t (try-each database database
(cdr list-of-goals) (car list-of-goals)
environment level))))
```

Функция prove получает следующие параметры: list-of-goal — резольвенту (в виде списка целей); environment — связи переменных; database — Пролог-программу в списочной форме; level — уровень.

Функция prove проверяет список целей, и если данный список пуст, считается, что целевое утверждение успешно доказано, выводятся связи переменных, вызывается функция y-or-n-p, которая запрашивает, продолжать или нет поиск альтернативных решений при доказательстве целевого утверждения. Если список целей list-of-goal не пуст, то вызывается функция try-each для доказательства целей из этого списка, причем первая цель из этого списка делается текущей.

Текст функции try-each представлен ниже:

```
(defun try-each (database-left database goals-left goal
environment level
&aux assertion new-enviroment)
(cond ((null database-left) nil)
(t (setq assertion
```

```
(rename-variables (car database-left)
(list level)))
(setq new-environment
(unify goal (car assertion) environment))
(cond ((null new-environment)
(try-each (cdr database-left) database
goals-left goal
environment level))
( (prove (append (cdr assertion) goals-left)
new-environment
database
(+ 1 level)))
(t (try-each (cdr database-left) database
goals-left goal
environment level))))))
```

Функция try-each получает следующие параметры: database-left — остаток Пролог-программы, используемой в процессе доказательства; database — всю Пролог-программу; goals-left — еще не доказанные цели из резольвенты; goal — цель, доказываемую в настоящий момент; environment — связи переменных; level — уровень.

Функция try-each проверяет список database-left. Если он пуст, то функция возвращает NIL — признак неудачи при доказательстве. В противном случае она выбирает очередное правило (assertion) из остатка Пролог-программы, переименовывает в нем переменные с помощью функции rename-variables и пытается выполнить унификацию головы выбранного правила (car assertion) и текущей цели (goal) с помощью функции unify. Если унификация прошла неуспешно (список new-environment пуст), то из остатка Пролог-программы выбрасывается текущее «неунифицируемое» правило и вновь вызывается функция try-each в попытке доказать ту же цель goal. Если унификация прошла успешно (список new-environment содержит новые, появившиеся при унификации связи переменных), то осуществляется редукция текущей цели goal путем занесения предикатов тела текущего правила assertion в список недоказанных целей goal-left

и вызов функции prove для доказательства новой резольвенты goal-left. Если доказательство этой резольвенты прошло неуспешно, то из остатка Пролог-программы выбрасывается текущее правило и делается попытка доказать прежнюю резольвенту goal на этом остатке Пролог-программы.

Функция unify предназначена для унификации двух термов — x и y. В качестве третьего параметра функция получает список environment, который представляет текущий НОУ. Функция возвращает НОУ или NIL (NIL — если термы неунифицируемы или НОУ является пустым списком). На языке Xlisp функция unify представляется следующим образом:

```
(defun unify (x y environment & aux new-environment)
(setq x (value x environment))
(setq y (value y environment))
(cond ( (variable-p x) (cons (list x y) environment))
((variable-p y) (cons (list y x) environment))
((or (atom x) (atom y))
(cond ((equal x y) environment)
(t nil)))
(t (setq new-environment (unify (car x) (car y)
environment))
(cond (new-environment (unify (cdr x) (cdr y)
new-environment))
(t nil)))))
```

Алгоритм функции unify (x, y, environment) — «унифицировать» может быть записан в следующем виде:

1. Попытаться присвоить x с помощью функции value значение x.

2. Попытаться присвоить y с помощью функции value значение y.

3. Если x — переменная, то добавить пару (x, y) к строящемуся НОУ (environment). Возврат environment.

4. Если y — переменная, то добавить пару (y, x) к строящемуся НОУ (environment). Возврат environment.

5. Если x или y является атомом и Если x=y, то Возврат environment, Иначе — Возврат NIL.

6. Если условия п. 3, 4, 5 не выполняются, то Делать путем унификации первых элементов списков x и y получить их НОУ — new-environment.

Если new-environment не пуст (унификация прошла успешно), то унифицировать хвосты списков x и y с учетом унификации голов списков x и y и построением НОУ new-environment.

Возврат НОУ, построенного в результате унификации хвостов

Конец

Иначе возврат NIL

Конец

Следует заметить, что НОУ представляет собой список пар вида:

( ).

Функция unify не проверяет вхождение переменной в замещающий терм.

Однако ситуация вхождения переменной в замещающий терм является ошибочной.

Функция value предназначена для определения значения переменной. Параметрами функции являются: x — переменная (в виде списка, начинающегося символом?); environment — связи переменных. Функция возвращает значение переменной, если с переменной связано значение, или имя переменной, если с переменной не связано значение в списке environment. В случае, если x — не переменная, то возвращается само невычисленное x. Текст функции представлен ниже:

```
(defun value (x environment & aux binding)
(cond ( (variable-p x)
(setq binding (assoc x environment :test #"equal))
(cond ((null binding) x)
(t (value (cadr binding) environment))))
(t x)))
```

Следует заметить, что даже неконкретизированная переменная может иметь значение — имя другой переменной, иными словами, ссылку на другую переменную. Таким образом, переменные в ассоциативном списке environment могут «сцепляться», то есть образовывать некий логический список. Функция value, по-

лучив в качестве начала данной логической цепи переменную x путем рекурсивных вызовов, «раскручивает» данную цепочку и возвращает в качестве своего значения элемент-конец цепочки.

Функция variable-p определяет, является ли ее единственный входной параметр x переменной. Текст данной функции имеет следующий вид:

```
(defun variable-p (x)
(and x (listp x) (eq (car x) " ?)))
```

В соответствии с описанием тела функции, x является переменной, если x является списком и первый элемент этого списка есть символ «?».

Встречающиеся в термах переменные «переименовываются» в уникальные имена функцией rename-variables, текст которой на языке Xlisp представлен ниже:

```
(defun rename-variables (term list-of-level)
(cond ((variable-p term) (append term list-of-level))
((atom term) term)
(t (cons (rename-variables (car term) list-of-level)
(rename-variables (cdr term) list-of-level)))))
```

Входными данными функции являются: term — терм в списочной форме; list-of-level — уровень. Функция возвращает терм с «переименованными» переменными. Переименование переменной сводится к добавлению к переменной численного значения уровня, что позволяет в других функциях отличать связи одноименных переменных из различных правил. Пример работы функции rename-variables:

```
> (rename-variables " ((parent (? x) (? y))) " (0))
(parent (? x 0) (? y 0))
```

Функция print-binding выводит значения связей переменных из списка environment-left, являющегося первым параметром функции. На языке Xlisp данная функция определяется как:

```
(defun print-bindings (environment-left environment)
(cond ((cdr environment-left)
(cond ((=0 (nth 2 (caar environment-left)))
(prin1 (cadr (caar environment-left)))
(princ «=»)
```

```
(print (value (caar environment-left)
environment))))
(print-bindings (cdr environment-left) environment))))
```

Функция работает следующим образом. Вначале осуществляется проверка на пустоту хвостовой части печатаемого остатка списка environment-left. Если она не пуста, то проверяется уровень переменной, стоящей первой в списке пар environment-left (переменная определяется предложением caar environment-left). Если он равен нулю, то печатаются имя переменной (стоит на втором месте в списке, представляющем переменную — (cadr (caar environment))), знак «=» и значение переменной, определяемое с помощью функции value. Если хвостовая часть environment-left пуста, то рекурсивно вызывается функция print-bindings по сути с пустым списком, что приводит на следующем этапе рекурсии к возврату вторым оператором cond значения NIL и, таким образом, «затуханию» рекурсии.

Функция y-or-n-p предназначена для ввода ответа, подтверждающего (y) или не подтверждающего (ответ, отличный от y) поиск альтернативного решения Пролог-интерпретатором. Текст функции представлен ниже:

```
(defun y-or-n-p (prompt)
(princ prompt)
(eq (read) " y))
```

Варианты заданий на выполнение

Модифицировать представленный выше Пролог-интерпретатор таким образом, чтобы реализовались следующие конструкции языка Пролог:

1) анонимные переменные. В языке Пролог анонимные переменные обозначаются через символ _. Анонимные переменные индивидуальны, но доступ к ним исключен.

2) предикат fail (отказ), который вызывает при выполнении правила неудачу.

3) арифметические операции: сложение, вычитание, умножение, деление. Операторы могут быть заданы в префиксной форме и представлены в виде термов.

4) операции сравнения: больше, больше или равно, меньше, меньше или равно, равно, не равно. Операторы могут быть заданы в префиксной форме и представлены в виде термов.

5) предикат cut (отсечение), используемый для управления бектрекингом. Обычно в языке Пролог данный оператор обозначается символом «!». Его действие состоит в том, что переменные, появляющиеся в текущем правиле слева от cut, после конкретизации становятся неизменяемыми. Иными словами, запрещается использовать все альтернативные выводы конъюнкции целей, находящихся левее от cut.

6) предикат repeat, предназначенный для порождения множественных решений в процессе возврата. Этот предикат генерирует бесконечное число циклов повторения поиска с возвратом. Предикат всегда истинен. Всякий раз, как до него доходит перебор, порождается новая ветвь вычислений. Поведение предиката полностью соответствует следующему Пролог-определению:

repeat.

repeat:- repeat.

7) предикаты retract и assert, которые соответственно удаляют и добавляют хорновские клозы в базу данных.

8) предикат call. Целевое утверждение call (X) считается согласованным с базой данных, если попытка доказать согласованность X завершается успехом.

9) предикат not. Целевое утверждение not (X) считается согласованным с базой данных, если попытка доказать согласованность X заканчивается неудачей. Целевое утверждение not (X) считается несогласованным, если попытка доказать согласованность X успешно завершается.

10) дизъюнкция целей. В Прологе для определения дизъюнкции целевых утверждений используется функтор «;». Если X и Y — целевые утверждения, то целевое утверждение X; Y согласуется с базой данных, если согласуется X или Y. В случае предложенного выше ЛИСП-интерпретатора Пролога необходимо внести коррективы в представление Пролог-программы. Можно ввести дополнительные списки, в которые включаются конъюнкции целей.

При этом дизъюнкция будет представлять собой список конъюнкций. Например, прологовское правило

F:- (A, B); C

можно записать в виде списка (F (A B) (C)).

11) операторы ввода и вывода термов. Для вывода в языке Пролог используется предикат write. Если значением переменной X является терм, то появление цели write (X) вызовет печать этого терма на дисплее. В случае если переменная X неконкретизирована, будет напечатан некоторый числовой уникальный идентификатор, начинающийся со знака «_». Для ввода термов в языке Пролог используется предикат read. Если переменная X не конкретизирована, то целевое утверждение read (X) приведет к вводу следующего терма и этот терм будет присвоен в качестве значения переменной X. Если в момент рассмотрения целевого утверждения read (X) его аргумент конкретизирован, то попытка доказать согласованность этого целевого утверждения с базой данных вызовет чтение следующего терма и попытку сопоставления его с аргументом, заданным в read. Согласованность цели с базой данных зависит от результатов этого сопоставления.

12) предикат trace. Выполнение предиката trace заключается в установлении режима полной трассировки. Трассировка описывается в терминах четырех видов происходящих событий: CALL (ВЫЗОВ), EXIT (ВЫХОД), REDO (ПЕРЕДЕЛКА), FAIL (НЕУДАЧА). Событие CALL фиксирует начало попытки Пролога согласовать цель с базой данных. Событие EXIT фиксирует момент, когда некоторая цель только что согласована с базой данных. Событие REDO фиксирует момент, когда система возвращается к цели, пытаясь повторно согласовать ее с базой данных. Событие FAIL фиксирует момент, когда попытка согласовать цель с базой данных заканчивается неудачно.

13) предикат functor. Предикат functor определен таким образом, что functor (T, F, N) означает, что T — это терм с функтором F, имеющим N аргументов.

4) предикат arg, используемый для доступа к конкретному аргументу терма. Предикат arg определен таким образом, что arg (N, T, A) означает, что N-й аргумент T есть A. Предикат arg всегда

должен использоваться с конкретизированными первым и вторым аргументами.

15) возможность работы со списками. Для представления списков в Прологе обычно используется скобочная форма записи. Она представляет собой заключенную в квадратные скобки последовательность элементов списка, разделенных запятыми. Например, в Прологе допустимы следующие списки:

[ ] — пустой список;

[a, b, [c, d, e]] — вложенные списки;

[a, V1, b, [X, Y]] — список, включающий переменные.

Работа со списками основана на расщеплении их на голову и хвост. В Прологе введена специальная форма для представления списка с головой X и хвостом Y. Это записывается как [X|Y], где для разделения X и Y используется вертикальная черта |. При реализации работы со списками следует модифицировать форму представления Пролог-программы, предложенную выше. При этом необходимо ввести специальные маркеры для идентификации списков (возможно, маркеры начала, конца, головы, хвоста).

16) предикат «=...», используемый для сборки или разборки терма. Целевое утверждение X=... L означает, что L есть список, состоящий из функтора терма X, за которым следуют аргументы X. Для реализации данного предиката должны быть реализованы средства работы со списками.

17) модифицировать функцию unify таким образом, чтобы она проверяла вхождение переменной в заменяемый терм и в случае вхождения выдавала сообщение об ошибке.

Контрольные вопросы

1. Что понимается под хорновскими дизъюнктами?
2. Перечислите основные виды хорновских дизъюнктов, используемых в языке Пролог.
3. Что такое резольвента?
4. В чем суть операции редукции цели?
5. Для чего используется переименование переменных в ходе интерпретации Пролог-программы?
6. Что такое унификация и наиболее общий унификатор?

7. Почему в подстановке переменная не должна входить в заменяемый терм?

8. Определите синтаксис списочной формы представления Пролог-программы для интерпретатора на языке ЛИСП, приведенного выше.

9. Для чего используется в данном интерпретаторе объект environment?

10. Каким образом в интерпретаторе осуществляется переименование переменных?

11. Как представляется в интерпретаторе резольвента и каким образом осуществляется редукция цели?

IX. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ЯЗЫКА ПРОГРАММИРОВАНИЯ ЛИСП

1. Основные особенности языка ЛИСП

От других языков программирования ЛИСП отличается следующими свойствами:

1. Одинаковая форма данных и программ.
2. Хранение данных, не зависящее от места.
3. Автоматическое и динамическое управление памятью.
4. Функциональная направленность.
5. ЛИСП — бестиповый язык.
6. Интерпретирующий и компилирующий режимы работы.
7. Пошаговое программирование.
8. Единый системный и прикладной язык программирования.

Теперь рассмотрим эти свойства более подробно.

1.1. Одинаковая форма данных и программ

В ЛИСПе формы представления программы и обрабатываемых ею данных одинаковы. И то и другое представляется списочной структурой, имеющей одинаковую форму. Таким образом, программы могут обрабатывать и преобразовывать другие программы и даже самих себя. В процессе трансляции можно введенное и сформированное в результате вычислений выражение данных проинтерпретировать в качестве программы и непосредственно выполнить. Это свойство обладает не только теоретическим, но и большим практическим значением.

Универсальный единообразный и простой ЛИСПовский синтаксис списка не зависит от применения, и с его помощью легко определять новые формы записи, представления и абстракции.

Даже сама структура языка является, таким образом, расширяемой и может быть заново определена. В то же время достаточно просто написание интерпретаторов, компиляторов, редакторов и других средств. К ЛИСПу необходимо подходить как к языку, с помощью которого реализуются специализированные языки, ориентированные на приложение, и создается окружение более высокого уровня. Присущая ЛИСПу расширяемость не встречается в традиционных замкнутых языках программирования.

1.2. Хранение данных, не зависящее от места

Списки, представляющие программы и данные, состоят из списочных ячеек, расположение и порядок которых в памяти не существенны. Структура списка определяется логически на основе имен символов и указателей. Добавление новых элементов в список или удаление из списка может производиться без переноса списка в другие ячейки памяти. Резервирование и освобождение могут в зависимости от потребности осуществляться динамически, ячейка за ячейкой.

1.3. Автоматическое и динамическое управление памятью

Пользователь не должен заботиться об учете памяти. Система резервирует и освобождает память автоматически в соответствии с потребностью. Когда памяти не хватает, запускается специальный мусорщик. Мусорщик перебирает все ячейки и собирает являющиеся мусором ячейки в список свободной памяти для того, чтобы их можно было использовать заново. Среда ЛИСПа постоянно содержится в порядке. В современных ЛИСП-системах выполнение операции сбора мусора занимает от одной до нескольких секунд. В задачах большого объема сборщик мусора запускается весьма часто, что резко ухудшает временные характеристики прикладных программ. Во многих системах мусор не образуется, поскольку он сразу же учитывается. Управление памятью просто и не зависит от физического расположения, поскольку свободная память логически состоит из цепочки списочных ячеек.

В первую очередь данные обрабатываются в оперативной и виртуальной памяти, которая может быть довольно большой.

Файлы используются в основном для хранения программ и данных в промежутках между сеансами.

1.4. Функциональная направленность ЛИСПа

Функциональное программирование, используемое в ЛИСПе, основывается на том, что в результате каждого действия возникает значение. Значения становятся элементами следующих действий, и конечный результат всей задачи выдается пользователю. Обойти это можно только при помощи специальной функции QUOTE. То обстоятельство, что результатом вычислений могут быть новые функции, является важным преимуществом ЛИСПа.

1.5. ЛИСП — бестиповый язык

В ЛИСПе имена символов, переменных, списков, функций и других объектов не закреплены предварительно за какими-нибудь типами данных. Типы в общем не связаны с именами объектов данных, а сопровождают сами объекты. Переменные в разные моменты времени могут представлять разные объекты. В этом смысле ЛИСП является бестиповым языком.

Динамическая, осуществляемая лишь в процессе исполнения проверка типа и позднее связывание допускают разностороннее использование символов и гибкую модификацию программ. Функции можно определять практически независимо от типов данных, к которым они применяются.

Но указанная бестиповость не означает, что в ЛИСПе нет данных различных типов. Наоборот, набор типов данных наиболее развитых ЛИСП-систем очень разнообразен.

Одним из общих принципов развития ЛИСП-систем было свободное включение в язык новых возможностей и структур, если считалось, что они найдут более широкое применение. Это было возможно в связи с естественной расширяемостью языка.

Платой за динамические типы являются действия по проверке типа на этапе исполнения. В более новых ЛИСП-системах (Common Lisp) возможно факультативное определение типов. В этом случае транслятор может использовать эту информацию

для оптимизации кода. В ЛИСП-машинах проверка осуществляется на уровне аппаратуры.

1.6. Интерпретирующий и компилирующий режимы работы

ЛИСП в первую очередь интерпретируемый язык. Программы не нужно транслировать, и их можно исправлять в процессе исполнения. Если какой-то участок программы отлажен и не требует изменений, то его можно оттранслировать, тогда она выполняется быстрее. В одной и той же программе могут быть транслированные и интерпретированные функции. Транслирование по частям экономит усилия программиста и время вычислительной машины.

Однако компилирующий режим предусмотрен далеко не во всех ЛИСП-системах, его использование накладывает ряд дополнительных ограничений на технику программирования.

1.7. Пошаговое программирование

Программирование и тестирование программы осуществляется функция за функцией, которые последовательно определяются и тестируются. Написание, тестирование и исправление программы осуществляются внутри ЛИСП-системы без промежуточного использования ОС.

Вспомогательные средства, такие, например, как редактор, трассировщик, транслятор и другие образуют общую интегрированную среду, язык которой нельзя отличить от системных средств. Отдельные средства по своему принципу являются прозрачными, чтобы их могли использовать другие средства. Часто работа может производиться на различных уровнях или в различных рабочих окнах. Такой способ работы особенно хорошо подходит для исследовательского программирования и быстрого построения прототипов.

1.8. Единый системный и прикладной языки программирования

ЛИСП является одновременно как языком прикладного, так и системного программирования. Он напоминает машинный язык

тем, что как данные, так и программы представлены в одинаковой форме. Язык превосходно подходит для написания интерпретаторов и трансляторов как для него самого, так и для других языков. Наиболее короткий интерпретатор Пролога, написанный на ЛИСПе, занимает несколько десятков строк.

Традиционно ЛИСП-системы в основном написаны на ЛИСПе. ЛИСП можно в хорошем смысле считать языком машинного и системного программирования высокого уровня. И это особенно хорошо для ЛИСП-машин, которые вплоть до уровня аппаратуры спроектированы для ЛИСПа, и системное программное обеспечение которых также написано на ЛИСПе.

2. Понятия языка ЛИСП

2.1. Атомы и списки

Основная структура данных в ЛИСПе — символьные или S-выражения, которые определяются как атомы или списки.

Символы и числа представляют собой те объекты ЛИСПа, из которых строятся остальные структуры. Поэтому их называют атомами.

Символ — это имя, состоящее из букв, цифр и специальных знаков, которое обозначает какой-нибудь предмет, объект, действие из реального мира. В ЛИСПе символы обозначают числа, другие символы или более сложные структуры, программы (функции) и другие ЛИСПовские объекты.

Пример: x, 1997, символ, function.

Числа не являются символами, так как число не может представлять иные ЛИСПовские объекты, кроме самого себя или своего числового значения. В ЛИСПе используется большое количество различных типов чисел (целые, десятичные и т. д.) — 24, 35.6, 6.3.

Символы T и NIL имеют в ЛИСПе специальное назначение: T обозначает логическое значение ИСТИНА, а NIL — логическое значение ЛОЖЬ. Символы T и NIL имеют всегда одно и то же фиксированное встроенное значение. Их нельзя использовать в качестве имен других ЛИСПовских объектов. Символ NIL обозначает также и пустой список.

Числа и логические значения T и NIL являются константами, остальные символы — переменными, которые используются для обозначения других ЛИСПовских объектов.

Список — это упорядоченная последовательность, элементами которой являются атомы или списки (подсписки). Списки заключаются в круглые скобки, элементы списка разделяются пробелами. Открывающие и закрывающие скобки находятся в строгом соответствии. Список всегда начинается с открывающей и заканчивается закрывающей скобкой. Список, который состоит из 0 элементов называют пустым и обозначают ( ) или NIL. Пустой список — это атом. Например: (а в (с о) р), (+ 3 6).

2.2. Внутреннее представление списка

Оперативная память машины логически разбивается на маленькие области, называемые списочными ячейками. Списочная ячейка состоит из двух частей, полей CAR и CDR (головы и хвоста соответственно). Каждое из полей содержит указатель. Указатель может ссылаться на другую списочную ячейку или на другой ЛИСПовский объект, например атом. Указатели между ячейками образуют как бы цепочку, по которой можно из предыдущей ячейки попасть в следующую и так до атомарных объектов. Каждый известный системе атом записан в определенном месте памяти лишь один раз. Указателем списка является указатель на первую ячейку списка. На одну и ту же ячейку может быть направлено несколько указателей, но исходит только один.

Графически списочная ячейка представляется прямоугольником, разделенным на части CAR и CDR. Указатель изображается в виде стрелки, начинающейся в одной из частей прямоугольника и заканчивающейся на изображении другой ячейки или атоме, на которые ссылается указатель.

Логически идентичные атомы содержатся в системе лишь один раз. Но логически идентичные списки могут быть представлены различными списочными ячейками, например список ((d c) a d c). В результате вычислений в памяти могут возникнуть структуры, на которые нельзя потом сослаться. Это происходит, если струк-

туры не сохранены с помощью функции SETQ или теряется ссылка на старое значение в связи с новым переприсваиванием.

В зависимости от способа построения, логическая и физическая структуры двух списков могут оказаться различными. Например, список ((d c) a d c) может иметь и другую структуру. Логическая структура всегда топологически имеет форму двоичного дерева, в то время как физическая структура может быть ациклическим графом, то есть ветви могут снова сходиться, но никогда не могут образовывать замкнутые циклы (указывать назад).

При логическом сравнивании списков используют предикат EQUAL, сравнивающий не физические указатели, а совпадение структурного построения списков и совпадение атомов, формирующих список.

Предикат EQ можно использовать лишь для сравнения двух символов. Во многих реализациях ЛИСПа этот предикат обобщен таким образом, что с его помощью можно определить физическое равенство двух выражений (то есть ссылаются ли значения аргументов на один физический ЛИСПовский объект) не зависимо от того, является ли он атомом или списком.

2.3. Написание программы на ЛИСПе

Любая ЛИСП-система представляет собой небольшую программу, назначение которой — выполнение программ путем интерпретации S-выражений, подаваемых на вход. Механизм работы ЛИСП-системы очень прост. Он состоит из трех последовательных шагов: считывание S-выражения; интерпретация S-выражения; печать S-выражения.

Обычно, написание программы на ЛИСПе — написание некоторой функции, возможно, весьма сложного вида, при вычислении использующей какие-либо другие функции или рекурсивно саму себя. Однако на практике часто оказывается, что более удобно решать задачи путем выполнения последовательности отдельных более или менее простых шагов с сохранением и дальнейшим использованием промежуточных результатов. Шагом в нашем случае будет вычисление некоторой функции ЛИСПа. Разрешая использовать переменные не только в качестве аргументов функций,

мы расстаемся с чистотой функционального программирования, но вместе с тем приобретаем инструмент, в ряде случаев облегчающий написание программ и нередко повышающий эффективность их работы на ВМ с традиционной архитектурой. Наиболее широко в практическом программировании на ЛИСПе распространен смешанный стиль программирования: программу стараются писать функционально, но при этом соответствующие функции не делают слишком сложными, переходя везде, где это облегчает написание соответствующей функции и понимание принципа ее работы, к императивному (второму) методу программирования.

Итак, как правило, программа, написанная на ЛИСПе, представляет собой последовательность вызовов некоторых функций, как встроенных в ЛИСП, так и предварительно описанных самим пользователем, а средством связи между последовательно вызываемыми функциями являются переменные, позволяющие запомнить любой объект (атом, список, точечную пару).

Все ЛИСП-системы имеют некоторый набор базовых функций (их мы рассматриваем в лабораторных работах), которые изначально встроены в интерпретатор. Кроме того, пользователь может определять собственные функции на языке ЛИСП, используя специальные конструкторы функций. Если атом f не удается интерпретировать как встроенную функцию языка или как функцию, определенную пользователем, большинство интерпретаторов выдают сообщение об ошибке.

Множества базовых функций различных диалектов сильно отличаются друг от друга, и их число колеблется от нескольких десятков до нескольких сотен. Встроенные функции выполняются с большей скоростью, чем функции, определенные пользователем, так как первые реализованы на том же языке, что и вся ЛИСП-система (Ассемблер, С, Паскаль). Чрезмерное увеличение базового набора функций приводит к уменьшению рабочей памяти, отводимой под задачи пользователя.

2.4. Определение функций

Определение функций и их вычисление в ЛИСПе основано на лямбда-исчислении Черча. В лямбда-исчислении Черча функция записывается в следующем виде:

lambda (x1, x2, ..., xn). fn

В ЛИСПе лямбда-выражение имеет вид

(LAMBDA (x1 x2 ... xn) fn)

Символ LAMBDA означает, что мы имеем дело с определением функции. Символы xi являются формальными параметрами определения, которые имеют аргументы в описывающем вычисления теле функции fn. Входящий в состав формы список, образованный из параметров, называют лямбда-списком.

Телом функции является произвольная форма, значение которой может вычислить интерпретатор ЛИСПа.

Формальность параметров означает, что их можно заменить любыми другими символами и это не отразится на вычислениях, определяемых функцией.

Лямбда-выражение — это определение вычислений и параметров функции в чистом виде без фактических параметров или аргументов. Для того чтобы применить такую функцию к некоторым аргументам, нужно в вызове функции поставить лямбда-определение на место имени функции:

(лямбда-выражение a1 a2 ... an)

Здесь ai — формы, задающие фактические параметры, которые вычисляются как обычно.

Вычисление лямбда-вызова, или применение лямбда-выражения к фактическим параметрам, производится в два этапа. Сначала вычисляются значения фактических параметров и соответствующие формальные параметры связываются с полученными значениями. Этот этап называется связыванием параметров. На следующем этапе с учетом новых связей вычисляется форма, являющаяся телом лямбда-выражения, и полученное значение возвращается в качестве значения лямбда-вызова. Формальным параметрам после окончания вычисления возвращаются те связи, которые у них, возможно, были перед вычислением лямбда-вызова.

Лямбда-вызовы можно свободно объединять между собой и другими формами. Вложенные лямбда-вызовы можно ставить как на место тела лямбда-выражения, так и на место фактических параметров.

Лямбда-выражение является как чисто абстрактным механизмом для описания и определения вычислений, так и механизмом для связывания формальных и фактических параметров на время выполнения вычислений. Лямбда-выражение — это безымянная функция, которая пропадает тотчас после вычисления значения формы. Ее трудно использовать снова, так как нельзя вызвать по имени, хотя ранее выражение было доступно как списочный объект. Однако используются и безымянные функции, например, при передаче функции в качестве аргумента другой функции или при формировании функции в результате вычислений, другими словами, при синтезе программ.

Лямбда-выражение соответствует используемому в других языках определению процедуры или функции, а лямбда-вызов — вызову процедуры или функции. Записывать вызовы функций полностью с помощью лямбда-вызовов не разумно, поскольку очень скоро выражения в вызове пришлось бы повторять, хотя разные вызовы одной функции отличаются лишь в части фактических параметров. Проблема разрешима путем именования лямбда-выражений и использования в вызове лишь имени.

Дать имя и определить новую функцию можно с помощью функции DEFUN:

(DEFUN имя лямбда-список тело)

DEFUN соединяет символ с лямбда-выражением, и символ начинает представлять определенные этим лямбда-выражением вычисления. Значением этой формы является имя новой функции.

В различных диалектах ЛИСПа вместо DEFUN используются другие имена и формы (DEFINE, DE, CSETQ и др.).

2.5. Рекурсия и итерация

Будучи языком функций, символьным языком и языком списков, ЛИСП является и рекурсивным языком. В программировании на ЛИСПе рекурсия используется для организации повторяющихся вычислений. На ней же основано разбиение проблемы на подзадачи, решение которых пытаются свести к уже решенной или решаемой в данный момент задачи.

Основная идея рекурсивного определения заключается в том, что функцию можно с помощью рекуррентных формул свести к некоторым начальным значениям, к ранее определенным функциям или к самой определяемой функции, но с более простыми аргументами. Вычисление такой функции заканчивается в тот момент, когда оно сводится к известным начальным значениям. Разумеется, можно организовать вычисления по рекуррентным формулам и без использования рекурсии, однако при этом встает вопрос о ясности программы и доказательстве ее эквивалентности исходным формулам. Использование рекурсии позволяет легко запрограммировать вычисление по рекуррентным формулам.

В рекурсивном описании действий имеет смысл обратить внимание на следующие обстоятельства. Во-первых, процедура содержит всегда по крайней мере одну терминальную ветвь и условие окончания. Во-вторых, когда процедура доходит до рекурсивной ветви, то функционирующий процесс приостанавливается, и новый такой же процесс запускается сначала, но уже на новом уровне. Прерванный процесс каким-нибудь образом запоминается. Он будет ждать и начнет исполняться лишь после окончания нового процесса. В свою очередь, новый процесс может приостановиться, ожидать и т. д.

Так образуется как бы стек прерванных процессов, из которых выполняется лишь последний в настоящий момент времени процесс; после окончания его работы продолжает выполняться предшествовавший ему процесс. Целиком весь процесс выполнен, когда стек снова опустеет или, другими словами, все прерванные процессы выполнятся.

Можно говорить о рекурсии по значению, когда вызов является выражением, определяющим результат функции. Если в качестве результата функции возвращается значение некоторой другой функции и рекурсивный вызов участвует в вычислении аргументов этой функции, то будем говорить о рекурсии по аргументам. Аргументом рекурсивного вызова может быть новый рекурсивный вызов, и таких вызовов может быть много.

Для обеспечения «идеологической» совместимости с другими языками программирования, а также для повышения

эффективности программ при решении некоторых частных задач в язык была введена группа функций, предоставляющих возможность организации итерационной обработки информации.

В указанной группе прежде всего выделяются так называемые отображающие или MAP-функции: MAPC, MAPCAR, MAPLIST и др. MAP — это функции, которые некоторым образом отображают список (последовательность) в новую последовательность или порождают побочный эффект, связанный с этой последовательностью. Каждая из них имеет более двух аргументов, значением первого должно быть имя определенной ранее или базовой функции, или лямбда-выражение, вызываемое MAP-функцией итерационно, а остальные аргументы служат для задания аргументов на каждой итерации. Естественно, что количество аргументов в обращении к MAP-функции должно быть согласовано с предусмотренным количеством аргументов у аргумента-функции. Различие между всеми MAP-функциями состоит в правилах формирования возвращаемого значения и механизме выбора аргументов итерирующей функции на каждом шаге.

Рассмотрим основные типы MAP-функций.

MAPCAR

Значение этой функции вычисляется путем применения функции fn к последовательным элементам xi списка, являющегося вторым аргументом функции. Например, в случае одного списка получается следующее выражение:

(MAPCAR fn " (x1 x2 ... xn))

В качестве значения функционала возвращается список, построенный из результатов вызовов функционального аргумента MAPCAR.

MAPLIST

MAPLIST действует подобно MAPCAR, но действия осуществляет не над элементами списка, а над последовательными CDR этого списка.

Функционалы MAPCAR и MAPLIST используются для программирования циклов специального вида и в определении других функций, поскольку с их помощью можно сократить запись повторяющихся вычислений.

Функции MAPCAN и MAPCON являются аналогами функций MAPCAR и MAPLIST. Отличие состоит в том, что MAPCAN и MAPCON не строят, используя LIST, новый список из результатов, а объединяют списки, являющиеся результатами, в один список.

LOOP

Другим типичным представителем группы итерационных функций может служить функция LOOP, имеющая в общем случае вид (LOOP expr1 expr2 ... exprN), где в качестве аргументов могут быть использованы любые синтаксически и семантически допустимые S-выражения либо специальные конструкции.

2.6. Функции интерпретации выражения

Почти во всех диалектах определены функции APPLY и FUNCALL, позволяющие интерпретировать S-выражения. Обращения к этим функциям имеют следующий вид:

(APPLY fun arg)

(FUNCALL fun arg1 arg2 ... argN)

APPLY и FUNCALL вычисляют функции, являющиеся их первыми аргументами, производя связывание формальных аргументов с указанными S-выражениями arg или arg1, arg2, ... argN. В качестве значения возвращается результат применения функции fun, которая может быть встроенной или определенной функцией или лямбда-выражением.

Необходимо отметить еще одну особенность языка ЛИСП, которая вытекает из природы организации структур данных и программ и механизма их интерпретации. На языке ЛИСП легко реализовать задачи автоматического синтеза программ. Он позволяет с помощью одних функций формировать определения других функций, программно анализировать и редактировать эти определения как S-выражения, а затем исполнять их.

2.7. Макросредства

Часто бывает полезно не выписывать вычисляемое выражение вручную, а сформировать его с помощью программы. Эта идея автоматического динамического программирования особенно

хорошо реализуется в ЛИСПе. Программное формирование выражений наиболее естественно осуществляется с помощью специальных макросов. Использование макросредств, предлагаемых современными ЛИСП-системами, — один из самых эффективных путей реализации сложных программ. Макросы дают возможность расширять синтаксис и семантику ЛИСПа и использовать новые подходящие для решаемой задачи формы предложений. Они дают возможность писать компактные, ориентированные на задачу программы, которые автоматически преобразуются в более сложный, но более близкий машине эффективный ЛИСПовский код. При наличии макросредств некоторые функции в языке могут быть определены в виде макрофункций. Такое определение фактически задает закон предварительного построения тела функции непосредственно перед фазой интерпретации.

Синтаксис определения макроса выглядит так же, как синтаксис используемой при определении функций формы DEFUN:

(DEFMACRO имя лямбда-список тело)

Вызов макроса совпадает по форме с вызовом функции, но его вычисление отличается от вычисления вызова функции. Первое отличие состоит в том, что в макросе не вычисляются аргументы. Тело макроса вычисляется с аргументами в том виде, как они записаны.

Второе отличие состоит в том, что интерпретация функций, определенных как макро, производится в два этапа. На первом, называемом макрорасширением, происходит формирование лямбда-определения функции в зависимости от текущего контекста, на втором осуществляется интерпретация созданного лямбда-выражения.

Макросы отличаются от функций и в отношении контекста вычислений. Во время расширения макроса доступны синтаксические связи из контекста определения. Вычисление же полученной в результате расширения формы производится вне контекста макровызова, и поэтому статические связи из макроса не действуют. Использование макрофункций облегчает построение языка с ЛИСПоподобной структурой, имеющего

свой синтаксис, более удобный для пользователя. Чрезмерное использование макросредств затрудняет чтение и понимание программ.

2.8. Функции ввода-вывода

Современные диалекты языка ЛИСП, как правило, имеют развитые средства управления вводом-выводом. Основу этих средств составляют три основные функции READ, RATOM и PRINT. Первые две позволяют осуществлять операции ввода S-выражений (READ) и атомов (RATOM), последняя выполняет вывод S-выражений.

ЛИСПовская функция чтения READ обрабатывает выражение целиком. Вызов функции осуществляется в виде

(READ)

Функция не показывает, что она ждет ввода выражения. Она лишь читает выражение и возвращает в качестве значения само это выражение, после чего вычисления продолжаются.

Для вывода выражений используют функцию PRINT. Это функция с одним аргументом, которая сначала вычисляет значение аргумента, а затем выводит это значение. Функция PRINT перед выводом аргумента переходит на новую строку, а после него выводит пробел. Таким образом, значение выводится всегда на новую строку. Более подробно эти и другие функции рассмотрим в лабораторных работах.

Базовый набор функций обычно наряду с указанными функциями включает различные их модификации и дополнительные функции, позволяющие при программировании легко получить некоторые дополнительные эффекты (автоматическое дополнение печатной строки с изображением S-выражения специальными символами перевода каретки на начало следующей строки, блокировка специальных метасимволов при выводе литеральных атомов, в печатных именах которых присутствуют непечатные символы, и т. д.). Кроме того, практически каждый диалект содержит набор функций управления входными и выходными потоками для связи с внешними устройствами ЭВМ. Однако указанные функции являются одной из наиболее

машинно-зависимых составляющих ЛИСП-систем, поскольку по необходимости учитывают специфику среды операционной системы.

3. Знания в ИИ

3.1. Требования к знаниям

Знания должны отвечать следующим требованиям:

1. Должны быть явными и доступными. Это главное отличие знаний от других программных продуктов.

2. Использовать высококачественный опыт специалистов, то есть знания должны отражать уровень наиболее квалифицированных специалистов в данной области.

3. Обладать гибкостью, то есть система может наращиваться постепенно, в соответствии с нуждами бизнеса или заказчика.

4. Иметь систему объяснений. Интересует не только ответ, но и то, как машина к нему пришла.

5. Обладать возможностью прогнозирования. Система должна не только давать ответы на конкретно поставленные вопросы, но и показывать, как они изменяются в новой ситуации.

6. Память должна быть инстуциональной, то есть специалисты уходят, их опыт остается. База знаний становится постоянно обновляющимся справочником наилучших стратегий и методов.

3.2. Основные типы знаний

Определение знания как понятия — трудная проблема. В области ИИ наиболее важные типы знаний классифицируются следующим образом:

1. Объекты и их свойства

Объекты — это существующие в прикладной области универсальные понятия и их представители: живые существа, предметы или материалы или абстрактные понятия.

2. События

События описывают участие объектов в деятельности и ситуациях. События характеризуют, например, время, место и причинно-следственные отношения.

3. Действия

Обычно интеллектуальная деятельность предполагает также способность совершать действия, то есть процедурные знания о том, каким образом что-то делается, например, каким образом из старых данных на основе правил выводятся новые.

4. Метазнания

Метазнания — это знания о знаниях и их использовании, например способность выбрать метод решения для проблем разных типов.

3.3. Методы представления знаний

Представление знаний — это основная область исследований по ИИ. Особенности представления знаний и механизм логического вывода определяют два основных элемента ЭС–БЗ и машину логического вывода. Любая работающая со знаниями программа должна каким-то образом отображать знания из своей области применения. Обычно для этого не достаточно примитивных структур данных, используемых в традиционной обработке данных, таких как числа, массивы, записи и другие, и методов работы с ними. В ИИ используются символьные языки представления знаний и формализмы, стоящие на более высоком понятийном уровне.

Рассмотрим важнейшие из общих методов и формализмов, разработанных для представления и обработки знаний:

1. Логика

В программах ИИ особенно часто используются различные формы логики предикатов первого порядка. Основное преимущество базирующихся на логике формализмов состоит в том, что обычно с их помощью проще обеспечить корректность структур и решений системы, чем с помощью других способов представления.

2. Продукционные системы

Наиболее распространенным и простым для понимания является представление знаний при помощи правил продукции вида «ЕСЛИ <условие>, ТО <следствие>». Такие системы называют продукционными. Эти правила похожи на условные операторы

IF-THEN языков программирования, однако совершенно иначе интерпретируются.

Через правила можно определить, как программа должна реагировать на изменение данных. При этом не нужно заранее знать блок-схему управления обработкой данных. В обычной программе схема передачи управления и использования данных предопределена самой программой. Ветвление в таких программах возможно только в заранее выбранных точках. Для задач, ход решения которых управляется самими данными, где ветвление скорее норма, чем исключение, этот способ малоэффективен.

В состав продукционных систем входят: база знаний (используют термин: «база правил»); рабочая память или база данных; машина вывода (используют термин: «управляющая структура»). База правил содержит правила продукций. Рабочая память отображает текущее состояние процесса консультации. Содержит текущие значения переменных и состояние машины вывода. Машина вывода, являющаяся, по сути, интерпретатором правил, определяет последовательность активизации правил, выполняет их, частично заполняет рабочую память по собственной инициативе и частично по инструкциям из базы правил. Работа интерпретатора правил состоит из циклически повторяющихся этапов. Сначала определяется, какие правила могут выполняться в данный момент, для чего отдельные части правил сравниваются с информацией, хранимой в рабочей памяти. Затем определяется, какое правило следует выполнять первым. Критерием может быть приоритет, скорость выполнения правила и некоторые другие вещи. Затем правило исполняется, под чем подразумевается изменение рабочей области, внутренних переменных машины вывода и окружения.

Существует два основных метода просмотра и выполнения правил. Это прямой и обратный выводы.

Прямой вывод, управляемый посылками правил, похож на выполнение зацикленной программы, на алгоритмическом языке, представляющей собой набор условных операторов. При прямом выводе выполняются только те правила, посылки которых принимают истинное значение. Процесс поиска и выполнения пра-

вил продолжается до тех пор, пока не будет достигнута цель консультации, или не будут исчерпаны все правила и ни в одном посылка не окажется истинной.

При обратном выводе, управляемом следствиями или целями правил (используется для создания ЭС диагностики и интерпретации), происходит движение от следствий к посылкам. Машина вывода выбирает очередную неизвестную переменную и пытается присвоить ей какое-то значение. Для этого она просматривает все правила, в THEN-поле которых присутствует эта переменная, и проверяет посылку правила. Если в посылке правила присутствуют не определенные переменные, то аналогичным способом машина вывода пытается найти их. Если правило с искомой переменной не выполняется, то ищутся другие правила, содержащие в THEN-поле эту переменную.

Возможен так же смешанный вывод.

3. Семантические сети

В семантической сети понятия и классы, а также отношения и связи между ними представлены в виде поименованного графа. Каждый узел содержит ссылку на один или несколько других узлов. Ссылка представляет собой также понятие, устанавливающее взаимосвязь между узлами. Предполагается, что понятий-ссылок меньше чем узлов сети, и с помощью этого ограниченного круга понятий можно определить каждый узел сети через узлы нижнего уровня, содержащие обобщенные понятия.

Преимущества такого формализма заключаются в его наглядности и непосредственной связанности понятий через сеть, которая позволяет быстро находить связи понятий и на этой основе управлять решениями.

4. Фреймы

Это частный случай семантических сетей. Это метод представления знаний, который связывает свойства с узлами, представляющими понятия и объекты. Свойства описываются атрибутами (называемыми слотами) и их значениями. Использование фреймов с их атрибутами и взаимосвязями позволяет хранить знания о предметной области в структурированном виде, представлять в БЗ абстракции и аналогии.

С операциями присваивания значений фреймам и другими операциями можно сочетать сложные побочные действия и взаимовлияния.

Используются и другие, связанные с конкретным применением способы представления, но они менее распространены и, как правило, не годятся для использования в общем случае.

Не всегда можно однозначно сказать, какой формализм представления использован в системе. В рамках одной и той же системы для представления различных видов знаний могут быть использованы различные формализмы.

ЛИТЕРАТУРА

1. *Амосов Н. М.* Нейрокомпьютеры и интеллектуальные роботы. Киев, 1991.

2. *Андерсон Дж. Р.* Учитель ЛИСПа / Дж. Р. Андерсон, Бр. Дж. Рейзер // Реальность и прогнозы искусственного интеллекта: сб. статей. М., 1987.

3. *Баррендрегт Х.* Лямбда-исчисление. Его синтаксис и семантика. М., 1985.

4. *Берджс В.* Методы рекурсивного программирования. М., 1983.

5. *Братко И.* Программирование на языке Пролог для искусственного интеллекта. М., 1990.

6. *Гаврилова Т. А.* Базы знаний интеллектуальных систем / Т. А. Гаврилова, В. Ф. Хорошевский. СПб., 2000.

7. *Гаврилова Т. А.* Извлечение и структурирование знаний для экспертных систем / Т. А. Гаврилова, К. Р. Червинская. М., 1992.

8. *Горбань А. Н.* Нейронные сети на персональном компьютере / А. Н. Горбань, Д. А. Россиев. М., 1990.

9. *Городняя Л. В.* Основы функционального программирования. М., 2004.

10. *Грэй П.* Логика, алгебра и базы данных. М., 1989.

11. *Джексон П.* Введение в экспертные системы. М., 2001.

12. *Доорс Дж.* Пролог — язык программирования будущего / Дж. Доорс, А. Р. Рейблейн, С. Вадера. М., 1990.

13. *Змитрович А. И.* Интеллектуальные информационные системы. Минск, 1997.

14. *Кирсанов Б. С.* Отечественные оболочки экспертных систем / Б. С. Кирсанов, Э. В. Попов. М., 1997.

15. *Клоксин У.* Программирование на языке Пролог / У. Клоксин, К. Меллиш. М., 1987.

16. *Крюков А. П.* Программирование на языке R-ЛИСП / А. П. Крюков [и др.]. М., 1991.

17. *Лавров С. С.* Автоматическая обработка данных. Язык ЛИСП и его реализация / С. С. Лавров, Г. С. Силагадзе. М., 1978.

18. *Левин Р.* Практическое введение в технологию искусственного интеллекта и экспертных систем с иллюстрациями на БЕЙСИКе / Р. Левин [и др.]. М., 1990.

19. Логистика: учебник / под ред. Б. А. Аникина. М., 2004.

20. *Лорьер Ж. Д.* Системы искусственного интеллекта. М., 1991.

21. *Любарский Ю. Я.* Интеллектуальные информационные системы. М., 1990.

22. *Малпас Дж.* Реляционный язык Пролог и его применение. М., 1990.

23. *Марселлус Д.* Программирование экспертных систем на Турбо-Прологе. М., 1994.

24. *Маурер У.* Введение в программирование на языке ЛИСП. М., 1976.

25. *Нейлор К.* Как построить свою экспертную систему. М., 1991.

26. *Неруш Ю.* Логистика: учебник. М., 2003.

27. *Нильсон Н.* Принципы искусственного интеллекта. М., 1985.

28. *Осипов Г. С.* Приобретение знаний интеллектуальными системами. М., 1991.

29. *Переверзев В. Н.* Логистика: справочная книга по логистике. М., 1995.

30. *Попов Э. В.* Статические и динамические экспертные системы / Э. В. Попов, И. В. Фоминых, Б. Б. Кисель. М., 1990.

31. *Поспелов Д. А.* Моделирование рассуждений. Опыт анализа мыслительных актов. М., 1989.

32. *Пратт Т.* Языки программирования: Разработка и реализация. СПб., 2002.

33. *Романов В. П.* Интеллектуальные и информационные системы в экономике: учеб. пособие. М., 2003.

34. *Сойер Б.* Программирование экспертных систем на Паскале / Б. Сойер, Д. Л. Фостер. М., 1990.

35. *Соломатин Н. М.* Информационно-семантические системы. М., 1989.

36. *Стерлинг Л.* Искусство программирования на языке Пролог / Л. Стерлинг, Э. Шапиро. М., 1990.

37. *Тельнов Ю. Ф.* Интеллектуальные информационные системы в экономике. М., 2002.

38. *Уинстон П.* Искусственный интеллект. М., 1980.

39. *Филд А.* Функциональное программирование / А. Филд, П. Харрисон. М., 1993.

40. *Хант Д.* Искусственный интеллект. М., 1986.

41. *Хендерсон П.* Функциональное программирование. Применение и реализация. М., 1983.

42. *Хоггер К.* Введение в логическое программирование: пер. с англ. М., 1988.

43. *Хювенен Э.* Мир ЛИСПа: в 2 т. / Э. Хювенен, И. Сеппянен. М., 1991. Т. 1: Введение в язык ЛИСП и функциональное программирование.

44. *Шабунин Л. В.* Комбинаторные исчисления. Чебоксары, 1984.

45. *Янсон А.* Турбо-Пролог в сжатом изложении. М., 1991.

ОГЛАВЛЕНИЕ

Введение 3

I. ВВЕДЕНИЕ В СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА 5
Понятие искусственного интеллекта 5
Традиционное программирование 5
Влияние ИИ на программирование 6
Человеческое мышление 6
Цели 6
Факты и правила 7
Упрощение 8
Механизм вывода 10
Резюме 11

II. РАЗРАБОТКА СИСТЕМ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА 12
Определение целей 12
Определение фактов 14
Получение данных 15
Правила и выводы 18
Верификация целей механизмом вывода 19
Упрощение 21

III. ОБЩИЕ СВЕДЕНИЯ ОБ ЭКСПЕРТНЫХ СИСТЕМАХ 23
Эвристические правила 24
Рабочая область 26
Разработать экспертную систему может каждый 26

IV. ОБРАБОТКА ЕСТЕСТВЕННОГО ЯЗЫКА 27
Лексический анализ 29
Синтаксический анализ 29
Семантический анализ 30

V. ПРЯМАЯ ЦЕПОЧКА РАССУЖДЕНИЙ 32
Пример прямой цепочки рассуждений 34
База знаний 36
Работа с базой знаний 36

Пример прямой цепочки рассуждений 43
Обобщенный алгоритм работы системы 48
Пример программы 49
Пояснение к программе 49
Листинг программы 49
Примеры выполнения программы 56
Пример диалога системы с пользователем 57
Вспомогательные таблицы 59

VI. ОБРАТНАЯ ЦЕПОЧКА РАССУЖДЕНИЙ 63
Разработка базы знаний: дерево решений 65
Преобразование дерева решений в правила 68
Создание правил 70
Работа с базой знаний 71
Список логических выводов 73
Список переменных 74
Список переменных условия 76
Стек логических выводов 78
Пример использования базы знаний 80
Замечания по разработке системы 89
Экспертная система 90
Пример программы 90
Пояснения к программе 91
Листинг программы 91
Примеры выполнения программы 97
Примеры диалогов с программой 99
Рабочие таблицы системы обратных рассуждений 101

VII. ВВЕДЕНИЕ В ЯЗЫК
ИСКУССТВЕННОГО ИНТЕЛЛЕКТА ПРОЛОГ 105
Лабораторная работа 1. Работа с простейшими программами
в системе Турбо-Пролог 105
1. Введение 105
2. Загрузка системы Турбо-Пролог, ввод и запуск программ ... 107
3. Работа с Пролог-программами в режиме диалога 108
4. Трассировка программ в среде системы Турбо-Пролога 109
5. Работа с программами, содержащими внутреннюю цель 110
6. Простейшая программа ввода-вывода данных 113
7. Построение простейшего интерфейса
для вывода результатов запросов 114
8. Содержание отчета по лабораторной работе 115
Лабораторная работа 2. Пролог-программы
как простейшие базы данных и знаний 115
1. Введение 115

2. Запросы к базе данных ... 116
3. Статические и динамические базы данных ... 119
4. Явные и неявные базы данных. Правила логического вывода ... 121
5. Использование структур в качестве доменов отношений ... 123
6. Процедуры как элемент представления знаний ... 126
7. Целостность и непротиворечивость баз данных и знаний ... 128
8. Содержание отчета по лабораторной работе 129
Лабораторная работа 3. Управление ходом выполнения программ в системе Турбо-Пролог ... 130
1. Работа системы Турбо-Пролог при выполнении запросов ... 130
2. Унификация термов ... 132
3. Поиск с возвратом при выполнении Пролог-программ ... 136
4. Использование отката после неудачи при использовании внутренней цели для организации простейшего интерфейса вывода ... 140
5. Содержание отчета по лабораторной работе ... 142
Лабораторная работа 4. Управление ходом выполнения Пролог-программ ... 142
1. Организация повторяющихся процессов 142
2. Управление поиском с возвратом ... 145
3. Управление ходом выполнения программ с использованием отсечения ... 147
4. Использование метода отката и отсечения ... 148
5. Откат и отсечение при реализации отношений вида «один-ко-многим» ... 153
6. Ступенчатые функции и отсечение ... 155
7. Содержание отчета по лабораторной работе ... 156

VIII. ВВЕДЕНИЕ В ЯЗЫК ИСКУССТВЕННОГО ИНТЕЛЛЕКТА ЛИСП ... 157
Лабораторная работа 1. Работа со списками 159
Лабораторная работа 2. Сопоставление с образцом ... 178
Лабораторная работа 3. Изучение интерпретатора языка Пролог 183

IX. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ЯЗЫКА ПРОГРАММИРОВАНИЯ ЛИСП ... 197
1. Основные особенности языка ЛИСП ... 197
1.1. Одинаковая форма данных и программ ... 197
1.2. Хранение данных, не зависящее от места ... 198
1.3. Автоматическое и динамическое управление памятью ... 198
1.4. Функциональная направленность ЛИСПа ... 199
1.5. ЛИСП — бестиповый язык ... 199

1.6. Интерпретирующий и компилирующий режимы работы ... 200
1.7. Пошаговое программирование ... 200
1.8. Единый системный и прикладной языки программирования ... 200
2. Понятия языка ЛИСП ... 201
2.1. Атомы и списки ... 201
2.2. Внутреннее представление списка ... 202
2.3. Написание программы на ЛИСПе ... 203
2.4. Определение функций ... 204
2.5. Рекурсия и итерация ... 206
2.6. Функции интерпретации выражения ... 209
2.7. Макросредства ... 209
2.8. Функции ввода-вывода ... 211
3. Знания в ИИ ... 212
3.1. Требования к знаниям ... 212
3.2. Основные типы знаний ... 212
3.3. Методы представления знаний ... 213

ЛИТЕРАТУРА ... 217

Учебное издание

ПУТЬКИНА Лидия Владимировна
ПИСКУНОВА Татьяна Григорьевна

Интеллектуальные информационные системы

Учебное пособие

Ответственный редактор *И. В. Петрова*
Дизайнер *В. Б. Клоков*
Технический редактор *Е. Ю. Николаева*
Корректор *В. Ю. Нечаева*

ISBN 978-5-7621-0425-8

Подписано в печать с оригинал-макета 21.02.08. Формат 60x90/16
Гарнитура Times New Roman. Усл. печ. л. 14,25. Тираж 1000 экз. Заказ № 26
Издательство Санкт-Петербургского Гуманитарного
университета профсоюзов
192238, Санкт-Петербург, ул. Фучика, 15
Отпечатано в типографии СПбГУП

ПУТЬКИНА Лидия Владимировна — заместитель заведующего кафедрой информатики по научной работе СПбГУП, кандидат технических наук, доцент.

Сфера научных интересов связана с исследованиями в области искусственного интеллекта, с автоматизацией экономических процессов на основе применения современных информационных технологий, с разработкой методов и средств моделирования информационных систем, с реинжинирингом бизнес-процессов.

Автор более 50 публикаций по основным проблемам реинжиниринга бизнес-процессов на предприятиях социально-культурной сферы, автоматического управления экономическими системами.

ПИСКУНОВА Татьяна Григорьевна — исполняющая обязанности заведующего кафедрой информатики СПбГУП, кандидат педагогических наук, доцент.

Сфера научных интересов связана с исследованиями в области искусственного интеллекта, с технологией использования обучающих мультимедийных комплексов в высшей профессиональной школе, с разработкой методов и средств моделирования информационных систем.

Автор более 50 публикаций по основным проблемам обучающих мультимедийных комплексов в профессиональной высшей школе.

Книга освещает современное состояние интеллектуальных информационных систем, применяемых в экономике и социокультурной сфере.

В учебном пособии рассматриваются практические примеры и их реализация на языках программирования Бейсик, Лисп и Пролог. Книга содержит введение в теорию искусственного интеллекта, средства решения задач искусственного интеллекта и язык теории предикатов, а также общие сведения об экспертных системах.

Издание предназначено для студентов, обучающихся по специальности «Прикладная информатика (по областям)», для аспирантов, преподавателей и специалистов в области искусственного интеллекта.