МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)» (СГАУ)

Системы искусственного интеллекта

Электронный учебно-методический комплекс
по дисциплине в LMS Moodle

САМАРА
2012

УДК 681.3.07

Автор-составитель: **Лёзина Ирина Викторовна**

Системы искусственного интеллекта [Электронный ресурс] : электрон. учеб.-метод. комплекс по дисциплине в LMS Moodle / Минобрнауки России, Самар. гос. аэрокосм. ун-т им. С. П. Королева (нац. исслед. ун-т); авт.-сост. И. В. Лёзина. - Электрон. текстовые и граф. дан. - Самара, 2012. – 1 эл. опт. диск (CD-ROM).

В состав учебно-методического комплекса входят:
1. Курс лекций.
2. Учебное пособие.
3. Методические указания к лабораторным работам.
4. Вопросы к экзамену.
5. Тесты для контроля знаний.
6. Примеры заданий для проведения коллоквиумов.
7. Рабочая программа курса.

УМКД «Системы искусственного интеллекта» предназначен для студентов факультета информатики, обучающихся по направлению подготовки бакалавров 230100.62 «Информатика и вычислительная техника» в 8 семестре.

УМКД разработан на кафедре информационных систем и технологий.

Курс лекций по дисциплине «Системы искусственного интеллекта».

1 Основные направления искусственного интеллекта.

1.1 История развития искусственного интеллекта

Термин искусственный интеллект (ИИ) является русским переводом английского термина artifical intelligence . Создателем ИИ многие ученые считают Алана Тьюринга, автора знаменитой машины Тьюринга, которая стала одним из математических определений алгоритма [1]. В 1950 году в английском журнале “ Mind” в статье “Computing Machinery and Intelligence” (в русском переводе статья называлась «Может ли машина мыслить?») Алан Тьюринг предложил критерий, позволяющий определить, обладает ли машина мыслительными способностями. Этот тест заключается в следующем: человек и машина при помощи записок ведут диалог, а судья (человек), находясь в другом месте, должен определить по запискам, кому они принадлежат, человеку или машине. Если ему это не удается, то это будет означать, что машина успешно прошла тест. До сих пор не одна машина такой тест не прошла.

Не существует единого и общепринятого определения ИИ. Это не удивительно, так как нет универсального определения человеческого интеллекта.

ИИ – это область информатики, предметом которой является разработка компьютерных систем, обладающих возможностями, традиционно связываемыми со способностями естественного интеллекта.

К области ИИ принято относить ряд алгоритмов и программных систем, которые могут решать некоторые задачи так, как это делает человек.

Первый шаг в исследованиях по ИИ был сделан в направлении изучения естественного интеллекта. При изучении этого вопроса был сделан ряд открытий в различных областях знаний. Так, в 1962 году Фрэнком Розенблаттом были предложены модели мозга, имитирующие биофизические процессы, которые протекают в головном мозге и которые были названы *персептронами*. Персептроны представляют собой различного вида сети из искусственных нейронов, в основе которых лежат модели, разработанные еще в 1943 году Уильямом Маккалоком и Уолтером Питтсом.

Первоначально, изучение персептронов было связано с задачей распознавания образов, однако, в настоящее время *нейронные сети* широко используются для решения задач аппроксимации, классификации и распознавания образов, прогнозирования, идентификации и оценивания, ассоциативного управления [5]. Нейронные сети представляют собой низкоуровневые модели мозговой деятельности человека.

Другое направление моделирования естественного интеллекта связано с созданием высокоуровневых моделей деятельности мозга человека,

которые позволяют моделировать процессы рассуждений и принятия решений.

В целом можно сказать, что изучение разумного поведения человека привело к появлению эвристических методов, моделирующих деятельность человека в проблемной ситуации и к разработке программно-аппаратных средств, реализующих эти методы, то есть к разработке систем искусственного интеллекта, называемых *решателями задач*.

Другим результатом этих исследований можно считать создание *экспертных систем*, то есть систем искусственного интеллекта, основанных на знаниях человека-эксперта.

Также к специфическим особенностям деятельности человека обычно относят способности к распознаванию сложных зрительных и слуховых образов, пониманию естественных языков, способности к обучению, рассуждениям и логическим выводам. Все эти особенности стали реализовываться в системах искусственного интеллекта.

В Советском союзе ИИ получил официальное признание в 1974 году, когда при президиуме АН СССР был создан научный совет по проблеме «Искусственный интеллект», хотя работы в этом направлении велись с 60-х годов Вениамином Пушкиным, Дмитрием Александровичем Поспеловым, Сергеем Масловым, В.Ф.Турчиным.

Первые положительные результаты были получены в области теории управления, так как в этой области имелся ряд задач, для решения которых традиционные методы не были пригодны из-за невозможности формализации цели управления объектом и невозможности установления точных количественных зависимостей между параметрами, оказывающими влияние на процесс управления [1]. В результате проведенных работ появились *логико-лингвистические модели,* в которых решающее значение имеют тексты на естественном языке. В таких моделях для принятия решения при управлении объектами используется семантическая информация для описания модели объекта, модели среды и блока принятия решения.

Моделирование рассуждений человека, осуществление логического вывода с помощью вычислительной машины стало возможным, благодаря использованию методов поиска решений в исчислении предикатов [3]. Эти методы стали основой общей теории дедуктивных систем. При этом все «творческие задачи» решаются интеллектуальным перебором в четко очерченном множестве – в фиксированной формальной теории, которая является ветвью математической логики и в которой реализуется процесс нахождения решений.

В настоящее время выделяют следующие направления развития исследований в области искусственного интеллекта [2]:

1. Разработка систем, основанных на знаниях. Целью этого направления является имитация способностей человека в области анализа неструктурированных и слабоструктурированных задач. В данной области исследований осуществляется разработка моделей представления,

извлечения и структурирования знаний, а также изучаются проблемы создания *баз знаний* (БЗ). К данному классу систем также относятся *экспертные системы* (ЭС).

2. Разработка естественно-языковых интерфейсов и машинный перевод. Данные системы строятся как интеллектуальные системы, так как основаны на БЗ в определенной предметной области и сложных моделях, обеспечивающих трансляцию «исходный язык – язык смысла – язык перевода». Эти модели основаны на последовательном анализе и синтезе естественно-языковых сообщений и ассоциативном поиске аналогичных фрагментов текста и их переводов в специальных *базах данных* (БД).

3. Генерация и распознавание речи. Решаются задачи обработки, анализа и синтеза фонемных текстов.

4. Обработка визуальной информации. Решаются задачи обработки, анализа и синтеза изображений. В задаче анализа исходные изображения преобразуются в данные другого типа, например, текстовые описания. При синтезе изображений в качестве входной информации используются алгоритмы построения изображений, а выходными данными являются графические объекты.

5. Обучение и самообучение. Данная область ИИ включает модели, методы и алгоритмы, реализующие автоматическое накопление и генерацию знаний с использованием процедур анализа и обобщения знаний. К данному направлению относятся системы *добычи данных (Data-mining)* и системы *поиска закономерностей в компьютерных базах данных (Knowledge Discovery)*.

6. Распознавание образов. Распознавание образов осуществляется на применении специальных математических моделей, обеспечивающих отнесение объектов к классам, которые описываются совокупностями определенных значений признаков.

7. Игры и машинное творчество. К данной области относятся системы сочинения компьютерной музыки, стихов, изобретения новых объектов, а также интеллектуальные компьютерные игры.

8. Программное обеспечение систем ИИ. К данной области относятся инструментальные средства для разработки интеллектуальных систем, включая специальные языки программирования, ориентирование на обработку символьной информации (LISP, SMALLTALK, РЕФАЛ), языки логического программирования (PROLOG), языки представления знаний (OPS 5, KRL, FRL), интегрирование программные среды (KE, ARTS, GURU, G2), а также оболочки экспертных систем (BUILD, EMYGIN, EXSYS Professional, ЭКСПЕРТ).

9. Новые архитектуры компьютеров. Это направление связано с созданием компьютеров не фон-неймановской архитектуры, ориентированных на обработку символьной информации. Известны удачные промышленные решения параллельных и векторных компьютеров, однако в настоящее время они имеют очень высокую стоимость и недостаточную совместимость с существующими вычислительными средствами.

10.Интеллектуальные роботы. В настоящее время данная область ИИ развивается очень бурно. Достигнуты значительные успехи в создании бытовых роботов, роботов, используемых в космических исследованиях, медицинских роботов.

1.2 Современное состояние искусственного интеллекта.

В настоящее время в области ИИ активно работают военные ведомства и ведущие западные фирмы, такие как AT&T, Intel, General Electric, Sharp, Hitachi, Siemens.

Военное научное агенство DARPA - крупнейший в мире финансист исследований по ИИ, особенно по *робототехнике* [1]. Создание современного оружия немыслимо без использования методов ИИ, особенно таких, как нейронные технологии, нечеткие экспертные системы и интеллектуальные решатели задач. Эти методы позволяют с помощью относительно малых ресурсов получать достаточно точные результаты. В этой связи состояние разработок в некоторых областях ИИ закрыто для широкого доступа.

С другой стороны, в настоящее время бурно развивается рынок бытовых роботов и интеллектуальных домашних устройств, которые приносят немалую прибыль фирмам-разработчикам. Так, например, компания NEC представила модель робота Personal Robot R100, которая может передвигаться, произносить 300 фраз, понимать сотни команд и различать 10 лиц. Робот может приносить мелкие вещи, вынимать почту из ящика, включать и выключать телевизор, записывать видеосообщения и передавать их по назначению.

Ведутся активные работы в области разработки и производства роботов, предназначенных для спасения людей в завалах, высадки на других планетах и астероидах и даже для проведения хирургических операций в полевых условиях. Похожие работы проводятся в российском научном центре сердечно-сосудистой хирургии имени Бакулева РАМН. Используемый там робот имеет несколько манипуляторов, способных держать различные инструменты. Он может работать в самых неудобных и недоступных для человека положениях. Врач за монитором следит за зоной операции и управляет манипуляторами, подавая через компьютер голосовые команды [1].

Группа ученых из Цюрихского института нейроинформатики и Манчестерского технологического института утверждает, что им удалось создать технологию, полностью повторяющую механизм функционирования человеческого мозга, который одновременно обрабатывает цифровую и аналоговую информацию. Для этого используется способность нейронных сетей к распространению важных сигналов и подавлению слабых сигналов. Аналогичные роботы были разработаны академиком Н. Амосовым 20 лет назад.

Вопросы создания кибернетических устройств, способных выполнять присущие человеку действия, все больше привлекают разработчиков.

Современный подход опирается на *теории адаптивных систем и эволюционного развития.* В соответствии с данным подходом предполагается, что устройства управления должны самостоятельно мутировать и развиваться, менять свою форму, размеры и так далее.

Так, например, DARPA финансирует проект создания системы сборки конструкций из кубиков Лего. Система состоит из манипулятора, видеокамеры и компьютера. В качестве исходных данных в систему заложены только элементарные правила стыковки кубиков и цель – конечное сооружение, после чего система начинает пробовать различные комбинации, экспериментально определяя прочность и стабильность собираемых конструкций. Пока такая система способна за один день собрать двухметровый игрушечный мост и кран, способный поднять груз 0,5 кг. Самое главное, что эти конструкции отвечают всем инженерным требованиям по надежности, о которых система и не подозревает. Следующая задача – автоматизировать сборку системой себе подобных роботов.

В настоящее время также развивается такая область робототехники, как создание искусственного сознания, которое называют *чат-роботом*. Крис Мак-Кинли разработал искусственное сознание по имени GAC (Generic Artifical Consclousness), которое в сети Интернет может вести беседу, отвечая на вопросы «да» или «нет». Создатель GAC рассчитывает в ближайшие 10 лет собрать для него 1 миллиард фактов, что должно сделать искусственное сознание, не отличимым по уровню интеллекта от среднего человека.

Получило дальнейшее развитие такое традиционное направление ИИ как экспертные системы (ЭС). В современных ЭС, основной акцент делается на принятие оперативных решений в реальном масштабе времени. Это объясняется потребностями современного бизнеса. Коммерческие ЭС контролируют крупные промышленные процессы, управляют большими сетями, распределенными СУБД, подсказывая оператору, как поступить в сложной обстановке, а в критических ситуациях берут управление на себя.

Достаточно активно развивается и такое направление ИИ, как *автоматическое накопление знаний*, реализующее качественный анализ различных процессов. В коммерческих продуктах применяются такие подходы, как нечеткие технологии, основанные на использовании логики с бесконечным числом состояний. Так, интеллектуальный решатель C-PRS, написанный в стандарте ANSI C, используется NASA, в авиапромышленности, в системах управления перевозками и мобильными роботами.

В России осуществлено несколько внедрений ЭС Gensym G2, которая осуществляет контроль, управление и моделирование сложных процессов.

Общение компьютера с человеком на естественном языке - одна из первых задач ИИ, также получила развитие в настоящее время. Еще 30 лет назад была написана программа, которая выделяла во фразах ключевые слова, строила на их основе простые вопросы, создавая неплохую иллюзию реального диалога.

Современные системы обычно строятся по тому же принципу, однако есть и исключения. Так, например, компания Artifical Life выпускает набор программных продуктов, использующих технологию *автономных агентов*. Сервер приложений Klone Server поддерживает работу автономных агентов (интерактивных персонажей), которые общаются с потребителями сайта на естественном языке. Пока они способны отвечать на 20% всех вопросов. Агенты могут решать различные проблемы общения, основываясь на системе логического вывода, имеющей несколько уровней синтаксического анализа фраз, слежения за контекстом разговора и понимания отдельных слов. Другие модули выполняют автоматический анализ и обработку поступающей по электронной почте информации, создают пользовательские профили, определяют и анализируют траектории путешествий людей по сайтам и так далее.

1.3 Классификация систем искусственного интеллекта.

Для систем искусственного интеллекта характерны следующие признаки [2]:

- развитые коммуникативные способности;
- умение решать сложные задачи;
- способность к самообучению;
- адаптивность.

В соответствии с данными признаками, системы искусственного интеллекта, можно разделить на классы, представленные на рис.1.1 [2].

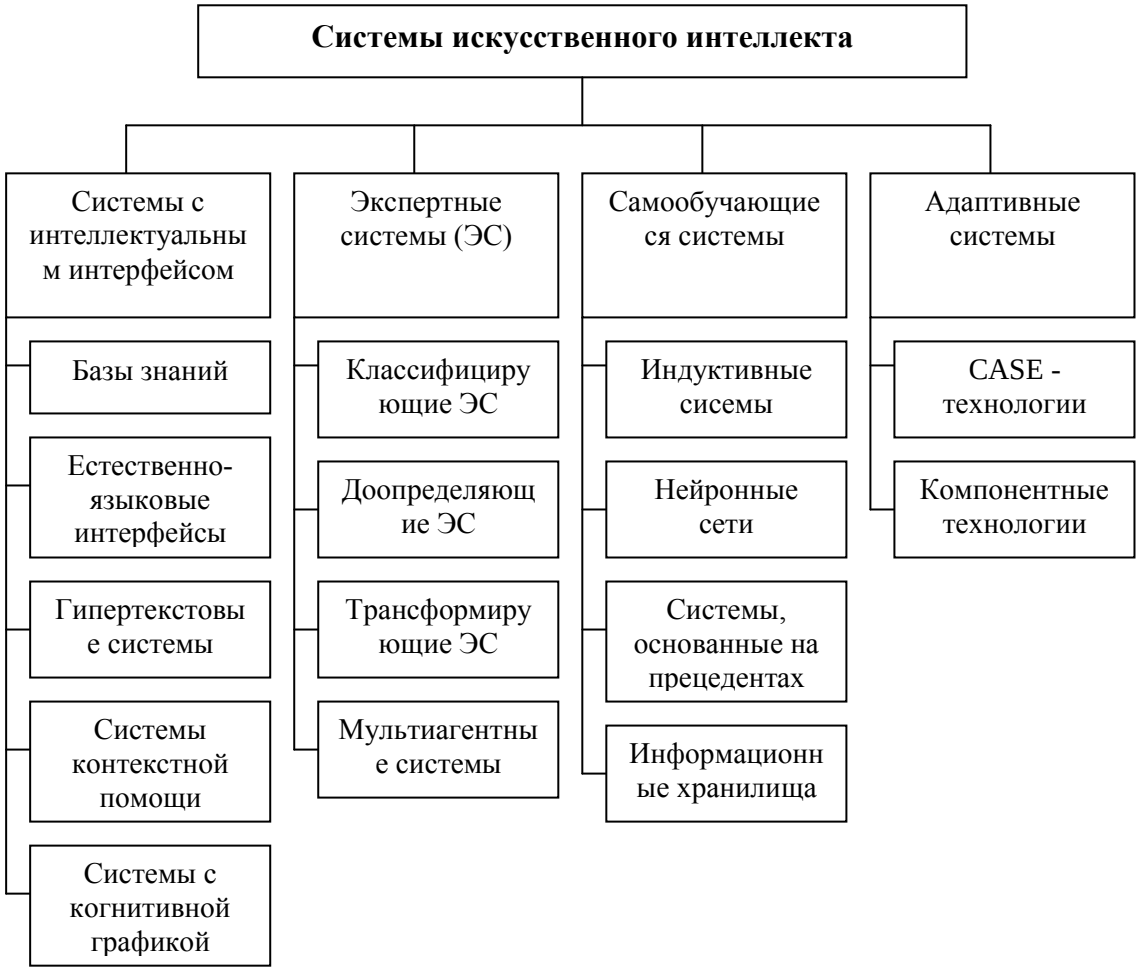


Рис.1.1 Классификация систем искусственного интеллекта

1.3.1 Системы с интеллектуальным интерфейсом

Базы знаний (БЗ) позволяют в отличии от традиционных баз данных (БД) обеспечивать выборку необходимой информации, не хранимой явно, а выводимой из совокупности хранимых данных.

Естественно-языковые интерфейсы применяются для доступа к БЗ, контекстного поиска текстовой информации, голосового ввода команд, машинного перевода с иностранных языков.

Гипертекстовые системы используются для реализации поиска по ключевым словам в БД с текстовой информацией.

Системы контекстной помощи являются частным случаем гипертекстовых систем и естественно-языковых систем. В отличие от них пользователь сам описывает проблему, а система выполняет поиск относящихся к ситуации рекомендаций.

Системы когнитивной графики ориентированы на общение с пользователем посредством графических образов, которые генерируются в соответствии с изменениями параметров моделируемых или наблюдаемых процессов.

1.3.2 Экспертные системы

Область исследования ЭС называется *инженерией знаний*. Экспертные системы предназначены для решения неформализованных задач, то есть задач, решаемых с помощью неточных знаний, которые являются результатом обобщения многолетнего опыта работы и интуиции специалистов. Неформализованные знания обычно представляют собой эвристические приемы и правила. ЭС обладают следующими особенностями:

- алгоритм решения не известен заранее, а строится самой ЭС с помощью символических рассуждений, базирующихся на эвристических приемах;
- ясность полученных решений, то есть система «осознает» в терминах пользователя, как она получает решение;
- способность анализа и объяснения своих действий и знаний;
- способность приобретения новых знаний от пользователя-эксперта, не знающего программирования, и изменения в соответствии с ними своего поведения;
- обеспечение «дружественного», как правило, естественно-языкового интерфейса с пользователем.

ЭС охватывают самые разные предметные области, среди которых преобладают медицина, бизнес, производство, проектирование и системы управления.

Классифицирующие ЭС решают задачи распознавания ситуаций. Основным методом формирования решений в них является дедуктивный логический вывод.

Доопределяющие ЭС используются для решения задач с не полностью определенными данными и знаниями. В качестве методов обработки неопределенных знаний могут использоваться байесовский вероятностный подход, коэффициенты уверенности, нечеткая логика.

Трансформирующие ЭС реализуют преобразование знаний в процессе решения задачи. В ЭС данного класса используются различные способы обработки знаний:

- генерация и проверка гипотез;
- логика предположений и умолчаний;
- использование метазнаний для устранения неопределенности.

Мультиагентные системы – это динамические ЭС, основанные на интеграции разнородных источников знаний, которые обмениваются между собой полученными результатами в процессе решения задач. Системы данного класса имеют следующие возможности:

- реализация альтернативных рассуждений;
- распределённое решение проблем, разделяемое на параллельно решаемые подзадачи;
- применение различных стратегий вывода заключений;
- обработка больших массивов информации из БД;
- использование математических моделей и внешних процедур для имитации развития ситуаций.

1.3.3 Самообучающиеся системы

Системы данного класса основаны на методах автоматической классификации ситуаций из реальной практики, или на методах обучения на примерах. Примеры составляют так называемую обучающую выборку. Элементы обучающей выборки описываются множеством классификационных признаков.

Стратегия обучения *«с учителем»* предполагает задание для каждого примера эталонных значений признаков, показывающих его принадлежность к определенному классу. При обучении *«без учителя»* система должна самостоятельно выделять классы ситуаций по степени близости значений классификационных признаков.

В процессе обучения проводится автоматическое построение обобщающих правил или функций, описывающих принадлежность ситуации к классам, которыми система будет впоследствии пользоваться при определении незнакомых ситуаций. При этом из обобщающих правил автоматически формируется БЗ, которая периодически корректируется.

Индуктивные системы позволяют обобщать примеры на основе принципа индукции «от частного к общему». Процедура обобщения сводится к классификации примеров по значимым признакам.

Нейронные сети – обобщенное название группы математических моделей и алгоритмов, обладающих способностью обучаться на примерах,

«узнавая» впоследствии черты встреченных образцов и ситуаций. Нейронные сети используются для решения задач аппроксимации и идентификации функций, классификации и распознавания образов, обработки сигналов, сжатия данных, прогнозирования и адаптивного управления.

Нейронная сеть – это кибернетическая модель нервной системы, которая представляет собой совокупность большого числа нейронов, топология соединения которых зависит от типа сети. Нейроны соединяются в слои. Различают сети прямого распространения и рекурентные сети (с обратными связями). Чтобы создать нейронную сеть для решения какой-либо задачи, необходимо выбрать тип сети и определить параметры сети в процессе ее обучения.

В системах, основных на прецедентах, БЗ содержит описания конкретных ситуаций (прецеденты). Поиск решений осуществляется на основе аналогий по значениям соответствующих признаков. В отличии от индуктивных систем допускается нечеткий поиск с получением множества допустимых альтернатив, каждая из которых может оцениваться некоторым коэффициентом уверенности.

Информационные хранилища отличаются от БЗ. Хранилище данных – это предметно-ориентированное, интегрированное, привязанное ко времени, неизменяемое собрание данных, применяемых для поддержки процессов принятия управленческих решений.

Технологии извлечения знаний из хранилища данных основаны на методах статистического анализа и моделирования, ориентированных на поиск моделей и отношений, скрытых в совокупности данных.

Для извлечения значимой информации используются специальные методы(OLAP – анализ, DATA Mining или Knowledge Discovery), основанные на применении методов математической статистики, нейронных сетей, индуктивных методов и других методах.

1.3.4 Адаптивные системы

Адаптивные системы должны удовлетворять ряду требований:

- адекватно отражать знания проблемной области в каждый момент времени;
- быть пригодными для легкой и быстрой реконструкции при изменениях проблемной среды.

Ядром систем данного класса является модель проблемной области, поддерживаемая в специальной БЗ – *репрозитории.* Ядро системы управляет процессами генерации или переконфигурирования программного обеспечения. При разработке адаптивных систем используется типовое или оригинальное проектирование.

Реализация оригинального проектирования основана на использовании *CASE- технологий* (Designer2000, SilverRun, Natural Light Strom и др.).

При типовом проектировании осуществляется адаптация типовых разработок к особенностям проблемной области. При этом используются инструментальные средства *компонентного (сборочного) проектирования* (R/3, BAAN, Prodis и др.).

При использовании CASE- технологий при изменении проблемной области каждый раз применяется генерация программного обеспечения, а при использовании сборочной технологии – конфигурирование программ или их переработка.

1.4 Характеристики знаний.

Основной проблемой, решаемой во всех системах ИИ, является проблема *представления знаний*. Информация представляется в компьютере в процедурной и декларативной форме. В процедурной форме представлены программы, в декларативной – данные. В системах искусственного интеллекта возникла концепция новой формы представления информации – знания, которая объединила в себе черты как процедурной, так и декларативной информации. Перечислим основные характеристики знаний:

1. *Внутренняя интерпретируемость.* Каждая информационная единица должна иметь уникальное имя, по которому система находит ее, а также отвечает на запросы, в которых это имя упомянуто. Данные в памяти лишены имен и могут идентифицироваться только программой, извлекающей их из памяти. При переходе к знаниям в память вводится информация о некоторой *протоструктуре информационных единиц и словари имен данных.* Каждая единица информации будет экземпляром протоструктуры. СУБД обеспечивают реализацию внутренней интерпретируемости информации, хранимой в базе данных.
2. *Структурированность.* Информационные единицы должны соответствовать «принципу матрешки», то есть рекурсивной вложенности одних информационных единиц в другие. Другими словами, должна существовать возможность произвольного установления между отдельными информационными единицами отношений типа «часть – целое», «род – вид» или «элемент – класс».
3. *Связность.* Между информационными единицами должна быть предусмотрена возможность установления связей различного типа, характеризующих отношения между информационными единицами. Семантика отношений может носить как декларативный характер, например, в отношениях «одновременно» и «причина – следствие», так и процедурный характер, например, в отношении «аргумент – функция». Все отношения можно разделить на четыре категории: *отношения структуризации* (задают иерархию информационных единиц), *функциональные отношения* (несут процедурную информацию, позволяющую вычислять одни информационные единицы через другие), *каузальные отношения* (задают причинно-следственные связи) и *семантические отношения* (все остальные отношения).
4. *Семантическая метрика.* Между информационными единицами задают *отношения релевантности*, которые характеризуют ситуационную близость информационных единиц, то есть силу

ассоциативной связи между информационными единицами (например, «покупка» или «регулирование движения на перекрестке»). Отношение релевантности позволяет находить знания, близкие к найденным ранее знаниям.

5. *Активность.* Выполнение программ в информационных системах должно инициализироваться не командами, а состоянием информационной базы, например, появлением в базе фактов или описаний событий или установление связей между информационными единицами.

Перечисленные характеристики определяют разницу между *данными* и *знаниями,* при этом *базы* данных перерастают в *базы знаний.*

1.5 Модели представления знаний.

В интеллектуальных системах используются четыре основных типа моделей знаний:

1. *Логические модели.* В основе моделей такого типа лежит *формальная система,* задаваемая четверкой вида $M = <T, S, A, B>$. Множество *T* есть множество *базовых элементов*, например слов из некоторого словаря, или деталей из некоторого набора. Для множества *T* существует некоторый способ определения принадлежности или непринадлежности произвольного элемента к данному множеству. Процедура такой проверки может быть любой, но она должна давать ответ на вопрос, является ли *x* элементом множества *T* за конечное число шагов. Обозначим эту процедуру *P(T).*

Множество *S* есть множество *синтаксических правил*. С их помощью из элементов *T* образуют синтаксически правильные совокупности. Например, из слов словаря строятся синтаксически правильные фразы, а из деталей собираются конструкции. Существует некоторая процедура *P(S)*, с помощью которой за конечное число шагов можно получить ответ на вопрос, является ли совокупность *X* синтаксически правильной.

Во множестве синтаксически правильных совокупностей выделяется некоторое подмножество *A*. Элементы *A* называются *аксиомами*. Как и для других составляющих формальной системы, должна существовать процедура *P(A)* , с помощью которой для любой синтаксически правильной совокупности можно получить ответ на вопрос о принадлежности ее к множеству *A.*

Множество *B* есть множество *правил вывода*. Применяя их к элементам *A*, можно получать новые синтаксически правильные совокупности, к которым снова можно применять правила из *B*. Так формируется *множество выводимых* в данной формальной системе *совокупностей.* Если имеется процедура *P(B)*, с помощью которой можно определить для любой синтаксически правильной совокупности, является ли она выводимой, то соответствующая формальная система называется

разрешимой. Это показывает, что именно правила вывода являются наиболее сложной составляющей формальной системы.

Для знаний, входящих в базу знаний, можно считать, что множество *A* образуют все информационные единицы, которые введены в базу знаний извне, а с помощью правил вывода из них выводятся новые *производные знания*. Другими словами, формальная система представляет собой генератор порождения новых знаний, образующих множество *выводимых* в данной системе *знаний*. Это свойство логических моделей позволяет хранить в базе лишь те знания, которые образуют множество *A*, а все остальные знания получать из них по правилам вывода.

2. *Сетевые модели.* Сетевые модели формально можно описать в виде $H = <I, C_1, C_2, \ldots, C_n, G)$. Здесь *I* есть множество информационных единиц; $C_1, C_2, \ldots, C_n$ - множество типов связей между ними. Отображение *G* задает связи из заданного набора типов связей между информационными единицами, входящими в *I*.

В зависимости от типов связей, используемых в модели, различают *классифицирующие сети*, *функциональные сети* и *сценарии, нейронные сети.* В классифицирующих сетях используются отношения структуризации. Такие сети позволяют в базах вводить иерархические отношения между информационными единицами. Функциональные сети характеризуются наличием функциональных отношений. Их часто называют *вычислительными моделями*, так как они позволяют описывать процедуры «вычислений» одних информационных единиц через другие. Нейронные сети можно отнести к классу функциональных сетей, однако нечёткие продукционные нейронные сети представляют собой гибридную модель, соединяющую в себе черты логической, продукционной и сетевой моделей. В сценариях используются каузальные отношения, а также отношения типа «средство – результат». Если в сетевой модели допускаются связи различного типа, то ее называют *семантической сетью*.

3.Продукционные модели. В моделях этого типа используются некоторые элементы логических и сетевых моделей. Из логических моделей заимствована идея правил вывода, которые здесь называются продукциями, а из сетевых моделей – описание знаний в виде семантической сети. В результате применения правил вывода к фрагментам сетевого описания происходит трансформация семантической сети за счет смены ее фрагментов, наращивания сети и исключения из нее ненужных фрагментов. Таким образом, в продукционных моделях процедурная информация явно выделена и описывается иными средствами, чем декларативная информация. Вместо логического вывода, характерного для логических моделей, в продукционных моделях появляется *вывод на знаниях*.

4.Фреймовые модели. В отличие от моделей других типов во фреймовых моделях фиксируется жесткая структура информационных

единиц, которая называется *протофреймом.* В общем виде она выглядит следующим образом:

(Имя фрейма:

Имя слота 1 (значение слота 1)
Имя слота 2 (значение слота 2)
. . . . . .
Имя слота К (значение слота К)).

Значением *слота* может быть все, что угодно: числа, математические соотношения, тексты на естественном языке, программы, правила вывода, ссылки на другие слоты данного фрейма или других фреймов. В качестве значения слота может выступать набор слотов более низкого уровня, что позволяет реализовать во фреймовых представлениях «принцип матрешки».

При конкретизации фрейма ему и слотам присваиваются имена, и происходит заполнение слотов. Таким образом, из протофреймов получаются *фреймы-экземпляры*. Переход от исходного протофрейма к фрейму-экземпляру может быть многошаговым, за счет постепенного уточнения значений слотов. Связи между фреймами задаются значениями специального слота с именем «Связь». Некоторые специалисты по ИС не выделяют фреймовые модели в отдельный класс, так как в ней объединены все основные особенности моделей остальных типов.

2 Логическое программирование и аксиоматические системы.

2.1 Общие положения

Теория формальных систем и, в частности, математическая логика являются формализацией человеческого мышления и представления наших знаний. Если предположить, что можно аксиоматизировать наши знания и можно построить алгоритм, позволяющий реализовать процесс вывода ответов на запрос из знаний, то в результате можно получить формальный метод для получения неформальных результатов.

Логическое программирование возникло в эру ЭВМ как естественное желание автоматизировать процесс логического вывода, поэтому оно является ветвью теории формальных систем.

Логическое программирование (в широком смысле) представляет собой семейство таких методов решения задач, в которых используются приемы логического вывода для манипулирования знаниями, представленными в декларативной форме[1]. Как писал Джордж Робинсон в 1984 году, в основе идеи логического программирования лежит описание задачи совокупностью утверждений на некотором формальном логическом языке и получение решения с помощью вывода в некоторой формальной (аксиоматической) системе. Такой аксиоматической системой являются исчисление предикатов первого порядка, поэтому в узком смысле логическое

программирование понимается как использование исчисления предикатов первого порядка в качестве основы для описания предметной области и осуществления резолюционного логического вывода.

Аксиоматической системой называется способ задания множества путем указания исходных элементов (аксиом исчисления) и правил вывода, каждое из которых описывает, как строить новые элементы из исходных элементов.

Под *аксиоматическим методом* [7] понимают способ построения научной теории, при которой за ее основу берется ряд основополагающих, не требующих доказательств положений этой теории, называемыми аксиомами или постулатами.

Аксиоматический метод зародился в работах древнегреческих геометров. Вплоть до начала XIX века единственным образцом применения этого метода была геометрия Евклида.

В начале XIX века Н.И.Лобачевский и Я.Больяй, независимо друг от друга, открыли новую неевклидову геометрию, заменив пятый постулат о параллельных прямых на его отрицание. Их открытие стало отправной точкой для развития аксиоматического метода, который лег в основу теории формальных систем.

С накоплением опыта построения формальных теорий и попытками аксиоматизации арифметики, предпринятыми Дж. Пеано, возникла *теория доказательств*. Теория доказательств – это раздел современной математической логики и предшественница логического программирования.

Исчислениями называют наиболее важные из аксиоматических логических систем – исчисление высказываний и исчисление предикатов.

Формальная теория строится как четко определенный класс выражений, формул, в котором некоторым точным способом выделяется подкласс теорем данной формальной системы. При этом формулы формальной системы непосредственно не несут в себе никакого содержательного смысла, они строятся из произвольных знаков или символов, исходя лишь из соображений удобства.

2.2 Исчисление высказываний.

2.2.1 Понятие высказывания

Высказывание есть утвердительное предложение, которое либо истинно, либо ложно, но не то и другое вместе. «Истина» или «ложь», приписанная некоторому высказыванию, называется истинностным значением этого высказывания. Причины истинности или ложности высказываний бывают разными. Рассмотрим три истинных высказывания:

- Земля вертится;
- За день до своей смерти он был еще жив;
- Если верно, что когда идет дождь, то дорога мокрая, то справедливо также и следующее утверждение: если дорога сухая, то дождя нет.

Первое предложение выражает некоторый факт из физики и астрономии и является *фактической истиной*.

Чтобы убедиться в правильности второго предложения, достаточно понимать смысл слов: это предложение является *истиной языка*.

Чтобы принять третье утверждение, достаточно понимать смысл некоторых слов (если…то, нет), а также знать, что части фразы «идет дождь» и «дорога мокрая» являются *высказываниями*, которые могут быть истинными или ложными, однако все предложение останется истинным, если заменить эти два высказывания другими. Если эти высказывания заменить переменными *x* и *y*, то предложение «если из *x* следует *y*, то из не *y* следует не *x*». Такие истины называются *логическими истинами* независимо от смысла (интерпретации) *x* и *y*.

2.2.2 Алфавит исчисления высказываний

Исчисление высказываний – формальная логическая система. Множество ее базовых элементов составляют логический словарь (алфавит) *T* из бесконечного счетного множества высказываний, обозначаемых строчными латинскими буквами (иногда с индексами) и называемых *атомами* и пяти элементарных *логических функций* (*связок*):

«отрицание» -¬, ~, -, not, не;

«конъюнкция» - ∧, &, and, и;

«дизъюнкция» - ∨, |, or, или;

«импликация» - →, ⊃, ⇒;

«эквивалентность» - ↔, ≡, ⇔.

Приведем таблицы истинности для перечисленных выше логических функций.

Операция отрицания является унарной операцией, обозначается ¬*x* и означает противоположное *x* значение.

Таблица 1. Таблица истинности для отрицания.

x	¬x
И	Л
Л	И

Конъюнкция является бинарной операцией, обозначается *x* ∧ *y*, читается как « *x и y*» и является истинной тогда и только тогда, когда истинны обе переменные.

Таблица 2. Таблица истинности для конъюнкции.

x	y	x ∧ y
И	И	И
И	Л	Л
Л	И	Л

Л	Л	Л

Дизъюнкция является бинарной операцией, обозначается $x \vee y$, читается как « *x или y*» и является ложной тогда и только тогда, когда ложны обе переменные.

Таблица 3. Таблица истинности для дизъюнкции.

x	y	$x \vee y$
И	И	И
И	Л	И
Л	И	И
Л	Л	Л

Импликация является бинарной операцией, обозначается $x \rightarrow y$, читается, как « *из x следует y*» и является ложной тогда и только тогда, когда *x* истинна, а *y* - ложна.

Таблица 4. Таблица истинности для импликации.

x	y	$x \rightarrow y$
И	И	И
И	Л	Л
Л	И	И
Л	Л	И

Эквивалентность является бинарной операцией, обозначается $x \leftrightarrow y$, читается как « *x тождественно y*» и является истинной тогда и только тогда, когда значения обеих переменных совпадают.

Таблица 5. Таблица истинности для эквивалентности.

x	y	$x \leftrightarrow y$
И	И	И
И	Л	Л
Л	И	Л
Л	Л	И

2.2.3 Правила построения формул

Словарь исчисления высказываний дает возможность строить составные высказывания из простых высказываний, соединяя их логическими связками. Правила построения S описывают выражения, являющиеся объектами языка. Такие высказывания называются формулами.

Совокупность правил построения формул выглядит так:

- Всякий атом (высказывание) является формулой;
- Если *X* и *Y* - формулы, то $\neg X$, $(X \wedge Y)$, $(X \vee Y)$, $(X \rightarrow Y)$ *и* $(X \leftrightarrow Y)$ – формулы;
- Никаких формул, кроме порожденных применением указанных выше правил, нет.

Круглые скобки позволяют указать порядок, в котором применялись правила. Если в третьем примере утверждений, приведенных выше, обозначим высказывание «идет дождь» буквой *P*, а высказывание «дорога мокрая» буквой *Q,* то, используя правила построения, все утверждение будет выглядеть следующим образом:

$(P \rightarrow Q) \rightarrow (\neg Q \rightarrow \neg P)$.

2.2.4 Интерпретация формул

Объектами изучения естественных и формальных языков являются, в частности, синтаксис, который позволяет распознавать фразы среди наборов слов, и семантика, которая придает определенное значение фразам. Это относится и к исчислению высказываний. Любое высказывание может быть либо истинно, либо ложно.

Введем семантическую область {И, Л}. Интерпретировать формулу - это, значит, приписать ей одно из двух значений истинности: И или Л. Значение истинности формулы зависит только от структуры этой формулы и от значений истинности составляющих ее высказываний. Таблица истинности логических связок исчисления высказываний приведена ниже.

P	Q	$\neg$ P	P $\wedge$ Q	P $\vee$ Q	P $\rightarrow$ Q	P $\leftrightarrow$ Q
И	И	Л	И	И	И	И
И	Л	Л	Л	И	Л	Л
Л	И	И	Л	И	И	Л
Л	Л	И	Л	Л	И	И

Если формула состоит из нескольких атомов, то истинность формулы определяется при всех возможных комбинациях истинностных значениях атомов, встречающихся в формуле.

Рассмотрим формулу: (P $\wedge$ Q) $\rightarrow$ ($\neg$R). Таблица истинности для нее будет выглядеть следующим образом:

P	Q	R	$\neg$R	P $\wedge$ Q	(P $\wedge$ Q) $\rightarrow$ ($\neg$R)
И	И	И	Л	И	Л
И	И	Л	И	И	И
И	Л	И	Л	Л	И
И	Л	Л	И	Л	И
Л	И	И	Л	Л	И
Л	И	Л	И	Л	И
Л	Л	И	Л	Л	И
Л	Л	Л	И	Л	И

Определение 1: *интерпретацией* формулы исчисления высказываний называется такое приписывание истинностных значений атомам формулы, при котором каждому из атомов приписано либо И, либо Л.

Определение 2: Формула *истинна при некоторой интерпретации* тогда и только тогда, когда она получает значение И в этой интерпретации, в противном случае формула ложна.

Определение 3: Формула является *общезначимой (тавтологией)* тогда и только тогда, когда она истинна при всех возможных интерпретациях. Формула является *необщезначимой* тогда и только тогда, когда она не является общезначимой.

Определение 4: Формула является *противоречивой (невыполнимой)* тогда и только тогда, когда она ложна при всех возможных интерпретациях. Формула является *непротиворечивой (выполнимой)* тогда и только тогда, когда она не является противоречивой.

2.2.5 Определение логического следствия

Если Q - тавтология, то ее обозначают как $\models Q$. Если E – множество формул, то запись $E \models Q$ означает, что при всех интерпретациях, при которых истинны все формулы из E, истинна также формула Q. Формула Q называется *логическим следствием* из E. Таким образом, тавтология – логическое следствие из пустого множества.

Если E содержит единственный элемент P, то $P \models Q$. Тогда Q является логическим следствием P тогда и только тогда, когда импликация $P \to Q$ есть тавтология, или $P \models Q \leftrightarrow \models (P \to Q)$.

В более общем виде, можно написать:

$\{ F_1, F_2, \ldots, F_n \} \models Q \leftrightarrow \models (F_1 \wedge F_2 \wedge \ldots F_n) \to / \models Q$.

Выявление того факта, что из множества высказываний (формул исчисления) логически следует некоторое другое высказывание (формула) и является, по существу, одной из основных задач исчисления.

Определение 5: Пусть даны формулы $F_1, F_2, \ldots, F_n$ и формула Q. Говорят, что Q есть *логическое следствие* формул $F_1, F_2, \ldots, F_n$ тогда и только тогда, когда для всякой интерпретации I, в которой $F_1 \wedge F_2 \wedge \ldots F_n$ истинна, Q также истинна. $F_1, F_2, \ldots, F_n$ называются аксиомами (или постулатами, или посылками, или гипотезами).

Если формулы P и Q – логические следствия друг друга, то они называются *логически эквивалентными*. Такая ситуация имеет место тогда и только тогда, когда формула $(P \leftrightarrow Q)$ является тавтологией.

2.2.6 Система аксиом исчисления высказываний

Понятие *тавтологии* совпадает с понятием *теоремы* в *аксиоматической системе.* Аксиоматическая система обладает свойством адекватности, то есть она состоит из множества аксиом, считающихся тавтологиями. Кроме аксиом в аксиоматическую систему входит множество

правил вывода, позволяющих строить новые тавтологии из аксиом и уже полученных тавтологий. Выводимая формула обозначается $\vdash P$.

Исчисление высказываний тоже является аксиоматической системой. Любая аксиоматическая система должна удовлетворять следующим требованиям:

1. Непротиворечивость: невозможность вывода отрицания уже доказанного выражения (которое считается общезначимым);
2. Независимость (минимальность): система не должна содержать бесполезных аксиом и правил вывода. Некоторое выражение *независимо* от аксиоматической системы, если его нельзя вывести с помощью этой системы. В минимальной системе каждая аксиома независима от остальной системы, то есть, не выводима из других аксиом.
3. Полнота (взаимность адекватности): любая тавтология выводима из системы аксиом. В адекватной системе аксиом любая выводимая формула есть тавтология, то есть верно, что $\vdash P \rightarrow \models P$. Соответственно в *полной систем верно:* $\models P \rightarrow \vdash P$.

Некоторое множество тавтологий составляет систему аксиом А. Приведем две наиболее известные системы аксиом, обладающие всеми вышеперечисленными свойствами.

Классическая система аксиом:

1. $P \rightarrow (Q \rightarrow P)$;
2. $(P \rightarrow (Q \rightarrow R)) \rightarrow ((P \rightarrow Q) \rightarrow (P \rightarrow R))$;
3. $(\neg P \rightarrow \neg Q) \rightarrow ((\neg P \rightarrow Q) \rightarrow P)$.

Система аксиом Новикова:

1. $P \rightarrow (Q \rightarrow P)$;
2. $(P \rightarrow (Q \rightarrow R)) \rightarrow ((P \rightarrow Q) \rightarrow (P \rightarrow R))$;
3. $P \wedge Q \rightarrow P$;
4. $P \wedge Q \rightarrow Q$;
5. $(P \rightarrow Q) \rightarrow ((P \rightarrow R) \rightarrow (P \rightarrow Q \wedge R))$;
6. $P \rightarrow P \vee Q$;
7. $Q \rightarrow P \vee Q$;
8. $(P \rightarrow R) \rightarrow ((Q \rightarrow R) \rightarrow (P \vee Q \rightarrow R))$;
9. $(P \rightarrow Q) \rightarrow (\neg Q \rightarrow \neg P)$;
10. $P \rightarrow \neg\neg P$;
11. $\neg\neg P \rightarrow P$.

2.2.7 Правила вывода исчисления высказываний

Существует три обязательных *правила вывода,* входящих в множество *B* в исчислении высказываний:

1. Все аксиомы выводимы.
2. Правило одновременной подстановки: если некоторая тавтология *U* содержит атом *P*, то одновременная замена всех вхождений атома *P* в *U* на любую формулу *Q* приводит к порождению тавтологии.

3. *Modus Ponens* (заключение): если P – тавтология, и $P \to Q$, то Q – тавтология.

Часто необходимо преобразовывать формулы из одной формы в другую. Поэтому, кроме аксиом и правил вывода необходимо иметь набор эквивалентных формул (законов), которые позволяют производить преобразования формул:

1. $P \leftrightarrow Q = P \to Q \wedge Q \to P$;
2. $P \to Q = \neg P \vee Q$;
3. *Коммуникативные законы:* $P \vee Q = Q \vee P$; $P \wedge Q = Q \wedge P$;
4. *Ассоциативные законы:* $(P \vee Q) \vee R = Q \vee (P \vee R)$; $(P \wedge Q) \wedge R = Q \wedge (P \wedge R)$;
5. *Дистрибутивные законы:* $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$; $P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$;
6. *Законы идемпотентности:* $P \vee P = P$; $P \wedge P = P$;
7. $P \vee Л = P$; $P \wedge И = P$;
8. $P \vee И = И$; $P \wedge Л = Л$;
9. $P \vee \neg P = И$; $P \wedge \neg P = Л$;
10. $\neg(\neg P) = P$;
11. *Законы де Моргана:* $\neg(P \vee Q) = \neg Q \wedge \neg P$; $\neg(P \wedge Q) = \neg Q \vee \neg P$;

Можно доказать следующие дополнительные правила вывода:

Утверждение 1.

Если P - тавтология, то Q → P – тавтология.

Доказательство.

Доказательство следует из аксиомы 1 классической системы аксиом и правила вывода 3.

По аксиоме 1 $P \to (Q \to P)$, тогда, если P – тавтология, то по правилу *Modus Ponens* $Q \to P$ – тоже тавтология.

Утверждение 2.

Свойство транзитивности отношения следования: если $P \to Q$, $Q \to R$ – тавтологии, то $P \to R$ – тавтология.

Доказательство.

Эквивалентная формулировка утверждения 2 заключается в том, что формула $(((P \to Q) \wedge (Q \to R)) \to (P \to R))$ является тавтологией. Воспользуемся законами эквивалентных преобразований исчисления высказываний:

$(((P \to Q) \wedge (Q \to R)) \to (P \to R)) \leftrightarrow$

$\neg((\neg P \vee Q) \wedge (\neg Q \vee R)) \vee (\neg P \vee R) \leftrightarrow$

$((P \wedge \neg Q) \vee (Q \wedge \neg R)) \vee (\neg P \vee R) \leftrightarrow$

$((P \vee \neg P) \wedge (\neg Q \vee \neg P)) \vee ((Q \vee R) \wedge (\neg R \vee R)) \leftrightarrow$

$((И \wedge (\neg Q \vee \neg P)) \vee ((Q \vee R) \wedge И)) \leftrightarrow$

$\neg Q \vee \neg P \vee Q \vee R \leftrightarrow$

$И \vee \neg P \vee Q \leftrightarrow И$.

Утверждение 3.

Теорема дедукции: необходимым и достаточным условием выводимости Q из гипотез R, P является выводимость $P \rightarrow Q$ из R. Данную теорему можно записать следующим образом: $R, P \vdash Q \leftrightarrow R \vdash (P \rightarrow Q)$.

2.3 Исчисление предикатов первого порядка.

2.3.1 Основные определения

Термин «первого порядка» означает, что в этой теории кванторы допускаются только по переменным и не допускаются по функциональным и предикатным символам. Кроме того, запрещены предикаты, которые в качестве своих аргументов имеют другие предикаты.

В логике высказываний атом рассматривается как единое целое, его структура и состав не анализируется. Однако, есть много умозаключений, которые не могут быть представлены таким простым способом. Например, рассмотрим следующее умозаключение:

Каждый человек смертен.

Так как Конфуций человек, то он смертен.

Приведенное рассуждение интуитивно корректно, однако, если мы введем обозначения:

P: Каждый человек смертен,

Q: Конфуций – человек,

R: Конфуций смертен,

то R не есть логическое следствие P и Q в рамках логики высказываний.

В логике предикатов первого порядка по сравнению с логикой высказываний имеет еще три логических понятия: термы, предикаты и кванторы.

Множество T базовых элементов исчисления предикатов включает в себя следующие символы:

1. Константы – это обычно строчные буквы a, b, c… или осмысленные имена объектов;
2. Переменные – это обычно строчные буквы x, y, z, …, возможно с индексами;
3. Функции – это обычно строчные буквы f, g, h,… или осмысленные слова из строчных букв;
4. Предикаты – это обычно прописные буквы P, Q, R … или осмысленные слова из прописных букв;
5. Логические связки: отрицание, дизъюнкция, конъюнкция, импликация, эквивалентность;
6. Кванторы всеобщности и существования - $\forall$, $\exists$;
7. Открывающая и закрывающая скобка.

Всякая функция или предикатный символ имеет определенное число аргументов. Если функция f имеет n аргументов, то f называется n- местной функцией. Аналогично, если предикат P имеет n аргументов, то P называется n- местным предикатом.

Определение 6. Термы определяются рекурсивно следующим образом:

- *Константа есть терм;*
- *Переменная есть терм;*
- *Если f есть n- местная функция и $t_1, t_2,\ldots,t_n$ – термы, то f ($t_1, t_2,\ldots,t_n$) – терм;*
- *Никаких термов, кроме порожденных применением указанных выше правил, нет.*

Определение 7. Предикат $P(t_1, t_2,\ldots,t_n)$ есть логическая функция, определенная на множестве термов $t_1, t_2,\ldots,t_n$, при фиксированных значениях которых она превращается в высказывания со значением истина (И) или ложь(Л).

Определение 8. Если P – n- местный предикат и $t_1, t_2,\ldots,t_n$ – термы, то $P(t_1, t_2,\ldots,t_n)$ – атом.

Для построения формул в исчислении предикатов используются пять логических связок и два квантора: $\forall$ - всеобщности и $\exists$ - существования. Если x – переменная, то $(\forall x)$ читается как «для всех x», «для каждого x», «для любого x», тогда как $(\exists x)$ читается как « существует x», «для некоторых x», «по крайней мере, для одного x».

Пример 1: запишем следующие утверждения:

1. Каждое рациональное число есть вещественное число.
2. Существует число, которое является простым.
3. Для каждого числа x существует такое число y , что x<y.

Обозначим *«x есть простое число»* через *P(x), «x есть рациональное число»* через *Q(x), «x есть вещественное число»* через *R(x)* и *«x меньше y»* через *МЕНЬШЕ (x, y)*.

Тогда указанные выше утверждения могут быть записаны соответственно выражениями:

1. $(\forall x)\,(Q(x)\rightarrow R(x))$,

2. $(\exists x)\,P(x)$,

3. $(\forall x)\,(\exists y)$ МЕНЬШЕ (x, y).

Каждое из выражений *1, 2, 3* называется формулой. Прежде чем дать формальное определение формулы в логике предикатов, следует установить различие между *связанными переменными* и *свободными переменными* и определить *область действия квантора*, входящего в формулу, как ту формулу, к которой этот квантор применяется. Так, область действия квантора существования в выражении *3* есть *МЕНЬШЕ (x, y)*, а область действия квантора всеобщности в выражении *3* есть *$(\exists y)$ МЕНЬШЕ (x, y)*.

Определение 9. Вхождение переменной *x* в формулу *F* называется связанным тогда и только тогда, когда оно совпадает с вхождением в квантор $(\forall xF)$ или $(\exists yF)$. Формула, к которой применяется квантор по данной переменной, называется *областью действия этого квантора*. Вхождение переменной в формулу свободно тогда и только тогда, когда оно не находится в области действия какого-либо квантора. Формула, не

содержащая свободных переменных, называется *замкнутой* и представляет собой *высказывание.*

Пример2.

$P(x, y) \wedge R(z) \rightarrow (\forall x)(R(x) \wedge Q(y))$. В этом примере первое вхождение переменной x является свободным, второе – связанным, первое вхождение переменной y является свободным, второе – связанным, единственное вхождение переменной z является свободным.

Отметим, что переменная в формуле может быть свободной и связанной одновременно.

2.3.2 Правила построения формул в исчислении предикатов

Определение 10. Правильно построенные формулы логики первого порядка рекурсивно определяются следующим образом:

1. Атом есть формула.
2. Если F и G – формулы, то $\neg(F)$, $(F \vee G)$, $(F \wedge G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$ – формулы.
3. Если F – формула, а x – свободная переменная в F, то $(\forall x)\, F$ и $(\exists x)\, F$ –формулы.
4. Формулы порождаются только конечным числом применений правил 1-3.

Определение 11. Терм t называется свободным для переменной x в формуле F, если ни x, ни другая произвольная переменная из t не находится в области действия никакого квантора $\forall x$ или $\exists x$ в F.

Пример3. Если термом является переменная y, а формула имеет вид $((\forall y)P(y)) \wedge Q(x)$, то y свободно для x, так как x не находится в области действия квантора по y, чего нельзя сказать в случае, если формула имеет вид $(\forall y)(P(y) \wedge Q(x))$.

При этом выполняются следующие правила:

- любой терм, не содержащий переменных, свободен для любой переменной в данной формуле;
- терм свободен для любой переменной в формуле, если никакая переменная терма не является связанной в этой формуле;
- переменная x свободна для x в любой формуле;
- любой терм свободен для переменной в формуле, не содержащей свободных вхождений этой переменной.

В аксиомах $(\forall x)F(x) \rightarrow F(t)$ и $F(t) \rightarrow (\exists x)F(x)$ t свободен для x в формуле F, F(t) получена из F(x)заменой всех свободных вхождений x на t.

Пример4: переведем в формулу утверждение «Каждый человек смертен. Конфуций – человек, следовательно, Конфуций смертен».

Обозначим «x есть человек» через $P(x)$, а «x смертен» через $Q(x)$. Тогда утверждение «Каждый человек смертен» может быть представлено формулой $(\forall x)\, (P(x) \rightarrow Q(x))$, утверждение «Конфуций – человек» формулой $P(Конфуций)$ и «Конфуций смертен» формулой $Q(Конфуций)$.

Утверждение в целом может быть представлено формулой

$(\forall x)\, (P(x) \rightarrow Q(x)) \wedge P(Конфуций) \rightarrow Q(Конфуций)$.

2.3.3 Интерпретация формул в логике предикатов первого порядка.

Как в логике высказываний, так и в логике предикатов, существуют два метода получения общезначимых формул: семантический, на основе интерпретации формул, и синтаксический, на основе формального вывода из аксиом и правил вывода и вспомогательных теорем.

Для исчисления высказываний существует метод построения таблиц истинности. Для исчисления предикатов этот метод либо затруднён, либо невозможен, так как область интерпретации может быть бесконечной. Если область интерпретации $D=\{a_1, a_2,\ldots,a_n\}$ конечна, то

$\forall x\, P(x) \equiv P(a_1) \wedge P(a_2) \wedge \ldots \wedge P(a_n)$;

$\exists x\, P(x) \equiv P(a_1) \vee P(a_2) \vee \ldots \vee P(a_n)$.

Заменив все формулы содержащие кванторы, с помощью этих соотношений, получим формулу, не содержащую кванторы. Истинность такой формулы на конечной области проверяется с помощью конечного числа подстановок и вычислений значений истинности. Для бесконечной области этот метод непригоден.

И семантический и синтаксический метод приводят к одному и тому же результату.

Терема Гёделя о полноте исчисления предикатов.

Исчисление предикатов первого порядка – полная формальная теория.

Теорема Чёрча.

Исчисление предикатов неразрешимо.

Чтобы определить интерпретацию для формулы логики первого порядка, мы должны указать предметную область, значения констант, функций и предикатов, встречающихся в формуле.

Определение 12. Интерпретация формулы F логики первого порядка состоит из непустой (предметной) области D и указания значения всех констант, функций и предикатов, встречающихся в F.

1. Каждой константе мы ставим в соответствие некоторый определенный элемент из D.
2. Каждой n- местной функции мы ставим в соответствие отображение из D^n в D (заметим, что $D^n = \{(x_1, x_2,\ldots,x_n) / x_1 \in D, x_2 \in D, \ldots, x_n \in D\}$).
3. Каждому n- местному предикату мы ставим в соответствие отображение D^n в {И, Л}.

Для каждой интерпретации формулы из области D формула может получить значение И или Л согласно следующим правилам:

1. Если заданы значения формул F и G, то истинностные значения формул $\neg(F)$, $(F \vee G)$, $(F \wedge G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$ получаются с помощью таблиц истинности соответствующих логических связок.
2. $(\forall x)$ F получает значение И, если F получает значение И для каждого x из D , в противном случае она получает значение Л.
3. $(\exists x)$ F получает значение И, если F получает значение И хотя бы для одного x из D , в противном случае она получает значение Л.

Отметим, что формула, содержащая свободные переменные, не может получить истинностное значение. Поэтому, в дальнейшем, будем считать, что формула либо не содержит свободных переменных, либо свободные переменные рассматриваются как константы.

Пример 5. Рассмотрим формулы $(\forall x)\ P(x)$ *и* $(\exists x)\ \neg P(x)$.

Пусть интерпретации такова: область - D={1,2};

Оценка для P :P(1) =И, P(2) =Л.

В таком случае $(\forall x)\ P(x)$ *есть Л, а* $(\exists x)\ \neg P(x)$ *есть И в данной интерпретации.*

Пример 6. Рассмотрим формулы $(\forall x)\ (\exists y)\ P(x, y)$.

Пусть интерпретации такова: область - D={1,2};

Оценка для P: P(1,1) =И, P(1,2) =Л, P(2,1) =Л, P(2,2) =И.

При x=1 существует y=1, что P(x, y) =И.

При x=2 существует y=2, что P(x, y) =И.

Следовательно, в указанной интерпретации, для каждого x из D существует такой y, что P(x, y)=И, то есть $(\forall x)\ (\exists y)\ P(x, y)$ *есть И в данной интерпретации.*

Определение 12: Формула *G* называется *непротиворечивой (выполнимой) в данной интерпретации,* если при некоторой подстановке констант в *G*, она превращается в истинное высказывание. В противном случае она называется *невыполнимой (противоречивой, ложной) в данной интерпретации*.

Определение 13: Формула *G* называется *истинной в данной интерпретации,* если она превращается в истинное высказывание при любой подстановке констант.

Определение 14: Формула *G* истинная в любой интерпретации, называется *тождественно истинной, общезначимой или тавтологией.*

Определение 15: Формула *G* ложная в любой интерпретации, называется *тождественно ложной, противоречивой или невыполнимой.*

Определение 16: Формула *G* есть *логическое следствие* формул F_1, $F_2, \ldots, F_n$ тогда и только тогда, когда для каждой интерпретации *I*, если $F_1 \wedge F_2 \wedge \ldots F_n$ истинна в *I*, то *G* также истинна в *I*.

Пример 7. Рассмотрим формулы:

F1: $(\forall x)\ (P(x) \rightarrow Q(x))$,

F2: $P(a)$.

Докажем, что Q(a) есть логическое следствие формул F1 и F2.

Рассмотрим любую интерпретацию I, в которой $\forall x(P(x) \rightarrow Q(x)) \wedge P(a)$ *является истинной. Конечно, в этой интерпретации P(a) есть И. Пусть Q(a) есть Л в данной интерпретации, тогда* $P(a) \rightarrow Q(a)$ *есть Л в данной интерпретации по определению операции импликации. Это значит, что* $\forall x\ (P(x) \rightarrow Q(x))$ *есть Л в I, что невозможно. Следовательно, Q(a) должна быть И в каждой интерпретации, которая удовлетворяет* $\forall x\ (P(x) \rightarrow Q(x)) \wedge P(a)$. *Это означает, что Q(a) есть логическое следствие из F1 и F2.*

Определение 17: *Формулы называются эквивалентными, если при всех подстановках констант (при всех интерпретациях) они принимают одинаковые значения. В частности, все общезначимые (тождественно истинные) и все противоречивые (тождественно ложные формулы) эквивалентны.*

Утверждение 8. Докажем эквивалентность следующих формул:

$\neg((\exists x)\, P(x)) \equiv (\forall x)\, (\neg P(x))$.

Доказательство:

Если правая часть истинна, то для любого x истинна $\neg P(x)$, то есть ложна P(x), а значит, не найдётся x, при котором истинна P(x), следовательно, истинно высказывание $\neg\exists x\, P(x)$.

Если правая часть ложна, то найдётся x=a такое, что P(a) истина, следовательно, существует x, при котором истинна P(x), и значит, $\neg\exists x\, P(x)$ ложна.

Таким образом, эти формулы эквивалентны.

2.3.4 Системы аксиом логики предикатов.

Системы аксиом исчисления высказываний остаются верными и в исчислении предикатов первого порядка, только к ним следует добавить еще две аксиомы, которые дают возможность оперировать с кванторами:

1. $(\forall x)\, P(x) \to P(y)$;
2. $P(y) \to (\exists x)\, P(x)$.

Эти 2 аксиомы, добавленные в классическую систему аксиом или в систему аксиом Новикова, образуют системы аксиом, обладающие свойствами полноты, независимости и непротиворечивости.

2.3.5 Правила вывода в исчислении предикатов.

Из правил вывода исчисления высказываний в исчислении предикатов действует только правило Modus Ponens. Правило одновременной подстановки модифицировано, а остальные правила вывода касаются выводимости формул, содержащих кванторы. В этих правилах предполагается, что $G(x)$ содержит свободные вхождения x. а F их не содержит.

1. *Modus Ponens: Если выводима формула F и выводима формула $F \to G$, то выводима и формула G. Часто это правило записывают следующим образом:*

   $$\frac{F,\ F \to G}{G};$$

2. *Правило одновременной подстановки: если терм t свободен для переменной x в формуле F, то можно подставить терм t вместо переменной x во всех вхождениях x в F.*
3. *Правило обобщения:*

   $$\frac{F \to G(x);}{F \to \forall x\, G(x)}$$

4. *Правило конкретизации:*
   $\underline{G(x) \to F}$;
   $\exists x\, G(x) \to F$
5. *Правило переименования. Из выводимости формулы G(x), содержащей свободное вхождение x, ни одно из которых не содержится в области действия кванторов $\forall y$ и $\exists y$ следует выводимость G(y).*
   Пример 6. Докажем правило переименования:

1. *+ G(x);*
2. *Из аксиомы 2 классической системы следует, что $G(x) \to (F \to G(x))$, где F – тавтология, не содержащая свободный вхождений x;*
3. *По правилу Modus Ponens следует, что* $\dfrac{G(x), G(x) \to (F \to G(x))}{F \to G(x)}$;
4. *Используя правило обобщения, получаем:* $F \to \forall x G(x)$;
5. *По правилу Modus Ponens следует, что* $\dfrac{F, F \to \forall x G(x)}{\forall x G(x)}$;
6. *Из аксиомы 2 логики предикатов и правила Modus Ponens следует, что* $\dfrac{\forall x G(x), \forall x G(x) \to G(y)}{G(y)}$.

2.3.6 Законы эквивалентных преобразований логики предикатов.

Законы эквивалентных преобразований логики высказываний используются и в логике предикатов. Кроме них, существуют другие эквивалентные формулы, содержащие кванторы.

Пусть *G* есть формула, содержащая свободную переменную *x*. Пусть *F* есть формула, которая не содержит переменной *x*. Тогда следующие пары эквивалентных формул являются законами эквивалентных преобразований логики предикатов:

1. *$(\forall x)\, G(x) \vee F = (\forall x)\, (G(x) \vee F)$;*
2. *$(\forall x)\, G(x) \wedge F = (\forall x)\, (G(x) \wedge F)$;*
3. *$(\exists x)\, G(x) \vee F = (\exists x)\, (G(x) \vee F)$;*
4. *$(\exists x)\, G(x) \wedge F = (\exists x)\, (G(x) \wedge F)$;*
5. *$\neg((\forall x)\, G(x)) = (\exists x)\, (\neg G(x))$;*
6. *$\neg((\exists x)\, G(x)) = (\forall x)\, (\neg G(x))$;*
7. *$(\forall x)\, G(x) \wedge (\forall x)\, F(x) = (\forall x)\, (G(x) \wedge F(x))$;*
8. *$(\exists x)\, G(x) \vee (\exists x)\, F(x) = (\exists x)\, (G(x) \vee F(x))$;*

Правила 7 и 8 называются правилами выноса кванторов, которые позволяют распределять квантор всеобщности и квантор существования по операциям конъюнкции и дизъюнкции соответственно. Следует отметить, что *нельзя* распределять квантор всеобщности и квантор существования по операциям дизъюнкции и конъюнкции соответственно, то есть не эквивалентны следующие пары формул:

$(\forall x)\, G(x) \vee (\forall x)\, F(x) <> (\forall x)\, (G(x) \vee F(x))$;

$(\exists x)\, G(x) \wedge (\exists x)\, F(x) <> (\exists x)\, (G(x) \wedge F(x))$;

В подобных случаях можно заменить связанную переменную x в формуле $(\forall x)\, F(x)$ на переменную z, которая не встречается в $G(x)$, так как связанная переменная является лишь местом для подстановки какой угодно переменной. Формула примет вид: $(\forall x)\, F(x) = (\forall z)\, F(z)$. Пусть z не встречается в $G(x)$. Тогда

$(\exists x)\, G(x) \wedge (\exists x)\, F(x) = (\exists x)\, G(x) \wedge (\exists z)\, F(z)$

$= (\exists x)\, (\exists z)(\, G(x) \wedge F(z))$ по правилу 1.

Аналогично, можно написать

$(\forall x)\, G(x) \vee (\forall x)\, F(x) = (\forall x)\, G(x) \vee (\forall z)\, F(z)$

$= (\forall x)\, (\forall z)(\, G(x) \vee F(z))$ по правилу 1.

В общем случае эти правила можно записать в следующем виде:

$9.(K_1 x)\, G(x) \vee (K_2 x)\, F(x) = (K_1 x)\, (K_2 z)(\, G(x) \vee F(z))$;

$10.(K_3 x)\, G(x) \wedge (K_4 x)\, F(x) = (K_3 x)\, (K_4 z)(\, G(x) \wedge F(z))$,

где K_1, K_2, K_3, K_4 есть кванторы $\forall$ или $\exists$, а z не входит в G(x).

Когда $K_1 = K_2 = \exists$ и $K_3 = K_4 = \forall$, то необязательно переименовывать переменную x, можно прямо использовать правила 7 и 8.

Используя законы эквивалентных преобразований логики высказываний и логики предикатов, всегда можно преобразовать любую формулу в ПНФ.

2.3.7 Теоремы о логическом следствии

Определение 18. *Выводом* формулы G из формул U_1, $U_2, \ldots$, U_n называется последовательность формул F_1, $F_2, \ldots$, F_n такая, что F_m есть G, а любая F_i либо аксиома, либо одна из формул U_i , либо формула, непосредственно выводимая из предшествующих ей формул.

Вследствие законов ассоциативности скобки в выражениях, связанных отношениями дизъюнкции и конъюнкции могут быть опущены, при этом выражение $F_1 \vee F_2 \vee \ldots \vee F_n$ называется дизъюнкцией формул F_1, $F_2, \ldots$, F_n, а выражение $F_1 \wedge F_2 \wedge \ldots \wedge F_n$ называется конъюнкцией формул F_1, $F_2, \ldots$, F_n.

Теорема 1. Пусть даны формулы F_1, $F_2, \ldots$, F_n и формула G. G есть логическое следствие F_1, $F_2, \ldots$, F_n тогда и только тогда, когда формула $((F_1 \wedge F_2 \wedge \ldots \wedge F_n) \rightarrow G)$ общезначима.

Теорема 2. Пусть даны формулы F_1, $F_2, \ldots$, F_n и формула G. G есть логическое следствие F_1, $F_2, \ldots$, F_n тогда и только тогда, когда формула $(F_1 \wedge F_2 \wedge \ldots \wedge F_n \wedge \neg G)$ противоречива.

Замечание.

Для того чтобы доказать, что данная формула является тавтологией, достаточно доказать, что ее отрицание является противоречием:

$\neg((F_1 \wedge F_2 \wedge \ldots \wedge F_n) \rightarrow G) =$

$\neg(\neg(F_1 \wedge F_2 \wedge \ldots \wedge F_n) \vee G) =$

$(F_1 \wedge F_2 \wedge \ldots \wedge F_n) \wedge \neg G)$.

Теоремы 1 и 2 очень важны. Из них следует, что доказательство логического следствия одной формулы из конечного множества формул

эквивалентно доказательству того факта, что некоторая связанная с конечным множеством формула общезначима или противоречива.

2.3.8 Предваренные (пренексные) нормальные формы исчисления предикатов.

Определение 19. Литерал (литера) есть атом или отрицание атома.

Определение 20. Формула F находится *в конъюнктивной нормальной форме* тогда и только тогда, когда F имеет вид: $F_1 \wedge F_2 \wedge \ldots \wedge F_n$, $n >= 1$, где каждая из $F_1, F_2, \ldots, F_n$ есть *дизъюнкция литералов*.

Определение 21. Формула F находится *в дизъюнктивной нормальной форме* тогда и только тогда, когда F имеет вид: $F_1 \vee F_2 \vee \ldots \vee F_n$, $n >= 1$, где каждая из $F_1, F_2, \ldots, F_n$ есть *конъюнкция литералов*.

В логике высказываний были введены две нормальные формы – КНФ и ДНФ. В логике предикатов также существуют нормальная форма - ПНФ, которая используется для упрощения процедуры доказательства общезначимости или противоречивости формул.

Определение 22: *Формула F логики предикатов находится в предваренной нормальной форме, тогда и только тогда, когда формула F имеет вид:*

$(K_1x_1)\ldots(K_nx_n)$ (M), где каждое (K_ix_i), $i = 1,\ldots,n$, есть или $(\forall x_i)$ или $(\exists x_i)$, и M есть формула, не содержащая кванторов. $(K_1x_1)\ldots(K_nx_n)$ называется префиксом, а M – матрицей формулы F.

Алгоритм преобразования формул в ПНФ.

Шаг 1. Используем законы законы 1 и 2 исчисления высказываний для того, чтобы исключить логические связки импликации и эквивалентности.

Шаг 2. Многократно используем закон двойного отрицания, законы де Моргана и законы 5 и 6 исчисления предикатов, чтобы внести знак отрицания внутрь формулы.

Шаг 3. Переименовываем связанные переменные, если это необходимо.

Шаг 4. Используем законы 1, 2, 3, 4, 7,8, 9 и 10 до тех пор, пока все кванторы не будут вынесены в самое начало формулы, чтобы получить ПНФ.

Пример 6. Приведем формулу $(\forall x)\, P(x) \rightarrow (\exists x)\, Q(x)$ к ПНФ:

$(\forall x)\, P(x) \rightarrow (\exists x)\, Q(x) = \neg((\forall x)\, P(x)) \vee (\exists x)\, Q(x)$(по закону 1 логики высказываний)

$= (\exists x)(\neg P(x)) \vee (\exists x)\,)\, Q(x)$(по закону 5 логики предикатов)

$= (\exists x)(\neg P(x) \vee Q(x))$(по закону 8 логики предикатов), что и есть ПНФ исходной формулы.

Пример 7. Привести формулу $(\forall x)\, (\forall y)((\exists z)(P(x, z) \wedge P(y, z)) \rightarrow (\exists u)\, Q(x, y, u))$ в ПНФ:

$(\forall x)\, (\forall y)((\exists z)(P(x, z) \wedge P(y, z)) \rightarrow (\exists u)\, Q(x, y, u))$

$= (\forall x)\, (\forall y)(\neg((\exists z)(P(x, z) \wedge P(y, z))) \vee (\exists u)\, Q(x, y, u))$(исключаем $\rightarrow$)

$= (\forall x)\, (\forall y)((\forall z)\, (\neg P(x, z) \vee \neg P(y, z)) \vee (\exists u)\, Q(x, y, u))$(по закону 6 законам де Моргана)

$= (\forall x)(\forall y)(\forall z)(\exists u)\, (\neg P(x, z) \vee \neg P(y, z)) \vee Q(x, y, u))$(закон 3)

2.4 Автоматизация доказательства в логике предикатов.

2.4.1 История вопроса

Поиск общей разрешающей процедуры для проверки общезначимости формул начал Лейбниц в XVII веке, затем был продолжен в начале XX века до тех пор, пока в 1936 году Черч и Тьюринг независимо не доказали, что не существует никакой общей разрешающей процедуры, никакого алгоритма, проверяющего общезначимость формул в логике предикатов первого порядка.

Тем не менее, существуют алгоритмы поиска доказательства, которые могут подтвердить, что формула общезначима, если она на самом деле общезначима (для необщезначимых формул эти алгоритмы, вообще говоря, не заканчивают свою работу).

Очень важный подход к автоматическому доказательству теорем был дан Эрбраном в 1930 году. По определению общезначимая формула есть формула, которая истинна при всех интерпретациях. Эрбран разработал алгоритм нахождения интерпретации, которая опровергает данную формулу. Однако, если данная формула действительно общезначима, то никакой интерпретации не существует и алгоритм заканчивает работу за конечное число шагов. Метод Эрбрана служит основой для большинства современных автоматических алгоритмов поиска доказательства.

Гилмор в 1959 году одним из первых реализовал процедуру Эрбрана. Его программа была предназначена для обнаружения противоречивости отрицания данной формулы, так как формула общезначима тогда и только тогда, когда ее отрицание противоречиво. Однако, программа Гилмора оказалась неэффективной и в 1960 году метод Гилмора был улучшен Девисом и Патнемом. Однако их улучшение оказалось недостаточным, так как многие общезначимые формулы логики предикатов все еще не могли быть доказаны на ЭВМ за разумное время.

Главный шаг вперед сделал Робинсон в 1965 году, который ввел так называемый метод резолюций, который оказался много эффективней, чем любая описанная ранее процедура. После введения метода резолюций был предложен ряд стратегий для увеличения его эффективности. Такими стратегиями являются *семантическая резолюция, лок-резолюция, линейная резолюция, стратегия предпочтения единичных и стратегия поддержки.*

2.4.2 Скулемовские стандартные формы.

Процедуры доказательства по Эрбрану или методу резолюций на самом деле являются процедурами опровержения, то есть вместо доказательства общезначимости формулы доказывается, что ее отрицание противоречиво. Кроме того, эти процедуры опровержения применяются к некоторой стандартной форме, которая была введена Девисом и Патнемом. По существу они использовали следующие идеи:

1. Формула логики предикатов может быть сведена к ПНФ, в которой матрица не содержит никаких кванторов, а префикс есть последовательность кванторов.

2. Поскольку матрица не содержит кванторов, она может быть сведена к конъюнктивной нормальной форме.
3. Сохраняя противоречивость формулы, в ней можно исключить кванторы существования путем использования скулемовских функций.

Алгоритм преобразования формулы в ПНФ известен. При помощи законов эквивалентных преобразований логики высказываний можно свести матрицу к КНФ.

Алгоритм преобразования формул в ДНФ и КНФ.

Шаг 1. Используем законы законы 1 и 2 исчисления высказываний для того, чтобы исключить логические связки импликации и эквивалентности.

Шаг 2. Многократно используем закон двойного отрицания, и законы де Моргана, чтобы внести знак отрицания внутрь формулы.

Шаг 3. Несколько раз используем дистрибутивные законы и другие законы, чтобы получить НФ.

Алгоритм преобразования формулы $(K_1x_1)\ldots(K_nx_n)\ (M)$, где каждое $(K_ix_i),\ i = 1,\ldots,n,$ есть или $(\forall x_i)$ или $(\exists x_i)$, и M есть *КНФ* в скулемовскую нормальную форму (СНФ) приведен ниже.

Алгоритм преобразования ПНФ в ССФ.

Шаг 1. Представим формулу в ПНФ $(K_1x_1)\ldots(K_nx_n)\ (M)$, где M есть *КНФ*. Пусть K_r есть квантор существования в префиксе $(K_1x_1)\ldots(K_nx_n)$, $1<=r<=n$.

Шаг 2. Если никакой квантор всеобщности не стоит левее K_r – выберем новую константу c, отличную от других констант, входящих в M, заменим все x_r в M на c и вычеркнем K_rx_r из префикса. Если $K_1,\ldots K_i$ – список всех кванторов всеобщности, встречающихся в префиксе левее K_r, $1< i<r$, выберем новый i –местный функциональный символ f, отличный от других функциональных символов, заменим все x_r в M на $f(x_1, x_2,\ldots x_i)$ и вычеркнем K_rx_r из префикса.

Шаг 3. Применим шаг 2 для всех кванторов существования в префиксе. Последняя из полученных формул есть *скулемовская стандартная форма* формулы. Константы и функции, используемые для замены переменных квантора существования, называются *скулемовскими* функциями.

Пример 8. Получить ССФ для формулы $(\exists x)(\forall y)(\forall z)(\exists u)(\forall v)(\exists w)\ (P(x, y, z, u, v, w)$.

В этой формуле левее $(\exists x)$ *нет никаких кванторов всеобщности, левее* $(\exists u)$ *стоят* $(\forall y)$ *и* $(\forall z)$, *а левее* $(\exists w)$ *стоят* $(\forall y)$, $(\forall z)$ *и* $(\forall v)$. *Следовательно, мы заменим переменную x на константу a, переменную u - на двухместную f(y, z), переменную w - на трехместную функцию g(y, z, v). Таким образом, мы получаем следующую стандартную форму написанной выше формулы:*

$(\forall y)(\forall z)(\forall v)(P(a, y, z, f(y, z), g(y, z, v))$.

Определение 22: *Дизъюнктом называется дизъюнкция литералов. Дизъюнкт, содержащий r литералов, называется r- литеральным*

дизъюнктом. Однолитеральный дизъюнкт называется единичным дизъюнктом. Если дизъюнкт не содержит никаких литералов, то он называется пустым дизъюнктом- □ . Так как пустой дизъюнкт не содержит литер, которые могли бы быть истинными при любых интерпретациях, то пустой дизъюнкт всегда ложен.

Определение 23: *Множество дизъюнктов S есть конъюнкция всех дизъюнктов из S , где каждая переменная в S считается управляемой квантором всеобщности.*

Вследствие последнего определения, ССФ может быть представлена множеством дизъюнктов.

Пример 9. Получить скулемовскую стандартную форму формулы

$(\forall x)\,(P(x) \wedge (\forall y)(\neg (\forall z)Q(z, y) \rightarrow (\exists u)R(u, y)))$ и представить её в виде множества дизъюнктов.

1. Исключим связки импликации:

$(\forall x)\,(P(x) \wedge (\forall y)(\neg \neg(\forall z)Q(x, y) \vee (\exists u)R(u, x, y)))$.

2. Удалим бесполезные кванторы:

$(\forall x)\,(P(x) \wedge (\forall y)(\neg \neg Q(z, y) \vee (\exists u)R(u, y)))$.

3. Применим правило двойного отрицания:

$(\forall x)\,(P(x) \wedge (\forall y)(Q(z, y) \vee (\exists u)R(u, y)))$.

4. Переместим кванторы в начало формулы:

$(\forall x)(\forall y)(\exists u)\,(P(x) \wedge (Q(z, y) \vee R(u, y)))$,

$(\forall x)(\forall y)(\exists u)\,(P(x) \wedge (Q(z, y) \vee R(u, y)))$.

Получим ПНФ.

$(\forall x)(\forall y)(\exists u)(P(x) \wedge (Q(z, y) \vee R(u, y)))$, у которой матрица находится в КНФ.

Избавимся от кванторов существования в префиксе:

Так как перед $(\exists u)$ есть $(\forall x),(\forall y)$ то переменная и заменяется двухместной функцией f(x, y). Таким образом, мы получаем следующую стандартную форму:

$(\forall x)(\forall y)(P(x) \wedge (Q(z, y) \vee R(f(x, y), y)))$.

Получим ССФ.

Отбросим кванторы всеобщности и заменим конъюкцию на перечисление:

$\{P(x), (Q(z, y) \vee R(f(x, y), y))\}$.

Получим множество из двух дизъюнктов.

Пример 10. Получить скулемовскую стандартную форму формулы

$(\forall x)(\exists y)(\exists z)((\neg P(x, y) \wedge Q(x, z)) \vee R(x, y, z) \wedge (\neg Q(x, z) \wedge P(x, y)))$.

Сначала сведем матрицу к КНФ:

$((\neg P(x, y) \vee R(x, y, z)) \wedge (Q(x, z) \vee R(x, y, z)) \wedge (\neg Q(x, z) \wedge P(x, y)))$.

Затем избавимся от кванторов существования в префиксе:

Так как перед $(\exists y)(\exists z)$ есть $(\forall x)$, то переменные y, z заменяются соответственно одноместными функциями f(x), g(x). Таким образом, мы получаем следующую стандартную форму:

$(\forall x)(((\neg P(x, f(x)) \vee R(x, f(x), g(x))) \wedge (Q(x, g(x)) \vee R(x, f(x), g(x)) \wedge (\neg Q(x, g(x)) \wedge P(x, f(x))))$.

Представим полученную ССФ в виде множества дизъюнктов:

$\{\neg P(x, f(x)) \vee R(x, f(x), g(x)), Q(x, g(x)) \vee R(x, f(x), g(x)), \neg Q(x, g(x)) \wedge P(x, f(x))\}$.

Теорема 3. Пусть S – множество дизъюнктов, которые представляют ССФ формулы F. Тогда F противоречива в том и только в том случае, когда S противоречиво.

На основании теорем 2 и 3 можно сделать вывод, что формула G является логическим следствием формулы F тогда, когда противоречива конъюнкция множества S и формулы $\neg G$, то есть противоречива формула $S_1 \wedge S_2 \wedge ...S_n \wedge \neg G$. Таким образом, если в множество S добавить негативный литерал $\neg G$ и доказать, что полученное множество противоречиво, то тем самым можно доказать выводимость G из множества S.

2.4.3 Метод резолюций в исчислении высказываний.

Основная идея метода резолюций состоит в том, чтобы проверить, содержит ли множество дизъюнктов пустой дизъюнкт. Если множество содержит пустой дизъюнкт, то оно противоречиво (невыполнимо). Если множество не содержит пустой дизъюнкт, то проверяется следующий факт: может ли пустой дизъюнкт быть получен из данного множества. Множество содержит пустой дизъюнкт, тогда и только тогда, когда оно пустое. Если множество можно свести к пустому, то тем самым можно доказать его противоречивость. В этом и состоит метод резолюций, который часто рассматривают как специальное правило вывода, используемое для порождения новых дизъюнктов из данного множества.

Определение 24:Если A атом, то литералы A и $\neg A$ контрарны друг другу, и множество $\{ A, \neg A \}$ называется контрарной парой.

Отметим, что дизъюнкт есть тавтология, если он содержит контрарную пару.

Определение 25: Правило резолюций состоит в следующем:

Для любых двух дизъюнктов C_1 и C_2, если существует литерал L_1 в C_1, который контрарен литералу L_2 в C_2, то вычеркнув L_1 и L_2 из C_1 и C_2 соответственно и построив дизъюнкцию оставшихся дизъюнктов, получим резолюцию (резольвенту) C_1 и C_2.

Пример 9: рассмотрим следующие дизъюнкты:

C_1: $P \vee R$,

C_2: $\neg P \vee Q$.

Дизъюнкт C_1 имеет литерал P, который контрарен литералу $\neg P$ в C_2. Следовательно, вычеркивая P и $\neg P$ из C_1 и C_2 соответственно, построим дизъюнкцию оставшихся дизъюнктов R и Q и получим резольвенту $R \vee Q$.

Важным свойством резольвенты является то, что любая резольвента двух дизъюнктов C_1 и C_2 есть логическое следствие C_1 и C_2. Это устанавлисвается в следующей теореме.

Теорема 4. Пусть даны два дизъюнкта C_1 и C_2. Тогда резольвента C дизъюнктов C_1 и C_2 есть логическое следствие C_1 и C_2.

Если есть два единичных дизъюнкта, то их резольвента, если она существует, есть пустой дизъюнкт □. Более существенно, что для невыполнимого множества дизъюнктов многократным применением правила резолюций можно породить □.

Определение 26: Пусть S – множество дизъюнктов. Резолютивный вывод C из S есть такая конечная последовательность C_1, C_2,..., C_k дизъюнктов, что каждый C_i или принадлежит S или является резольвентой дизъюнктов, предшествующих C_i, и C_k=C. Вывод □ из S называется опровержением (или доказательством невыполнимости) S.

Пример 11. Рассмотрим множество S:

1. $\neg P \vee Q$,

2. $\neg Q$,

3. P.

Из 1 и 2 получим резольвенту

4. $\neg P$.

Из 4 и 3 получим резольвенту

5. □.

Так как □ получается из S применениями правила резолюций , то согласно теореме 4 □ есть логическое следствие S, следовательно S невыполнимо.

Метод резолюций является наиболее эффективным в случае применения его к множеству Хорновских дизъюнктов.

Определение 27: Фразой называется дизъюнкт, у которого негативные литералы размещаются после позитивных литералов в конце дизъюнкта.

Пример 11: $P_1 \vee P_2 \vee ... P_n \vee \neg N_1 \vee \neg N_2 ... \vee \neg N_m$

Определение 28: Фраза Хорна это фраза, содержащая только один позитивный литерал.

Пример 12: преобразовать фразу Хорна в обратную импликацию.

$P \vee \neg N_1 \vee \neg N_2 ... \vee \neg N_m$

$\neg N_1 \vee \neg N_2 ... \vee \neg N_m = \neg (N_1 \wedge N_2 \wedge ... \wedge N_m)$

$P \leftarrow (N_1 \wedge N_2 \wedge ... \wedge N_m)$

$P \leftarrow N_1, N_2, ... N_m$

При представлении дизъюнктов фразами Хорна негативные литералы соответствуют гипотезам, а позитивный литерал представляет заключение. Единичный позитивный дизъюнкт представляет некоторый факт, то есть заключение, не зависящее ни от каких гипотез. Часто задача состоит в том, что надо проверить некоторую формулу, называемую целью, логически выведенную из множества правил и фактов. Резолюция является методом доказательства от противного: исходя из фактов, правил и отрицания цели, приходим к противоречию (пустому дизъюнкту).

2.4.4 Правило унификации в логике предикатов.

Правило резолюций предполагает нахождение в дизъюнкте литерала, контрарного литералу в другом дизъюнкте. Для дизъюнктов логики высказываний это очень просто. Для дизъюнктов логики предикатов процесс усложняется, так как дизъюнкты могут содержать функции, переменные и константы.

Пример 12. Рассмотрим дизъюнкты:

C_1*:* $P(y) \vee Q(y)$,

C_2*:* $\neg P(f(x)) \vee R(x)$.

Не существует никакого литерала в C_1*, контрарного какому-либо литералу в* C_2*. Однако, если подставить f(a) вместо y в* C_1 *и a вместо x в* C_2*, то исходные дизъюнкты примут вид:*

C_1'*:* $P(f(a)) \vee Q(f(a))$,

C_2'*:* $\neg P(f(a)) \vee R(a)$.

Так как P(f(a)) контрарен $\neg P(f(a))$*, то можно получить резольвенту*

C_3'*:* $Q(f(a)) \vee R(a)$.

В общем случае, подставив f(x) вместо y в C_1*, получим*

C_1''*:* $P(f(x)) \vee Q(f(x))$.

Литерал P(f(x)) в C_1'' *контрарен литералу* $\neg P(f(x))$ *в* C_2*. Следовательно, можно получить резольвенту*

C_3*:* $Q(f(x)) \vee R(x)$.

Таким образом, если подставлять подходящие термы вместо переменных в исходные дизъюнкты, можно порождать новые дизъюнкты. Отметим, что дизъюнкт C_3 из примера 13 является наиболее общим дизъюнктом в том смысле, что все другие дизъюнкты, порожденные правилом резолюции будут частным случаем данного дизъюнкта.

Определение 29: Подстановка θ*– это конечное множество вида* $\{t_1/v_1,\ldots,t_n/v_n\}$*, где каждая* v_i *– переменная, каждый* t_i *– терм, отличный от* v_i*, все* v_i *различны.*

Определение 30: Подстановка θ *называется унификатором для множества* $\{E_1,\ldots, E_k\}$ *тогда и только тогда, когда* $E_1\theta=E_2\theta=\ldots E_k\theta$*. Множество* $\{E_1,\ldots, E_k\}$ *унифицируемо, если для него существует унификатор.*

Прежде чем применить правило резолюции в исчислении предикатов переменные в литералах необходимо унифицировать.

Унификация производится при следующих условиях:

1. Если термы константы, то они унифицируемы тогда и только тогда, когда они совпадают.
2. Если в первом дизъюнкте терм переменная, а во втором константа, то они унифицируемы, при этом вместо переменной подставляется константа.
3. Если терм в первом дизъюнкте переменная и во втором дизъюнкте терм тоже переменная, то они унифицируемы.

4. Если в первом дизъюнкте терм переменная, а во втором - употребление функции, то они унифицируемы, при этом вместо переменной подставляется употребление функции.
5. Унифицируются между собой термы, стоящие на одинаковых местах в одинаковых предикатах.

Пример 13. Рассмотрим дизъюнкты:

1. Q(a, b, c) и Q(a, d, l). Дизъюнкты не унифицируемы.

2. Q(a, b, c) и Q(x, y, z). Дизъюнкты унифицируемы. Унификатор - Q(a, b, c).

Определение 31: Унификатор σ для множества $\{E_1,\ldots, E_k\}$ будет наиболее общим унификатором тогда и только тогда, когда для каждого унификатора θ для этого множества существует такая подстановка λ, что $\theta=\sigma \circ \lambda$, то есть θ является композицией подстановок σ и λ.

Определение 32: Композицией подстановок σ и λ есть функция $\sigma \circ \lambda$, определяемая следующим образом $(\sigma \circ \lambda)[t]=\sigma[\lambda[t]]$, где t – терм, σ и λ - подстановки, а $\lambda[t]$ – терм, который получается из t путем применения к нему подстановки λ.

Определение 33: Множество рассогласований непустого множества дизъюнктов $\{E_1,\ldots, E_k\}$ получается путем выявления первой (слева) позиции, на которой не для всех дизъюнктов из E стоит один и тот же символ, и выписывания из каждого дизъюнкта терма, который начинается с символа, занимающего данную позицию. Множество термов и есть множество рассогласований в E.

Пример 14. Рассмотрим дизъюнкты:

{P(x, f(y, z)), P(x, a), P(x, g(h(k(x))))}.

Множество рассогласований состоит из термов, которые начинаются с пятой позиции и представляет собой множество {f(x, y), a, g(h(k(x)))}.

Алгоритм унификации для нахождения наиболее общего унификатора.

Пусть E – множество дизъюнктов, D – множество рассогласований, k – номер итерации, σ_k наиболее общий унификатор на k-ой итерации.

Шаг 1. Присвоим k=0, σ_k=e (пустой унификатор), E_k=E.

Шаг 2. Если для E_k не существует множества рассогласований D_k, то остановка: σ_k – наиболее общий унификатор для E. Иначе найдем множество рассогласований D_k.

Шаг 3. Если существуют такие элементы v_k и t_k в D_k, что v_k переменная, не входящая в терм t_k, то перейдем к шагу 4. В противном случае остановка: E не унифицируемо.

Шаг 4. Пусть $\sigma_{k+1}=\sigma_k \{ t_k / v_k\}$, заменим во всех дизъюнктах E_k t_k на v_k.

Шаг5. K=k+1. Перейти к шагу 2.

Пример 16. Рассмотрим дизъюнкты:

E={P(f(a), g(x)), P(y, y)}.

1. *E_0=E, k=0, σ_0=e.*
2. *D_0={f(a),y}, v_0=y, t_0=f(a).*

3. $\sigma_1 = =\{f(a)/y\}$, $E_1=\{P(f(a), g(x)), P(f(a), f(a))\}$.

4. $D_1=\{g(x),f(a)\}$.

5. *Нет переменной в множестве рассогласований* D_1.

Следовательно, алгоритм унификации завершается, множество E – не унифицируемо.

2.4.5 Метод резолюций в исчислении предикатов

С помощью унификации можно распространить правило резолюций на исчисление предикатов. При унификации возникает одна трудность: если один их термов есть переменная x, а другой терм содержит x, но не сводится к x, унификация невозможна. Проблема решается путем переименования переменных таким образом, чтобы унифицируемые дизъюнкты не содержали одинаковых переменных.

Определение 34: Если два или более литерала (с одиниковым знаком) дизъюнкта C имеют наиболее общий унификатор σ*, то* $C\sigma$ *- называется склейкой C.*

Пример 15. Рассмотрим дизъюнкты:

Пусть $C= P(x) \vee P(f(y)) \vee \neg Q(x)$.

Тогда 1 и 2 литералы имеют наиболее общий унификатор $\sigma = \{f(y)/x\}$. *Следовательно,* $C\sigma=P(f(y)) \vee \neg Q(f(y))$ *есть склейка C.*

Определение 35: Пусть C_1 *и* C_2 *– два дизъюнкта, которые не имеют никаких общих переменных. Пусть* L_1 *и* L_2 *- два литерала в* C_1 *и* C_2 *соответственно. Если* L_1 *и* $\neg L_2$ *имеют наиболее общий унификатор* σ*, то дизъюнкт* $(C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma)$ *называется резольвентой* C_1 *и* C_2.

Пример 16:Пусть $C_1= P(x) \vee Q(x)$ *и* $C_2= \neg P(a) \vee R(x)$. *Так как x входит в* C_1 *и* C_2, *то мы заменим переменную в* C_2 *и пусть* $C_2= \neg P(a) \vee R(y)$. *Выбираем* $L_1= P(x)$ *и* $L_2=\neg P(a)$. L_1 *и* L_2 *имеют наиболее общий унификатор* $\sigma=\{a/x\}$. *Следовательно,* $Q(a) \vee R(y)$ *– резольвента* C_1 *и* C_2.

Пример 17:Доказать, что формула R(a, z) выводима из множества дизъюнктов $S=\{\neg P(x, f(x)) \vee R(x, f(x), g(x)), Q(x, g(x)) \vee R(x, f(x), g(x)), \neg Q(x, g(x)) \wedge P(x, f(x))\}$.

Пусть $C_1=\neg P(x, f(x)) \vee R(x, f(x)$, $C_2= Q(x, g(x)) \vee R(x, f(x)$, $C_3=\neg Q(x, g(x)) \vee P(x, f(x))$. *Добавим к множеству S единичный дизъюнкт* $C_4=\neg R(a, z)$.

Так как в дизъюнктах C_1 , C_2 , C_3 *есть одинаковая переменная x, то заменим её в* C_2 *на y, а в* C_3 *на u. Таким образом, решение исходной задачи сводится к доказательству противоречивости следующего множества дизъюнктов:*

$C_1=\neg P(x, f(x)) \vee R(x, f(x))$;

$C_2= Q(y, g(y)) \vee R(y, f(y))$;

$C_3=\neg Q(u, g(u)) \vee P(u, f(u))$;

$C_4=\neg R(a, z)$.

Найдём резольвенту C_5 *дизъюнктов* C_1 *и* C_3 *и добавим её к множеству дизъюнктов, для чего произведём унификацию переменных x и u, таким образом, что u заменим на x, и получим* $C_5= R(x, f(x)) \vee \neg Q(x, g(x))$.

Найдём резольвенту C_6 дизъюнктов C_2 и C_5 и добавим её к множеству дизъюнктов, для чего произведём унификацию переменных y и x, таким образом, что y заменим на x, и получим $C_6 = R(x, f(x)) \vee R(x, f(x)) = R(x, f(x))$.

Найдём резольвенту C_7 дизъюнктов C_2 и C_6 и добавим её к множеству дизъюнктов, для чего произведём замену переменной x на константу a, а переменную z заменим на функцию f(a), таким образом получим $C_7 = \square$, то есть пустой дизъюнкт.

Алгоритм метода резолюций.

Шаг 1. Если в S есть пустой дизъюнкт, то множество противоречиво (невыполнимо), иначе перейти к шагу 2.

Шаг 2. Найти в исходном множестве S такие дизъюнкты или склейки дизъюнктов C_1 и C_2, которые содержат унифицируемые контрараные литералы $L_1 \in C_1$ и $L_2 \in C_2$. Если таких дизъюнктов нет, то исходное множество выполнимо, иначе перейти к шагу 3.

Шаг 3. Вычислить резольвенту C_1 и C_2 и добавить ее в множество S. Перейти к шагу 1.

3 Введение в язык логического программирования ПРОЛОГ

3.1 Теоретические основы

Язык Пролог объединяет два подхода: логический и процедурный.

По мнению Дж. Робинсона в основе идеи логического программирования лежит принцип описания задачи при помощи совокупности утверждений на некотором формальном логическом языке и получение решения при помощи вывода в некоторой формальной системе. Основой языка Пролог является логика предикатов первого порядка.

Программа на Прологе включает в себя постановку задачи в виде множества фраз Хорна (раздел clauses) и описание цели (раздел goal), - формулировку теоремы, которую нужно доказать, исходя из множества правил и фактов, содержащихся в этой постановке.

Процесс поиска доказательства основан на методе линейной резолюции (дизъюнкты подбираются в порядке их следования в тексте программы).

Проиллюстрируем принцип логического программирования на простом примере: запишем известный метод вычисления наибольшего общего делителя двух натуральных чисел – алгоритм Евклида в виде Хорновских дизъюнктов. При этом примем новую форму записи фразы Хорна, например $\neg P \wedge \neg Q \wedge \neg R \wedge S$ будем записывать как *S: - P, Q, R*. Тогда алгоритм Евклида можно записать в виде трех фраз Хорна:

1. *NOD (x, x, x): -.*
2. *NOD (x, y, z): - B (x, y), NOD (f (x, y), y, z).*
3. *NOD (x, y, z): -B (y, x), NOD (x, f (y, x), z).*

Предикат *NOD* – определяет наибольший общий делитель *z* для натуральных чисел *x* и *y*, предикат *B* – определяет отношение «больше», функция *f* – определяет операцию вычитания. Если мы заменим предикат *B* и функцию *f* обычными символами, то фразы примут вид:

1. *NOD (x, x, x): -.*
2. *NOD (x, y, z): - x>y, NOD ((x- y), y, z).*
3. *NOD (x, y, z): -y>x, NOD ((x, y- x), z).*

Для вычисления наибольшего общего делителя двух натуральных чисел, например 4 и 6, добавим к описанию алгоритма четвертый дизъюнкт:

4. *□: - NOD (4, 6, z).*

Последний дизъюнкт – это цель, которую мы будем пытаться вывести из первых трех дизъюнктов.

3.2 Основы языка программирования Пролог

3.2.1 Общие положения

Язык программирования Пролог (PROgramming LOGic) предполагает получение решения задачи при помощи логического вывода из ранее известных фактов. Программа на языке Пролог не является последовательностью действий – она представляет собой набор фактов и

правил, обеспечивающих получение логических заключений из данных фактов. Поэтому Пролог считается *декларативным* языком программирования.

Пролог базируется на *фразах (предложениях) Хорна*, являющихся подмножеством формальной системы, называемой *логикой предикатов.*

Пролог использует упрощенную версию синтаксиса логики предикатов, он прост для понимания и очень близок к естественному языку.

Пролог имеет механизм вывода, который основан на сопоставлении образцов. С помощью подбора ответов на запросы Пролог извлекает хранящуюся информацию. Пролог пытается ответить на запрос, запрашивая информацию, о которой уже известно, что она истинна.

Одной из важнейших особенностей Пролога является то, что он ищет не только ответ на поставленный вопрос, но и все возможные альтернативные решения. Вместо обычной работы программы на процедурном языке от начала и до конца, Пролог может возвращаться назад и просматривать все остальные пути при решении всех частей задачи.

Программист на Прологе описывает *объекты* и *отношения*, а также *правила,* при которых эти отношения являются истинными.

Объекты рассуждения в Прологе называются *термами* – синтаксическими объектами одной из следующих категорий:

- константы,
- переменные,
- функции (составные термы или структуры), состоящие из имени функции и списка аргументов-термов, имена функций начинаются со строчной буквы.

Константа в Прологе служит для обозначения имен собственных и начинается *со строчной буквы*.

Переменная в Прологе служит для обозначения объекта на который нельзя сослаться *по имени*.

Пролог не имеет оператора присваивания.

Переменные в Прологе инициализируются при сопоставлении с константами в фактах и правилах.

До инициализации переменная свободна, после присвоения ей значения она становится связанной. Переменная остается связанной только то время, которое необходимо для получения решения по запросу, затем Пролог освобождает ее и ищет другое решение.

Переменные в Прологе предназначены для установления соответствия между термами предикатов, действующих в пределах одной фразы (предложения), а не местом памяти для хранения данных. Переменная начинается с прописной буквы или знаков подчеркивания.

В Прологе программист свободен в выборе имен констант, переменных, функций и предикатов. Исключения составляют резервированные имена и числовые константы. Переменные от констант отличаются первой буквой имени: у констант она строчная, у переменных – заглавная буква или символ подчеркивания.

Область действия имени представляет собой часть программы, где это имя имеет один и тот же смысл:

- для переменной областью действия является предложение (факт, правило или цель), содержащее данную переменную;
- для остальных имен (констант, функций или предикатов) – вся программа.

Специальным знаком «_» обозначается анонимная переменная, которая используется тогда, когда конкретное значение переменной не существенно для данного предложения. Анонимные переменные не отличаются от обычных при поиске соответствий, но не принимают значений и не появляются в ответах. *Различные вхождения знака подчеркивания означают различные анонимные переменные.*

Отношения между объектами в Прологе называются фактами. Факт соответствует фразе Хорна, состоящей из одного положительного литерала.

Факт – это простейшая разновидность предложения Пролога.

Любой факт имеет соответствующее значение истинности и определяет отношение между термами.

Факт является простым предикатом, который записывается в виде функционального терма, состоящего из имени отношения и объектов, заключенных в круглые скобки, например:

мать(мария, анна).

отец(иван, анна).

Точка, стоящая после предиката, указывает на то, что рассматриваемое выражение является фактом.

Вторым типом предложений Пролога является вопрос или *цель. Цель* – это средство формулировки задачи, которую должна решать программа. Простой вопрос (цель) синтаксически является разновидностью факта, например:

Цель: мать (мария, юлия).

В данном случае программе задан вопрос, является ли мария матерью юлии. Если необходимо задать вопрос, кто является матерью юлии, то цель будет иметь следующий вид:

Цель: мать(X, юлия).

Сложные цели представляют собой конъюнкцию простых целей и имеют следующий вид:

Цель: Q_1, Q_2,…,Q_n, где запятая обозначает операцию конъюнкции, а Q_1, Q_2,…,Q_n – подцели главной цели.

Конъюнкция в Прологе истинна только при истинности всех компонент, однако, в отличие от логики, в Прологе учитывается *порядок оценки истинности компонент* (слева направо).

Пример 18.

Пусть задана семейная БД при помощи перечисления родительских отношений в виде списка фактов:

мать(мария, анна).

мать(мария, юлия).

мать(анна, петр).

отец(иван, анна).

отец(иван, юлия).

Тогда вопрос, является ли иван дедом петра, можно задать в виде следующей цели:

Цель: отец(иван, X), мать(X, петр).

На самом деле БД Пролога включает не только факты, но и правила. Факты и правила представляют собой не множество, а список. Для получения ответа БД просматривается по порядку, то есть в порядке следования фактов и предикатов в тексте программы.

Цель достигнута, если в БД удалось найти факт или правило, который (которое) удовлетворяет предикату цели, то есть превращает его в истинное высказывание. В нашем примере первую подцель удовлетворяют факты *отец(иван, анна). и отец(иван, юлия).* Вторую подцель удовлетворяет факт *мать(анна, петр).* Следовательно, главная цель удовлетворена, переменная *X* связывается с константой *анна*.

Третьим типом предложения является *правило*. Правило позволяет вывести один факт из других фактов. Иными словами, правило – это заключение, для которого известно, что оно истинно, если одно или несколько других найденных заключений или фактов являются истинными.

Правила – это предложения вида

H: - P_1, P_2,..., P_n.

Символ «: -» читается как «если», предикат H называется заключением, а последовательность предикатов P_1, P_2,..., P_n называется посылками. Приведенное правило является аналогом хорновского дизъюнкта $\neg P_1 \vee \neg P_2, ..., \vee \neg P_n \vee H$. Заключение истинно, если истинны все посылки. В посылках переменные связаны квантором существования, а в заключении - квантором всеобщности.

Пример 19.

Добавим в БД примера 18 правила, задающие отношение «дед»:

мать(мария, анна).

мать(мария, юлия).

мать(анна, петр).

отец(иван, анна).

отец(иван, юлия).

дед (X, Y): - отец(X, Z), мать(Z, Y).

дед (X, Y): - отец(X, Z), отец(Z, Y).

Тогда вопрос, является ли иван дедом петра, можно задать в виде следующей цели:

Цель: дед(иван, петр).

Правила - самые общие предложения Пролога, факт является частным случаем правила без правой части, а цель – правило без левой части.

Все предложения для одного предиката связаны между собой отношением «или».

Очень часто правила в Прологе являются рекурсивными. Например, для нашей семейной БД предикат «предок» определяется рекурсивно:

предок(x, y): - мать(x, y).

предок(x, y): - отец(x, y).

предок(x, y): - мать(x, z), предок(z, y).

предок(x, y): - отец (x, z), предок(z, y).

Рекурсивное определение предиката обязательно должно содержать нерекурсивную часть, иначе оно будет логически некорректным и программа зациклится. Чтобы избежать зацикливания, следует также позаботиться о порядке выполнения предложений, поэтому практически полезно, а порой и необходимо придерживаться принципа: *«сначала нерекурсивные выражения».*

Программа на Прологе - это конечное множество предложений.

3.2.2 Использование дизъюнкции и отрицания

Чистый Пролог разрешает применять в правилах и целях только конъюнкцию, однако, язык, используемый на практике, допускает применение дизъюнкции и отрицания в телах правил и целях. Для достижения цели, содержащей дизъюнкцию, Пролог–система сначала пытается удовлетворить левую часть дизъюнкции, а если это не удается, то переходит к поиску решения для правой части дизъюнкции. Аналогичные действия производятся при выполнении тела правил, содержащих дизъюнкцию. Для обозначения дизъюнкции используется символ « ; ».

В Прологе отрицание имеет имя *«not»* и для представления отрицания какого-либо предиката *P* используется запись *not(P). Цель not(P) достижима тогда и только тогда, когда не удовлетворяется предикат (цель) P.* При этом переменным значения не присваиваются. В самом деле, если достигается *P,* то не достигается *not(P),* значит надо стереть все присваивания, приводящие к данному результату. Наоборот, если *P* не достигается, то переменные не принимают никаких значений.

3.2.3 Унификация в Прологе

Общий принцип выполнения программ на Прологе прост: производится поиск ответа на вопросы, задаваемые БД, состоящей из фактов и правил, то есть проверяется соответствие предикатов вопроса предложениям из БД. Это частный случай метода резолюций.

Отметим, что в Прологе не проводится синтаксического различия между предикатом и функцией (составным термом), а также между нечисловой константой и функцией без аргументов. Следовательно, суть действия состоит в том, что ищется попарное соответствие между термами, один из которых является целью, а другой принадлежит БД.

Установление соответствия между термами является основной операцией при вычислении цели. Она осуществляется следующим образом: на каждом шаге выбирается очередной терм и отыскивается соответствующее выражение в БД. При этом переменные получают или

теряют значения. Этот процесс можно описать в терминах текстуальных подстановок: « подставить терм t вместо переменной Y». *Свободными* переменными в Прологе называются переменные, которым не были присвоены значения, а все остальные переменные называются *связанными* переменными. Переменная становится связанной только во время унификации, переменная вновь становится свободной, когда унификация оказывается неуспешной или цель оказывается успешно вычисленной. В Прологе присваивание значений переменным выполняется внутренними подпрограммами унификации. Переменные становятся свободными, как только для внутренних подпрограмм унификации отпадает необходимость связывать некоторое значение с переменной для выполнения доказательства подцели.

3.2.4 Правила унификации

1. Если x и y-константы, то они унифицируемы, только если они равны.
2. Если x- константа или функция, а Y-переменная, то они унифицируемы, при этом Y принимает значение x .
3. Если x и y -функции, то они унифицируемы тогда и только тогда, когда у них одинаковые имена функций (функторы) и набор аргументов и каждая пара аргументов функций унифицируемы.

Есть особый предикат «=», который используется в Прологе для отождествления двух термов. Использование оператора «=» поможет лучше понять процесс означивания переменной. В Прологе оператор «=» интерпретируется как оператор присваивания или как оператор проверки на равенство в зависимости от того, являются ли значения термов свободными или связанными.

Пример 20: X=Y, если X и Y – связанные переменные, то производится проверка на равенство, например: если X=5 и Y=5, то результат ДА (истина); если X=6 а Y=5, то результат НЕТ(ложь). Если одна из переменных X или Y – свободная, то ей будет присвоено значение другой переменной, для Турбо-Пролога несущественно слева или справа от знака «=» стоит связанная переменная.

Оператор «=» ведет себя точно так, как внутренние подпрограммы унификации при сопоставлении целей или подцелей с фактами и правилами программы.

3.2.5 Вычисление цели. Механизм возврата

Каноническая форма цели (вопроса) является конъюнкцией атомарных предикатов, то есть последовательностью подцелей, разделенных запятыми:

$Q=Q_1, Q_2,..., Q_n$.

Пролог пытается вычислить цель при помощи унификации термов предикатов подцелей с соответствующими элементами в фактах и заголовках правил. Поиск ответа на вопрос напоминает поиск пути в лабиринте: следует поворачивать налево в каждой развилке лабиринта до тех пор, пока не

попадете в тупик. В этом случае следует вернуться к последней развилке и повернуть направо, после чего опять следует повернуть направо и так далее. Унификация выполняется слева направо, как и поиск пути в лабиринте.

Некоторые подцели при унификации с некоторыми фактами или правилами могут оказаться неуспешными, поэтому Прологу требуется способ запоминания точек отката, в которых он может продолжить альтернативные поиски решения. Прежде чем реализовать один из возможных путей вычисления подцели, Пролог фактически помещает в программу указатель, который определяет точку, в которую может быть выполнен возврат, если текущая попытка поиска цели будет неудачной.

Если некоторая подцель оказывается неуспешной, то Пролог возвращается влево к ближайшей точке возврата. С этой точки Пролог начинает попытку найти другое решение для неуспешной цели. До тех пор, пока следующая цель на данном уровне не будет успешной, Пролог будет повторять возврат к ближайшей точке возврата. Эти попытки выполняются внутренними подпрограммами унификации и механизмом возврата.

Замечание: единственным способом освободить переменную, связанную в предложении является откат при поиске с возвратом.

Алгоритм вычисления цели – частный случай правила резолюции применительно к дизъюнктам Хорна. Вопрос Q является правилом без заголовка, аналогом выражения $\neg Q$. Пусть D – база данных (множество дизъюнктов). На вопрос Q, следует найти такую подстановку σ, для которой множество $\sigma[D \cup (\neg Q)]$ невыполнимо. Стратегия выбора очередной пары дизъюнктов для резолюции здесь очень проста: подцели и предложения просматриваются в текстуальном порядке.

Пример 21: пусть есть БД семья:

1. *мать(мария, анна).*
2. *мать(мария, юлия).*
3. *мать(анна, петр).*
4. *отец(иван, анна).*
5. *отец(иван, юлия).*
6. *дед (X, Y): - отец(X, Z), мать(Z, Y).*
7. *дед (X, Y): - отец(X, Z), отец(Z, Y).*
8. *бабка (X, Y): - мать(X, Z), мать(Z, Y).*
9. *бабка (X, Y): - мать(X, Z), отец(Z, Y).*

Зададим сложную цель:

Q1, Q2 = отец(X, Y), мать(мария, Y).

Подцель Q1= отец(X,Y) соответствует четвертому предложению БД :отец (иван,анна). Это дает подстановку σ_1*={X=иван,Y=анна}. Затем найденная подстановка применяется к* σ_1*[Q2]= мать(мария, анна). Этой подцели соответствует 1 предложение БД, что дает пустую подцель по правилу резолюции, следовательно ответ найден: X= иван, Y= анна.*

Для получения нового ответа в БД ищется новая унификация для $\sigma_1[Q_2]$. *Так как в БД нет соответствующего предложения, то вычисления прекращаются, система вновь рассматривает последовательность* Q_1, Q_2 *и для* Q_1 *ищется новая унификация в БД, начиная с пятого предложения. Это и есть возврат. Пятое предложение сразу же дает желаемую унификацию с подстановкой* σ_2*={X=иван,Y=юлия}. Вновь найденная подстановка применяется к* $\sigma_1[Q_2]$*= мать(мария, юлия). Этой подцели соответствует второе предложение БД. Далее указанная процедура повторяется и в итоге имеем:*

Цель: отец(X, Y), мать(мария, Y).

2 решения: X=иван, Y=анна

X=иван, Y=юлия.

Это описание объясняет, как работает утилита TestGoal в Visual Prolog.

При реализации механизма возврата выполняются следующие правила:

1. Подцели вычисляются слева-направо.
2. Предложения при вычислении подцели проверяются в текстуальном порядке, то есть сверху-вниз.
3. Если подцель сопоставима с заголовком правила, то должно быть вычислено тело этого правила, при этом тело правила образует новое подмножество подцелей для вычисления.
4. Цель считается успешно вычисленной, когда найден соответсвующий факт для каждой подцели.

Если в Visual Prolog создать программу для автономного исполнения (то есть создать свой project-файл для разрабатываемой программы), то поиск цели в ней будет вестись так же, как для внутренней цели в PDC Prolog, то есть до первого успешного решения.

3.2.6 Управление поиском решения

Встроенный в Пролог механизм поиска с возвратом может привести к поиску ненужных решений, в результате чего снижается эффективность программы в случае, если надо найти только одно решение. В других случаях бывает необходимо продолжить поиск, даже если решине найдено.

Пролог обеспечивает два встроенных предиката, которые дают возможность управлять механизмом поиска с возвратом: предикат *fail* – используется для инициализации поиска с возвратом и предикат *отсечения !* – используется для запрета возврата.

Предикат fail всегда имеет ложное значение!

Пример 22: Использование предиката fail. Для примера 21 можно добавить правило для печати всех матерей, которые есть в БД:

печать_матерей:-мать(X,Y), write(X,” есть мать”,Y),nl,fail.

goal

печать_матерей.

В результате будет выдано 3 решения:

X=мария, Y= анна.

X=мария, Y= юлия.

X=анна, Y= петр.

Отсечение так же, как и *fail* помещается в тело правила. ***Однако, в отличие от fail предикат отсечения имеет всегда истинное значение.***

При этом выполняется обращение к другим предикатам в теле правила, следующим за отсечением. Следует иметь в виду, что невозможно произвести возврат к предикатам, расположенным в теле правила перед отсечением, а также невозможен возврат к другим правилам данного предиката.

Существует только два случая применения предиката отсечения:

1. Если заранее известно, что определенные посылки никогда не приведут к осмысленным решениям – это так называемое «зеленое отсечение».
2. Если отсечения требует сама логика программы для исключения альтернативных подцелей – это так называемое «красное отсечение».

3.2.7 Процедурность Пролога

Пролог – декларативный язык. Описывая задачу в терминах фактов и правил, программист предоставляет Прологу самому искать способ решения. В процедурных языках программист должен сам писать процедуры и функции, которые подробно «объясняют» компьютеру, какие шаги надо сделать для решения задачи.

Тем не менее, рассмотрим Пролог с точки зрения процедурного программирования:

1. Факты и правила можно рассматривать как определения процедур.
2. Использование правил для условного ветвления программы. Правило, в отличие от процедуры, позволяет задавать множество альтернативных определений одной и той же процедуры. Поэтому, правило можно считать аналогом оператора *case* в Паскале.
3. В правиле может быть выполнено сравнение, как в условных операторах.
4. Отсечение можно считать аналогом *go to*.
5. Возврат вычисленного значения производится аналогично процедурам. В Прологе это делается путем связывания свободных переменных при сопоставлении цели с фактами и правилами.

3.2.8 Структура программ Пролога

Программа, написанная на Прологе, состоит из пяти основных разделов: раздел описания доменов, раздел базы данных, раздел описания предикатов, раздел описания предложений и раздел описания цели. Ключевые слова *domains, constants, database (facts), predicates, clauses и goal*

отмечают начала соответствующих разделов. Назначение этих разделов таково:

- раздел *domains* содержит определения доменов, которые описывают различные типы данных, используемых в программе;
- раздел *constants* используется для объявления символических констант, используемых в программе;
- раздел *database (facts)* содержит описания предикатов внутренней базы данных Пролога, если программа такой базы данных не требует, то этот раздел может быть опущен;
- раздел *predicates* служит для описания предикатов, не принадлежащих внутренней базе данных;
- в раздел *clauses* заносятся факты и правила самой программы;
- в разделе *goal* на языке Пролог формулируется назначение создаваемой программы. Составными частями при этом могут являться некие подцели, из которых формируется единая цель программы.

В Visual Prolog разрешает объявление разделов domains, facts, predicates, clauses как глобальных разделов, то есть с ключевым словом global.

Пролог имеет следующие встроенные типы доменов:

Тип данных	**Ключевое слово**	**Диапазон значений**	**Примеры использования**
Символы	char	Все возможные символы	‘a’, ’b’, ‘#’, ‘B’, ‘%’
Целые числа	integer byte word dword	От –32768 до 32767 От 0 до 255 От 0 до 65535 От 0 до	-63, 84, 2349
Действительные числа	real short ushort long ulong unsigned	От +1E-307 до +1E308 16 битов со знаком 16 битов без знака 32 бита со знаком 32 бита без знака 16 или 32 бита без знака	360, - 8324, 1.25E23, 5.15E-9
Строки	string	Последовательность символов (не более 250)	«today», «123», «school_day»
Символические имена	symbol	1.Последовательность букв, цифр, символов подчеркивания; первый символ – строчная буква. 2. Последовательность	flower, school_day «string and

		любых символов, заключенная в кавычки.	symbol»
Ссылочный тип	ref		
Файлы	file	Допустимое в DOS имя файла	mail.txt, LAB.PRO

Если в программе необходимо использовать новые домены данных, то они должны быть описаны в разделе *domains.*

Пример 22:

```
domains
number=integer
name, person=symbol.
```

Различие между *symbol* и *string* - в машинном представлении и выполнении, синтаксически они не различимы.

Visual Prolog выполняет автоматическое преобразование типов между доменами *string* и *symbol.* Однако, по принятому соглашению, символическую строку в двойных кавычках нужно рассматривать как *string,* а без кавычек – как *symbol:*

Symbol - имена, начинающиеся с символа нижнего регистра и содержащие только символы, цифры, и символы подчеркивания.

String – в двойных кавычках могут содержать любую комбинацию символов, кроме #0, который отмечает конец строки.

Visual Prolog поддерживает и другие типы стандартных доменов данных, например, для работы с внешними БД или объектами.

Предикаты описываются в разделе *predicates*. Предикат представляет собой строку символов, первым из которых является строчная буква. Предикаты могут не иметь аргументов, например «go» или «repeat». Если предикаты имеют аргументы, то они определяются при описании предикатов в разделе predicates:

Пример 23:

```
predicates
mother (symbol, symbol)
father (symbol, symbol).
```

Факты и правила определяются в разделе *clauses,* а вопрос к программе задается в разделе *goal* – в этом случае цель называется *внутренней целью.* Программа на Турбо-Прологе может не содержать раздел *goal,* в этом случае цель задается в процессе работы программы и является *внешней целью*. Для задания внешней цели в окне *Dialog* будет выведено приглашение *Goal.* После удовлетворения внешней цели программа на Турбо-Прологе не заканчивает свою работу, а просит ввести следующую цель, таким образом можно задать программе несколько различных целей. Если цель в программе является внутренней целью, то процесс вычисления остановится после первого ее успешного вычисления. Если цель в программе является внешней

целью, то процесс вычисления цели повторяется до тех пор, пока не будут найдены все успешные способы вычисления цели.

В Visual Prolog раздел goal в тексте программы является обязательным. Разница в режимах исполнения программы состоит в разном использовании утилиты Test Goal. Если утилита создается для запуска любой программы, то при этом ищутся все решения, если утилита создается для автономного запуска программы – то ищется одно решение.

3.2.9 Использование составных термов

В Прологе функциональный терм или предикат можно рассматривать как структуру данных, подобную записи в языке Паскаль. Терм, представляющий совокупность термов, называется составным термом или структурой. Составные структуры данных в Прологе объявляются в разделе *domains.* Если термы структуры относятся к одному и тому же типу доменов, то этот объект называется *однодоменной структурой данных*. Если термы структуры относятся к разным типам доменов, то такая структура данных называется *многодоменной структурой данных*. Использование доменной структуры упрощает структуру предиката.

Аргументами составного терма данных могут быть простые типы данных, составные термы или списки.

Синтаксически составной терм выглядит так же, как и предикат: у терма есть функтор и список аргументов, заключенных в круглые скобки.

Составной терм может быть унифицирован с простой переменной или составным объектом (при этом переменные могут быть использованы как часть внутренней структуры терма). Это означает, что составной объект можно использовать для того, чтобы передавать целый набор значений, как единый объект, а затем применять унификацию для их разделения.

Пример 24: Необходимо создать БД, содержащую сведения о книгах из личной библиотеки. Зададим составной терм с именем personal_library, имеющим следующую структуру: personal_library= book (title, author, publisher, year), и предикат collection (collector, personal_library). Терм book называется функтором структуры данных. Пример программы, использующей составные термы для описания личной библиотеки и поиска информации о книгах, напечатанных в 1990 году, выглядит следующим образом:

```
domains
collector, title, author, publisher = symbol
year = integer
personal_library = book (title, author, publisher, year)
predicates
collection (collector, personal_library)
clauses
collection (irina, book («Using Turbo Prolog», «Yin with Solomon», «Moscow, World», 1993)).
```

collection (petr, book («The art of Prolog», «Sterling with Shapiro», »Moscow, World», 1990)).

collection (anna, book («Prolog: a relation language and its applications», «John Malpas», »Moscow, Science», 1990)).

goal

collection (X, book(Y,_, _, 1990)

В данном случае переменная *Y* используется для унификации части составного терма.Если цель задать в виде:

collection (X, Z),Z=book(Y,_, _, 1990), то простая переменная *Z* унифицируется с сотавным термом *book.*

Представление данных часто требует наличия большого числа структур. В Прологе эти структуры должны быть описаны. Для более компактного описания структур данных в Прологе предлагается использование альтернативных описаний доменов.

Пример 25: Необходимо создать БД, содержащую сведения о книгах и аудиозаписях из личной библиотеки.

domains

person, title, author, artist, album, type = symbol

thing = book (title, author); record (artist, album, type)

predicates

owns (person, thing)

clauses

owns (irina, book («Using Turbo Prolog», «Yin with Solomon»)).

owns (petr, book («The art of Prolog», «Sterling with Shapiro»)).

owns (anna, book («Prolog: a relation language and its applications», «John Malpas»)).

owns (irina, record («Elton John», «Ice Fair», «popular»)).

owns (petr, record («Benny Goodman», «The King of Swing», »jazz»)).

owns (anna record («Madonna», «Madonna», «popular»»)).

goal

owns (X, record(_, _, «jazz»)

Visual Prolog позволяет конструировать многоуровневые составные термы. Например, в терме *record («Elton John», «Ice Fair», «popular»)* вместо фамилии артиста можно использовать новую структуру, которая будет описывать артиста более детально:

artist=art(family,name),

family,name=symbol,

при этом терм будет выглядеть следующим образом: record (art(“Elton”,”John”),«Ice Fair», «popular»).

3.2.10 Использование списков

Список является составной рекурсивной структурой данных, хотя явно и не объявляется как рекурсивная структура. Список – это упорядоченный набор объектов одного и того же типа. Элементами списка могут быть целые числа, действительные числа, символы, строки, символические имена и

структуры. Порядок расположения элементов в списке играет важную роль: те же самые элементы списка, упорядоченные иным способом, представляют уже совсем другой список.

Совокупность элементов списка заключается в квадратные скобки ([]), элементы друг от друга отделяются запятыми. Список может содержать произвольное число элементов, единственным ограничением является объем оперативной памяти. Количество элементов в списке называется его длиной. Список может содержать один элемент и даже не содержать ни одного элемента. Список, не содержащий элементов, называется пустым или нулевым списком.

Непустой список можно рассматривать как список, состоящий из двух частей: головы – первого элемента списка; и хвоста – остальной части списка. Голова является элементом списка, хвост является списком. Голова списка – это неделимое значение, хвост представляет собой список, составленный из того, что осталось от исходного списка в результате «отделения головы». Этот новый список обычно можно делить и дальше. Если список состоит из одного элемента, то его можно разделить на голову, которой будет этот самый элемент, и хвост, являющийся пустым списком.

Пустой список нельзя разделить на голову и хвост!

Операция деления списка на голову и хвост обозначается при помощи вертикальной черты (|):

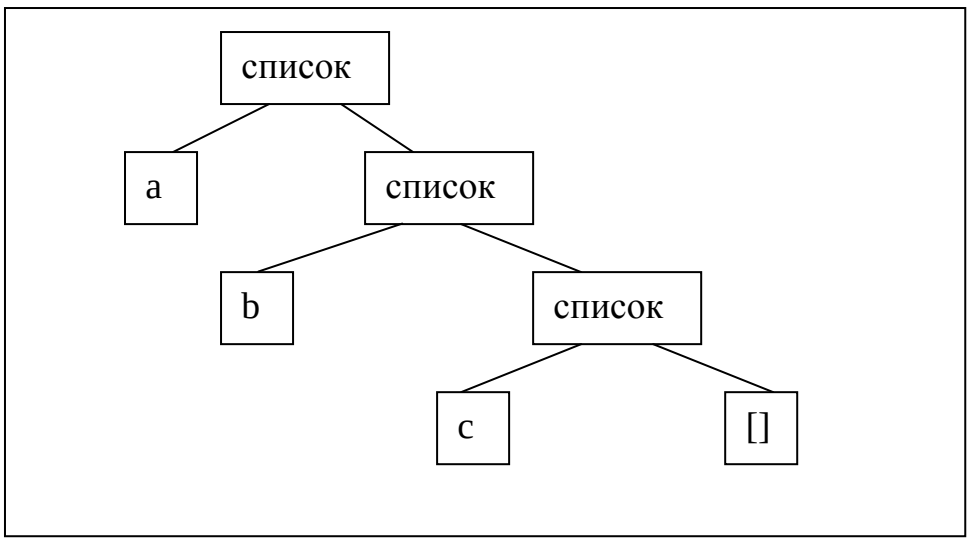
[Head | Tail].

Голова списка всегда имеет тип элемента списка, хвост списка – тип списка!

Head здесь является переменной для обозначения головы списка, переменная Tail обозначает хвост списка (для имен головы и хвоста списка пригодны любые допустимые Прологом имена).

Данная операция также присоединяет элемент в начало списка, например, для того, чтобы присоединить X к списку S следует написать *[X|S]*.

В концептуальном плане, список имеет структуру дерева, как и другие составные термы. Так, например, список *[a,b,c]* можно представить в виде структуры:



Отличительной особенностью описания списков является наличие звездочки (*) после имени домена элементов.

Пример 34: объявление списков, состоящих из элементов стандартных типов доменов или типа структуры.

```
domains
list1=integer*
list2=char*
list3=string*
list4=real*
list5=symbol*
personal_library = book (title, author, publisher, year)
list6= personal_library*
list7=list1*
list8=list5*
```

В первых пяти объявлениях списков в качестве элементов используются стандартные домены данных, в шестом типе списка в качестве элемента используется домен структуры personal_library, в седьмом и восьмом типе списка в качестве элемента используется ранее объявленный список.

Пример 35: демонстрация разделения списков на голову и хвост.

Список	*Голова*	*Хвост*
[1, 2, 3, 4, 5]	*1*	*[2, 3, 4, 5]*
[6.9, 4.3]	*6.9*	*[4.3]*
[cat, dog, horse]	*Cat*	*[dog, horse]*
[‘S’, ‘K’, ‘Y’]	*‘S’*	*[‘K’, ‘Y’]*
[«PIG»]	*«PIG»*	*[]*
[]	*Не определена*	*Не определен*

Visual Prolog допускает объявление и использвание составных списков. Составной список – это список, в котром используется более чем один тип элемента. Для создания такого списка, необходимо использовать структуры, так как домен может содержать более одного типа данных только для структуры.

Пример 36: объявление списка, который может содержать символы, целые числа или строки:

```
domains
llist=i(integer);c(char);s(string)
list=llistl*
```

Для применения списков в программах на Прологе необходимо описать домен списка в разделе *domains*, предикаты, работающие со списками необходимо описать в разделе *predicates*, задать сам список можно либо в разделе *clauses* либо в разделе *goal*.

Над списками можно реализовать различные операции: поиск элемента в списке, разделение списка на два списка, присоединение одного списка к

другому, удаление элементов из списка, сортировку списка, создание списка из содержимого БД и так далее.

3.2.11 Поиск элемента в списке

Поиск элемента в списке является очень распространенной операцией. Поиск представляет собой просмотр списка на предмет выявления соответствия между объектом поиска и элементом списка. Если такое соответствие найдено, то поиск заканчивается успехом, в противном случае поиск заканчивается неуспехом. Стратегия поиска при этом будет состоять в рекурсивном выделении головы списка и сравнении ее с объектом поиска.

Пример 37: поиск элемента в списке.

```
domains
list=integer*
predicates
member (integer, list)
clauses
member (Head, [Head |_ ]).
member (Head, [_ | Tail ]):- member (Head, Tail).
goal
member (3, [1, 4, 3, 2]).
```

Правило поиска может сравнить объект поиска и голову текущего списка, эта операция записана в виде факта предиката member. Этот вариант предполагает наличие соответствия между объектом поиска и головой списка. Отметим, что хвост списка в данном случае не важен, поэтому хвост списка присваивается анонимной переменной. Если объект поиска и голова списка различны, то в результате исполнения первого предложения будет неуспех, происходит возврат и поиск другого правила или факта, с которыми можно попытаться найти соответствие. Для этой цели служит второе предложение, которое выделяет из списка следующий по порядку элемент, то есть выделяет голову текущего хвоста, поэтому текущий хвост представляется как новый список, голову которого можно сравнить с объектом поиска. В случае исполнения второго предложения, голова текущего списка ставится в соответствие анонимной переменной, так как значение головы с писка в данном случае не играет никакой роли.

Процесс повторяется до тех пор, пока первое предложение даст успех, в случае установления соответствия, либо неуспех, в случае исчерпания списка. В представленном примере предикат find находит все совпадения объекта поиска с элементами списка. Для того, чтобы найти только первое совпадение следует модифицировать первое предложение следующим образом:

```
member (Head, [Head |_ ]):- !.
```

Отсечение отменяет действие механизма возврата, поэтому поиск альтернативных успешных решений реализован не будет.

3.2.12 Объединение двух списков

Слияние двух списков и получение, таким образом, третьего списка принадлежит к числу наиболее полезных при работе со списками операций. Обозначим первый список L1, а второй список - L2. Пусть L1 = [1, 2, 3], а L2 = [4, 5]. Предикат append присоединяет L2 к L1 и создает выходной список L3, в который он должен переслать все элементы L1 и L2. Весь процесс можно представить следующим образом:

1. Список L3 вначале пуст.
2. Элементы списка L1 пересылаются в L3, теперь значением L3 будет [1, 2, 3].
3. Элементы списка L2 пересылаются в L3, в результате чего тот принимает значение [1, 2, 3, 4, 5].

Тогда программа на языке Турбо-Пролог имеет следующий вид:

Пример 38: объединение двух списков.

```
domains
list=integer*
predicates
append (list, list, list)
clauses
append ( [], L2, L2).
append ([H|T1], L2, [H|T3 ]):- append (T1, L2, T3).
goal
append ( [1, 2, 3], [4, 5], L3).
```

Основное использование предиката *append* состоит в объединении двух списков, что делается при помощи задания цели вида *append ([1, 2, 3], [4, 5], L3)*. Поиск ответа на вопрос типа: *append (L1, [3, 4, 5], [1, 2, 3, 4, 5])* – сводится к поиску такого списка *L1=[1, 2]*, который при слиянии со списком *L2 = [3, 4, 5]* даст список *L3 = [1, 2, 3, 4, 5]*. При обработки *цели append (L1, L2, [1, 2, 3, 4, 5])* ищутся такие списки *L1* и *L2*, что их объединение даст список *L3 = [1, 2, 3, 4, 5]*.

3.2.13 Определение длины списка

Число элементов в списке можно подсчитать при помощи рекурсивных предикатов *count_list1 и count_list2*. Предикат *count_list1* ведёт подсчёт числа элементов в списке на прямом ходе рекурсии, начиная от головы списка, при этом первый параметр типа *byte* является текущим счётчиком, а второй - необходим для возвращения результата при выходе из рекурсии. Предикат *count_list2* ведёт подсчёт числа элементов в списке на обратном ходе рекурсии, начиная с последнего элемента, при этом параметр типа *byte* является текущим счётчиком и результатом одновременно. Два варианта решения задачи приводятся в примере 39.

Пример 39: определение длины списка.

```
domains
list=integer*
predicates
```

```
count_list1(list,byte,byte)
count_list2(list,byte)
clauses
count_list1([],N,N).
count_list1([_|T],N,M):- N1=N+1, count_list1(T,N1,M).
count_list2([],0).
count_list1([_|T],N):- count_list1(T,N1), N=N1+1.
goal
count_list1([0,-1,2,6,-9],0,N), count_list2([4,3,-8],K).
```

3.2.14 Поиск максимального и минимального элемента в списке

Найти максимальный или минимальный элемент в списке можно при помощи рекурсивных предикатов *max_list и min_list.* Предикат *max_list* ищет максимальный элемент в списке на прямом ходе рекурсии, начиная от головы списка, при этом первый параметр типа *integer* является текущим максимумом, а второй - необходим для возвращения результата при выходе из рекурсии. Предикат *min_list* ищет минимальный элемент в списке на обратном ходе рекурсии, начиная с последнего элемента, при этом параметр типа *integer* является текущим минимумом и результатом одновременно. Два варианта решения задачи приводятся в примере 40.

Пример 40: поиск максимального т минимального элемента в списке.

```
domains
list=integer*
predicates
max_list(list, integer, integer)
min_list(list, integer)
clauses
max_list ([],M,M).
max_list ([H|T],N,M):- H>N, max_list(T,H,M).
max_list ([H|T],N,M):- H<=N, max_list(T,N,M).
min_list ([H|[]],H).
min_list ([H|T],M):- min_list(T,M1), H>M1, M=H.
min_list ([H|T],M):- min_list(T,M1), H<=M1, M=M1.

goal
L=[1,5,3,-6,8,-4],L=[H|T],max_list(T,H,Max), min_list([4,3,-8,6],Min).
```

3.2.15 Сортировка списков

Сортировка представляет собой переупорядочение элементов списка определенным образом. Назначением сортировки является упрощение доступа к нужным элементам. Для сортировки списков обычно применяются три метода:

- метод перестановки,

- метод вставки,
- метод выборки.

Также можно использовать комбинации указанных методов.

Первый метод сортировки заключается в перестановке элементов списка до тех пор, пока он не будет упорядочен. Второй метод осуществляется при помощи неоднократной вставки элементов в список до тех пор, пока он не будет упорядочен. Третий метод включает в себя многократную выборку и перемещение элементов списка.

Второй из методов, метод вставки, особенно удобен для реализации на Прологе.

Пример 39: сортировка списков методом перестановки (пузырька).

```
domains
list=integer*
predicates
puz(list,list)
perest(list,list)
clauses
puz(L1,L2):-perest(L1,L3),!,puz(L3,L2).
puz(L1,L1).
perest([X,Y|T],[Y,X|T]):-X>Y.
perest([Z|T],[Z|T1]):-perest(T,T1).
goal
puz([1,3,4,5,2],L).]
```

Пример 40: сортировка списков методом вставки.

```
domains
list=integer*
predicates
insert_sort (list, list)
insert (integer, list, list)
asc_order (integer, integer)
clauses
insert_sort ( [], []).
insert_sort ([H1|T1], L2):- insert_sort (T1, T2),
                                    insert(H1, T2, L2).
insert (X, [H1| T1], [H1| T2]) :- asc_order (X, H1), !,
                                  insert (X, T1, T2).
insert (X, L1, [X| L1]).
asc_order (X, Y):- X>Y.
goal
insert_sort ([4, 7, 3, 9], L).
```

Для удовлетворения первого правила оба списка должны быть пустыми. Для того, чтобы достичь этого состояния, по второму правилу происходит рекурсивный вызов предиката *insert_sort*, при этом значениями *H1* последовательно становятся все элементы исходного списка, которые

затем помещаются в стек. В результате исходный список становится пустым и по первому правилу выходной список также становится пустым.

После того, как произошло успешное завершение первого правила, Пролог пытается выполнить второй предикат *insert,* вызов которого содержится в теле второго правила. Переменной *H1* сначала присваивается первое взятое из стека значение *9,* а предикат принимает вид *insert (9, [], [9]).*

Так как теперь удовлетворено все второе правило, то происходит возврат на один шаг рекурсии в предикате *insert_sort.* Из стека извлекается 3 и по третьему правилу вызывается предикат *asc_order,* то есть происходит попытка удовлетворения пятого правила *asc_order (3, 9):- 3>9.* Так как данное правило заканчивается неуспешно, то неуспешно заканчивается и третье правило, следовательно, удовлетворяетсячетвертое правило и *3* вставляется в выходной список слева от *9*: *insert (3, [9], [3, 9]).*

Далее происходит возврат к предикату *insert_sort*, который принимает следующий вид: *insert_sort ([3, 9], [3, 9]).*

На следующем шаге рекурсии из стека извлекается 7 и по третьему правилу вызывается предикат *asc_order* в виде *asc_order (7, 3):- 7>3.* Так как данное правило заканчивается успешно, то элемент 3 убирается в стек и *insert* вызвается рекурсивно еще раз, но уже с хвостом списка – *[9]: insert (7, [9], _).* Так как правило *asc_order (7, 9):- 7>9* заканчивается неуспешно, то выполняется четвертое правило, происходит возврат на предыдущие шаги рекурсии сначала *insert,* затем *insert_sort.*

В результате *7* помещается в выходной список между элементами *3* и *9*: *insert (7, [3, 9], [3, 7, 9]).*

При возврате еще на один шаг рекурсии из стека извлекается 4 и по третьему правилу вызывается предикат *asc_order* в виде *asc_order (4, 3):- 4>3.* Так как данное правило заканчивается успешно, то элемент 3 убирается в стек и *insert* вызвается рекурсивно еще раз, но уже с хвостом списка – *[7, 9]: insert (4, [7, 9], _).* Так как правило *asc_order (4, 7):- 4>7* заканчивается неуспешно, то выполняется четвертое правило, происходит возврат на предыдущие шаги рекурсии сначала *insert,* затем *insert_sort.*

В результате *4* помещается в выходной список между элементами *3* и *7*:

insert (4, [3, 7, 9], [3, 4, 7, 9]).
insert_sort [4, 7, 3, 9], [3, 4, 7, 9]).

3.2.16 Компоновка данных в список

Иногда при программировании определенных задач возникает необходимость собрать данные из фактов БД в список для последующей их обработки. Пролог содержит встроенный предикат *findall*, который позволяет выполнить данную операцию. Описание предиката *findall* выглядит следующим образом:

Findall (Var_, Predicate_, List_), где Var_ обозначает имя для терма предиката Predicate_, в соответствии с типом которого формируются элементы списка List_.

Пример 41: использование предиката findall.

```
domains
d=integer
predicates
decimal (d)
write_decimal
clauses
decimal (0)
decimal (1)
decimal (2)
decimal (3)
decimal (4)
decimal (5)
decimal (6)
decimal (7)
decimal (8)
decimal (9)
write_decimal:- findall(C, decimal (C), L), write (L).
goal
write_decimal.
```

3.2.17 Повторение и рекурсия в Прологе

Очень часто в программах необходимо выполнить одну и ту же операцию несколько раз. В программах на Прологе повторяющиеся операции обычно выполняются при помощи правил, которые используют возврат и рекурсию. Существует четыре способа построения итеративных и рекурсивных правил:

- механизм возврата;
- метод возврата после неудачи;
- правило повтора, использующее бесконечный цикл;
- обобщенное рекурсивное правило.

Правила повторений и рекурсии должны содержать средства управления их выполнением. Встроенные предикаты Пролога *fail* и *cut (!)* используются для управления возвратами, а условия завершения используются для управления рекурсией. Правила выполняющие повторения, используют возврат, а правила, выполняющие рекурсию, используют самовызов.

3.2.18 Механизм возврата

Возврат является автоматически инициируемым системой процессом, если не используются специальные средства управления им. Если в предикате цели есть хотя бы одна свободная переменная, то механизм возврата переберет все возможные способы связывания этой переменной, то есть реализует итеративный цикл.

Пример 42: распечатать все десятичные цифры.

```
domains
d=integer
predicates
decimal (d)
write_decimal (d)
clauses
decimal (0).
decimal (1).
decimal (2).
decimal (3).
decimal (4).
decimal (5).
decimal (6).
decimal (7).
decimal (8).
decimal (9).
write_decimal (C):- decimal (C), write (C), nll.
goal
write_decimal (C).
```

3.2.19 Метод возврата после неудачи

Метод возврата после неудачи может быть использован для управления вычислением внутренней цели при поиске всех возможных решений. Данный метод либо использует внутренний предикат Пролога *fail*, либо условие, которое порождает ложное значение.

Пример 43: распечатать все десятичные цифры.

```
domains
d=integer
predicates
decimal (d)
write_decimal
clauses
decimal (0)
decimal (1).
decimal (2).
decimal (3).
decimal (4).
decimal (5).
decimal (6).
decimal (7).
decimal (8).
decimal (9).
write_decimal:- decimal (C), write (C), nl, fail.
goal
```

write_decimal

В программе есть 10 предикатов, каждый из которых является альтернативным предложением для предиката decimal (d). Во время попытки вычислить цель внутренние подпрогрммы унификации связывают переменную С с термом первого предложения, то есть с цифрой 0. Так как существует следующее предложение, которое может обеспечить вычисление подцели decimal (C), то помещается указатель возврата, значение 0 выводится на экран. Предикат fail вызывает неуспешное завершение правила, внутренние подпрограммы унификации выполняют возврат и процесс повторяется до тех пор, пока не будет обработано последнее предложение.

Пример 44: подсчитать значения квадратов всех десятичных цифр.

```
domains
d =integer
predicates
decimal (d)
s (d, d)
cikl
clauses
decimal (0).
decimal (1).
decimal (2).
decimal (3).
decimal (4).
decimal (5).
decimal (6).
decimal (7).
decimal (8).
decimal (9).
s( X, Z):- Z=X*X.
cikl:-decimal (I), s(I , S), write (S), nl, fail.
goal
not(cikl)
```

Пример 45: необходимо выдать десятичные цифры до 5 включительно.

```
domains
d=integer
predicates
decimal (d)
write_decimal
make_cut (d)
clauses
decimal (0).
decimal (1).
decimal (2).
```

decimal (3).
decimal (4).
decimal (5).
decimal (6).
decimal (7).
decimal (8).
decimal (9).
write_decimal:- decimal (C), write (C), nl, make_cut (C),!.
make_cut (C):-C=5.
goal
write_decimal

Предикат ! используется для того, чтобы выполнить отсечение в указанном месте. Неуспешное выполнение предиката make_cut порождает предикат fail, который используется для возврата и доступа к цифрам до тех пор, пока цифра не будет равна 5.

Пример 46: необходимо выдать из БД первую цифру , равную 5.

domains
d=integer
predicates
decimal (d)
write_decimal
clauses
decimal (0).
decimal (5).
decimal (2).
decimal (3).
decimal (4).
decimal (5).
decimal (6).
decimal (5).
decimal (8).
decimal (9).
write_decimal:- decimal (C), C=5, write (C), nl, !.
goal
write_decimal

Если из тела правила убрать предикат !, то будут найдены все три цифры 5, что является результатом применения метода возврата после неудачи. При внесении отсечения будет выдана единственная цифра 5.

3 2 19 Метод повтора, использующий бесконечный цикл

Вид правила повторения, порождающего бесконечный цикл:

repeat.
repeat:- repeat.

Первый *repeat* является предложением, объявляющим предикат *repeat* истинным. Однако, поскольку имеется еще один вариант для данного

предложения, то указатель возврата устанавливается на первый *repeat*. Второй *repeat* – это правило, которое использует само себя как компоненту (третий *repeat*). Второй *repeat* вызывает третий *repeat*, и этот вызов вычисляется успешно, так как первый *repeat* удовлетворяет подцели *repeat*. Предикат *repeat* будет вычисляться успешно при каждой новой попытке его вызвать после возврата. Факт будет использоваться для выполнения всех подцелей. Таким образом, *repeat* это рекурсивное правило, которое никогда не бывает неуспешным. Предикат *repeat* широко используется в качестве компонента других правил, который вызывает повторное выполнение всех следующих за ним компонентов.

Пример 47: ввести с клавиатуры целые числа и вывести их на экран. Программа завершается при вводе числа 0.

```
domains
d=integer
predicates
repeat
write_decimal
do_echo
check (d)
clauses
repeat.
repeat:- repeat.
write_decimal:-nl, write( «Введите, пожалуйста, цифры»), nl,
                   write(«Я повторю их»), nl,
                   write(«Чтобы остановить меня, введите 0»), nl, nl.
do_echo:- repeat, readln (D), write(D), nl, check (D), !.
check (0):- nl, write («-OK!»).
check(_):- fail.
goal
write_decimal, do_echo.
```

Правило *do_echo* – это конечное правило повтора, благодаря тому, что предикат *repeat* является его компонентом и вызывает повторение предикатов *readln, write, и check.* Предикат *check* описывается двумя предложениями. Первое предложение будет успешным, если вводимая цифра 0, при этом курсор сдвигается на начало следующей строки и на экране появляется сообщение «OK!» и процесс повторения завершается, так как после предиката *check* в теле правила следует предикат отсечения. Если введенное значение отличается от 0, то результатом выполнения предиката *check* будет *fail* в соответствии со вторым предложением. В этом случае произойдет возврат к предикату *repeat. Repeat* повторяет посылки в правой части правила, находящиеся правее *repeat* и левее условия выхода из бесконечного цикла.

3.2.20 Методы организации рекурсии

Рекурсивная процедура – это процедура, которая вызывает сама себя. В рекурсивной процедуре нет проблемы запоминания результатов ее выполнения потому, что любые вычисленные значения можно передавать из одного вызова в другой, как аргументы рекурсивного предиката.

Например, в приведенной ниже программе вычисления факториала (пример 48), приведен пример написания рекурсивного предиката. При этом Пролог создает новую копию предиката *factorial* таким образом, что он становится способным вызвать сам себя как самостоятельную процедуру.

При этом не происходит копирования кода выполнения, но все аргументы и промежуточные переменные копируются в стек, который создается каждый раз при вызове рекурсивного правила.

Когда выполнение правила завершается, занятая стеком память освобождается и выполнение продолжается в стеке правила-родителя.

Пример 48: написать программу вычисления факториала.

```
predicates
factorial (byte, word)
clauses
factorial (0, 1).
factorial (1, 1):-!.
factorial (N, R):- N1=N-1, factorial (N1, R1), R=N*R1.
goal
f (7, Result).
```

Для вычисления факториала используется последовательное вычисление произведения ряда чисел. Его значение образуется после извлечения значений соответствующих переменных из стека, используемых как список параметров для последнего предиката в теле правила. Этот последний предикат вызывается после завершения рекурсии.

Пример 49: написать программу, генерирующую числа Фибоначчи до заданного значения.

```
predicates
f (byte, word)
clauses
f (1, 1).
f (2, 1).
f(N, F):- N1=N-1, f (N1, F1), N2=N1-1, f (N2,F2), F=F1+F2.
goal
f (10, Fib).
```

У рекурсии есть три основных преимущества:

- логическая простота по сравнению с итерацией;
- широкое применение при обработке списков;
- возможность моделирования алгоритмов, которые нельзя эффективно выразить никаким другим способом (например, описания задач, содержащих в себе подзадачи такого же типа).

У рекурсии есть один большой недостаток – использование большого объема памяти. Всякий раз, когда одна процедура вызывает другую, информация о выполнении вызывающей процедуры должна быть сохранена для того, чтобы вызывающая процедура могла, после завершения вызванной процедуры, возобновить выполнение на том месте, где остановилась.

Рассмотрим специальный случай, когда процедура может вызвать себя без сохранения информации о своем состоянии.

Предположим, что процедура вызывается последний раз, то есть после вызванной копии, вызывающая процедура не возобновит свою работу, то есть при этом стек вызывающей процедуры должен быть заменен стеком вызванной копии. При этом аргументам процедуры просто присваиваются новые значения и выполнение возвращается на начало вызывающей процедуры. С процедурной точки зрения этот процесс напоминает обновление управляющих переменных в цикле.

Эта операция в Visual Prolog называется оптимизацией хвостовой рекурсии или оптимизацией последнего вызова.

Создание хвостовой рекурсии в программе на Прологе означает, что:

- вызов рекурсивной процедуры является самой последней посылкой в правой части правила;
- до вызова рекурсивной процедуры в правой части правила не было точек отката.

Приведем примеры хвостовой рекурсии.

Пример 50:рекурсивный счетчик с оптимизированной хвостовой рекурсией.

```
count(100).
count(N):-write(N),nl,N1=N+1,count(N1).
goal
nl, count(0).
```

Модифицируем этот пример так, чтобы хвостовой рекурсии не стало.

Пример 51:рекурсивный счетчик без хвостовой рекурсии.

```
count1(100).
count1(N):-write(N),N1=N+1,count1(N1),nl.
goal
nl, count1(0).
```

Из-за вызова предиката *nl* после вызова рекурсивного предиката должен сохраняться стек.

Пример 52:рекурсивный счетчик без хвостовой рекурсии.

```
count2(100).
count2(N):-write(N),nl,N1=N+1,count2(N1).
count2(N):-N<0, write("N – отрицательное число").
goal
nl, count2(0).
```

Здесь есть непроверенная альтернатива до вызова рекурсивного предиката (третье правило), которое должно проверяться, если второе правило завершится неудачно. Таким образом стек должен быть сохранен.

Пример 53:рекурсивный счетчик без хвостовой рекурсии.

```
count3(100).
count3(N):-write(N),nl,N1=N+1,check(N1),count3(N1).
check(Z):-Z>=0.
check(Z):-Z<0.
goal
nl, count3(0).
```

Здесь тоже есть непроверенная альтернатива до вызова рекурсивного предиката (предикат *check*). Случаи в примерах 52 и 53 хуже, чем в примере 51, так как они генерируют точки возврата.

Очень просто сделать рекурсивный вызов последним в правой части правила, но как избежать альтернатив? Для этого следует использовать предикат отсечения, который предотвращает возвраты в точки, левее предиката отсечения. Модифицируем 52 и 53 примеры так, чтобы была хвостовая рекурсия.

Пример 54:рекурсивный счетчик из примера 52 с хвостовой рекурсией.

```
count4(100).
count4(N):-N>0,!,write(N),N1=N+1,count4(N1).
count4(N):- write("N – отрицательное число").
goal
nl, count4(0).
```

Пример 55:рекурсивный счетчик из примера 53 с хвостовой рекурсией.

```
count5(100).
count5(N):-write(N),N1=N+1 check(N1),!, count5(N1).
check(Z):-Z>=0.
check(Z):-Z<0.
goal
nl, count5(0).
```

3.2.21 Создание динамических баз данных

В Прологе существуют специальные средства для организации внутренних и внешних баз данных. Эти средства рассчитаны на работу с реляционными базами данных. Внутренние подпрограммы унификации осуществляют автоматическую выборку фактов из внутренней (динамической) базы данных с нужными значениями известных параметров и присваивают значения неопределенным параметрам.

Раздел программы *facts* в Visual Prolog предназначен для описания предикатов *динамической (внутренней) базы данных*. База данных называется динамической, так как во время работы программы из нее можно удалять любые факты, а также добавлять новые факты. В этом состоит ее отличие от *статических баз данных*, где факты являются частью кода программы и не могут быть изменены во время исполнения.

Иногда бывает полезно иметь часть информации базы данных в виде фактов статической БД - эти данные заносятся в динамическую БД сразу после активизации программы. В общем случае, предикаты статической БД

имеют другое имя, но ту же самую форму представления данных, что и предикаты динамической БД. Добавление латинской буквы d к имени предиката статической БД - обычный способ различать предикаты динамической и статической БД.

Следует отметить два ограничения, объявленные в разделе *facts* :

- в динамической базе данных Пролога могут содержаться только факты;
- факты базы данных не могут содержать свободные переменные.

Допускается наличие нескольких разделов *facts* , тогда в описании каждого раздела *facts* нужно явно указать его имя, например *facts – mydatabase.* В двух различных разделах *facts* нельзя использовать одинаковые имена предикатов. Также нельзя использовать одинаковые имена предикатов в разделах *facts* и *predicates*. Если имя базы данных не указывается, то ей присваивается стандартное имя *dbasedom*. Программа может содержать локальные безымянные разделы фактов, если она состоит из единственного модуля, который не объявлен как часть проекта. Среда разработки компилирует программный файл как единственный модуль только при использовании утилиты *TestGoal*. Иначе безымянный раздел фактов должен быть объявлен глобальным, то есть как *global facts.*

В Прологе есть специальные встроенные предикаты для работы с динамической базой данных:

* *assert;*
* *asserta;*
* *assertz;*
* *retract;*
* *retractall;*
* *save;*
* *consult.*
* Предикаты *assert, asserta, assertz,* - позволяют занести факт в БД, а предикаты *retract, retractall* - удалить из нее уже имеющийся факт.

Предикат *assert* заносит новый факт в БД в произвольное место, предикат *asserta* добавляет новый факт перед всеми уже внесенными фактами данного предиката, *assertz* добавляет новый факт после всех фактов данного предиката.

Предикат *retract* удаляет из БД первый факт, который сопоставляется с заданным фактом, предикат *retractall* удаляет из БД все факты, которые сопоставляются с заданным фактом.

Предикат *save* записывает все факты динамической БД в текстовый файл на диск, причем в каждую строку файла заносится один факт. Если файл с заданным именем уже существует, то старый файл будет затерт.

Предикат *consult* записывает в динамическую БД факты, считанные из текстового файла, при этом факты из файла дописываются в имеющуюся БД. Факты, содержащиеся в текстовом файле должны быть описаны в разделе *domains*.

Пример 55:Написать программу, генерирующую множество 4-разрядных двоичных чисел и записывающих их в динамическую БД.

```
facts
dbin (byte, byte, byte, byte)
predicates
cifra (byte)
bin (byte, byte, byte, byte)
clauses
cifra (0).
cifra (1).
bin (A, B, C, D):- cifra (A), cifra (B), cifra (C), cifra (D),
                   assert (bin (A, B, C, D)).
goal
bin (A, B, C, D).
```

Пример 56:Написать программу, подсчитывающую число обращений к программе.

```
facts
dcount (word)
predicates
modcount
clauses
dcount (0).
modcount:- dcount (N), M=N+1, retract (dcount (N)),asserta (dcount (M)).
goal
modcount.
```

Пример 57:Написать программу, определяющую родственные отношения.

```
facts
dsisters(symbol,symbol)
dbrothers(symbol,symbol)
predicates
parents(symbol,symbol)
pol(symbol,symbol)
sisters(symbol,symbol)
brothers(symbol,symbol)
clauses
parents (anna, olga).
parents (petr, olga).
parents (anna, irina).
parents (petr, irina).
parents (anna, ivan).
parents (petr, ivan).
pol(olga, w).
pol(anna ,w).
pol(petr, m).
```

pol(irina, w).
pol(ivan, m).
sisters (X, Y):-dsisters(X, Y).
sisters (X, Y):- parents (Z, X), parents (Z,Y),pol(X,w),
pol(Y,w),not(X=Y),not(dsisters(X,Y)),
asserta(dsisters(X, Y)).
brothers (X ,Y):-dbrothers(X, Y).
brothers (X, Y):- parents (Z,X), parents l(Z,Y),pol(X,m),
pol(Y,m),not(X=Y),not(dbrothers(X,Y)),
asserta(dbrothers(X,Y)).
goal
sisters (X, Y), save ("mybase.txt").

3 2 22 Использование строк в Прологе.

Строка – это набор символов. При программировании на Прологе символы могут быть «записаны» при помощи алфавитно-цифрового представления или при помощи их ASCII-кодов. Обратный слэш (\), за которым непосредственно следует ASCII-код (N) символа, интерпретируется как символ. Для представления одиночного символа выражение \N должно быть заключено в апострофы ('\N'). Для представления строки символов ASCII-коды помещаются друг за другом и вся строка заключается в кавычки («\N\N\N»).

Операции, обычно выполняемые над строками, включают:

- объединение строк для образования новой строки;
- разделение строки для создания двух новых строк, каждая из которых содержит некоторые из исходных символов;
- поиск символа или подстроки внутри данной строки.

Для удобства работы со строками Пролог имеет несколько встроенных предикатов, манипулирующих со строками:

- str_len – предикат для нахождения длины строки;
- concat – предикат для объединения двух строк;
- frontstr – предикат для разделения строки на две подстроки;
- frontchar – предикат для разделения строки на первый символ и остаток;
- fronttoken – предикат для разделения строки на лексему и остаток.

Синтаксис предиката str_len:

str_len (Str_value, Srt_length), где первый терм имеет тип string, а второй терм имеет тип integer.

Пример 58:

str_len («Today», L)- в данном случае переменная L получит значение 5;

str_len («Today», 5) – в данном случае будет выполнено сравнение длины строки «Today» и 5. Так как они совпали, то предикат выполнится успешно, если бы длина строки не была равна 5, то предикат вылонился бы неуспешно.

Синтаксис предиката concat:

concat (Str1, Str2, Str3), где все термы имеют тип string.

Пример 59:

concat («Today», «Tomorrow», S3)- в данном случае перменная S3 получит значение «TodayTomorrow»;

concat (S1, «Tomorrow», «TodayTomorrow») – в данном случае S1 будет присвоено значение «Today»;

concat («Today», S2, «TodayTomorrow») – в данном случае S2 будет присвоено значение «Tomorrow»;

concat («Today», «Tomorrow», «TodayTomorrow»)- будет проверена возможность склейки строк «Today» и «Tomorrow» в строку «TodayTomorrow».

Синтаксис предиката frontstr:

frontstr (Number, Str1, Str2, Str3), где терм Number имеет тип integer, а остальные термы имеют тип string. Терм Number задает число символов, которые должны быть скопированы из строки Str1 в строку Str2, остальные символы будут скопированы в строку Str3.

Пример 60:

frontstr (6,«Expert systems», S2, S3)- в данном случае перменная S2 получит значение «Expert», а S3 получит значение ,« systems».

Синтаксис предиката frontchar:

frontchar (Str1, Char_, Str2), где терм Char_ имеет тип char, а остальные термы имеют тип string.

Пример 61:

frontchar («Today », C, S2)- в данном случае перменная C получит значение «T», а S2 получит значение ,«oday»;

frontchar («Today », 'T', S2) – в данном случае S2 будет присвоено значение «oday»;

frontchar («Today», C, «oday») – в данном случае C будет присвоено значение «T»;

frontchar (S1, «T», «oday») – в данном случае S1 будет присвоено значение «Today»;

frontchar («Today», «T», «oday»)- будет проверена возможность склейки строк «T» и «oday» в строку «Today».

Синтаксис предиката fronttoken:

fronttoken (Str1, Lex, Str2), где все термы имеют тип string. В терм Lex копируется первая лексема строки Str1, остальные символы будут скопированы в строку Str2. Лексема – это имя в соответствии с синтаксисом языка Турбо-Пролог или строчное представление числа или отдельный символ (кроме пробела).

Пример 62:

fronttoken («Expert systems», Lex, S2)- в данном случае перменная Lex получит значение «Expert», а S2 получит значение ,« systems».

fronttoken («$Expert», Lex, S2)- в данном случае перменная Lex получит значение «$», а S2 получит значение ,«Expert».

fronttoken («Expert systems», Lex, « systems»)- в данном случае перменная Lex получит значение «Expert»;

fronttoken («Expert systems», «Expert», S2)- в данном случае перменная S2 получит значение « systems»;

fronttoken (S1, «Expert», « systems»)- в данном случае перменная S1 получит значение «Expert systems»;

fronttoken («Expert systems», «Expert», « systems»)- в данном случае будет проверена возможность склейки лексемы и остатка в строку «Expert systems».

3.2.23 Преобразование данных в Прологе

Для преобразования данных из одного типа в другой Пролог имеет следующие встроенные предикаты:

upper_lower;
str_char;
str_int;
str_real;
char_int.

Все предикаты преобразования данных имеют два терма. Все предикаты имеют два направления преобразования данных в зависимости от того, какой терм является свободной или связанной переменной.

Пример 63:

```
upper_lower («STARS», S2).
upper_lower (S1,«stars»).
str_char («T», C).
str_char (S, ’T’).
str_int («123», N).
str_int (S, 123).
str_real («12.3», R).
str_real (S, 12.3).
char_int (‘A’, N).
char_int (C, 61).
```

В Прологе нет встроенных предикатов для преобразования действительных чисел в целые и наоборот, или строк в символы. На самом деле, правила преобразования данных типов очень просты и могут быть заданы в программе самими программистом.

Пример 64:

```
predicates
conv_real_int (real, integer)
conv_int_real (integer, real)
conv_str_symb (string, symbol)
clauses
conv_real_int (R, N):- R=N.
conv_int_real (N, R):- N=R.
conv_str_symb (S, Sb):- S=Sb.
goal
conv_real_int (5432.765, N). (N= 5432).
```

conv_int_real (1234, R). (R=1234).

conv_str_symb («Visual Prolog», Sb). (Sb=Visual Prolog).

Пример 65: преобразование строки в список символов с использованием предиката frontchar.

```
domains
list=char*
predicates
convert (string, list)
clauses
convert («», []).
convert (Str, [H\T]):- frontchar(Str, H, Str1),
                       convert(Str1, T).
```

3.2.24 Представление бинарных деревьев

Одной из областей применения списков является представление множества объектов. Недостатком представления множества в виде списка является неэффективная процедура проверки принадлежности элемента множеству. Используемый для этой цели предикат *member* неэффективен при использовании больших списков.

Для представления множеств могут использоваться различные древовидные структуры, которые обеспечивают более эффективную реализацию проверки принадлежности элемента множеству, в частности, в данном разделе для этой цели рассматриваются бинарные деревья.

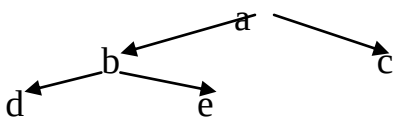
Представление бинарных деревьев основано на определении рекурсивной структуры данных, использующей функцию типа *tree (Top, Left, Right) или tree (Left, Top, Right)*, где *Top* - вершина дерева, *Left и Right* - соответственно левое и правое поддерево. Пустое дерево обозначим термом *nil*. Объявить бинарное дерево можно следующим образом:

Пример 66:

```
domains
treetype1=tree(symbol, treetype1, treetype1);nil
treetype2=tree(treetype2, symbol, treetype2);nil
```

Пример 67:

Пусть дано дерево следующего вида:



Такое дерево может быть задано следующим образом:

1. левое поддерево: tree (b, tree (d, nil, nil), tree (e, nil, nil)).

2. правое поддерево: tree (c, nil, nil).

3. все дерево: tree (a, tree (b, tree (d, nil, nil), tree (e, nil, nil)), tree (c, nil, nil)).

Пример 69: написать программу проверки принадлежности вершины бинарному дереву.

```
domains
treetype = tree(symbol, treetype, treetype);nil()
predicates
in(symbol, treetype)
clauses
in(X, tree(X,_,_).
in(X, tree(_,L,_):-in(X, L).
in(X, tree(_,_,R):-in(X, R).
goal
in(d,tree(a, tree(b, tree(d, nil, nil),
                     tree(e, nil, nil)),
                     tree(c, nil, nil))).
```

Поиск вершины в неупорядоченном дереве также неэффективен, как и поиск элемента в списке. Если ввести отношение упорядочения между элементами множества, то процедура поиска элемента становится гораздо эффективнее. Можно ввести отношение *упорядочения слева направо непустого дерева tree(X, Left, Right)* следующим образом:

1. Все узлы в левом поддереве *Left* меньше *X*.
2. Все узлы в правом поддереве *Right* больше *X*.
3. Оба поддерева также являются упорядоченными.

Преимуществом упорядочивания является то, что для поиска любого узла в дереве достаточно провести поиск не более, чем в одном поддереве. В результате сравнения узла и корня дерева из рассмотрения исключается одно из поддеревьев.

Пример 70: написать программу проверки принадлежности вершины упорядоченному слева направо бинарному дереву.

```
domains
treetype = tree(byte, treetype, treetype);nil()
predicates
in(byte, treetype)
clauses
in(X, tree(X,_,_).
in(X, tree(Root,L,R):-Root>X,in(X, L).
in(X, tree(Root,L,R):-Root<X,in(X, R).
goal
in(6,tree(4, tree(2, nil, tree(3, nil, nil)),
      tree(5,tree(1,nil,nil),nil)),tree(8,tree(7,nil,nil),tree(9,nil,tree(10,nil,nil)
))).
```

Пример 71: написать программу печати вершин бинарного дерева, начиная от корневой и следуя правилу левого обхода дерева.

```
domains
treetype = tree(symbol, treetype, treetype);nil()
predicates
```

```
print_all_elements(treetype)
clauses
print_all_elements(nil).
print_all_elements(tree(X, Y, Z)) :-
        write(X), nl, print_all_elements(Y),
        print_all_elements(Z).
goal
print_all_elements(tree(a, tree(b, tree(d, nil, nil),
                        tree(e, nil, nil)),
                        tree(c, nil, nil))).
```

Пример 72: написать программу проверки изоморфности двух бинарных деревьев.

```
domains
treetype = tree(symbol, treetype, treetype);nil
predicates
isotree (treetype, treetype)
clauses
isotree (T, T).
isotree (tree (X, L1, R1), tree (X, L2, R2)):- isotree (L1, L2), isotree (R1,
R2).
isotree (tree (X, L1, R1), tree (X, L2, R2)):- isotree (L1, R2), isotree (L2,
R1).
```

Пример 73: написать предикаты создания бинарного дерева из одного узла и вставки одного дерева в качестве левого или правого поддерева в другое дерево.

```
domains
treetype = tree(symbol, treetype, treetype);nil
predicates
create_tree(symbol, tree)
insert_left(tree, tree, tree)
insert_rigth(tree, tree, tree)

clauses
create_tree(N, tree(N,nil,nil)).
insert_left(X, tree(A,_,B), tree(A,X,B)).
insert_rigth(X,tree(A,B,_), tree(A,B,X)).
goal
create_tree(a, T1)), insert_left(tree(b, nil, nil), tree(c, nil, nil), T2),
insert_rigth(tree(d, nil, nil), tree(e, nil, nil), T3).
```

В результате: T1=tree(a, nil, nil), T2=tree(c, tree(b, nil, nil), nil), T3=tree(d, nil, tree(e, nil, nil)).

3.2.25 Представление графов в языке Пролог

Для представления графов в языке Пролог можно использовать три способа:

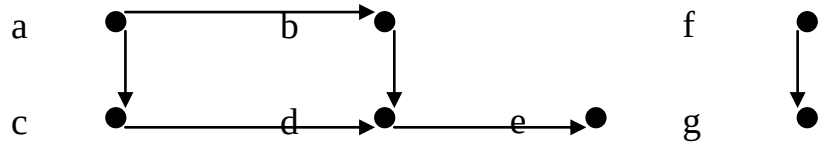
1. Использование факта языка Пролог для описания дуг или рёбер графа.
2. Использование списка или структуры данных для объединения двух списков: списка вершин и списка рёбер.
3. Использование списка списков: каждый подсписок в качестве головы содержит вершину графа, а в качестве хвоста - список смежных вершин.

Две самые распространённые операции, которые выполняются над графами:

- Поиск пути между двумя вершинами;
- Поиск в графе подграфа, который обладает некоторыми заданными свойствами.

Пример 74:

Определить наличие связи между вершинами графа, представленного на рисунке:



Две вершины графа связаны, если существует последовательность ребер, ведущая из первой вершины во вторую. Используем для описания структуры графа факты языка Пролог.

```
domains
top=symbol
predicates
edge (top, top)
/* аргументы обозначают имена вершин */
path( top,top)
/*Предикат connected(symbol, symbol) определяет отношение связанности вершин.*/
clauses
edge (a, b).
edge (c, d).
edge (a, c).
edge (d, e).
edge (b, d).
edge (f, g).
path (X, X).
path (X, Y):- edge (X, Z), path (Z, Y).
```

Пример 75:

Решить задачу из примера 74, используя списочные структуры для представления графа. Граф задается двумя списками: списком вершин и списком ребер. Ребро представлено при помощи структуры edge.

```
domains
edge= e(symbol, symbol)
list=edge*
predicates
path(list, symbol, symbol)
member(list,edge)
/*предикат path(graf, symbol, symbol) определяет отношение связности вершин.*/
clauses
member([H|_],H).
member([_|T],X):-member(T,X).
%path ([],_,_):-fail.
path(L,X,Y):-member(L,e(X,Y)),!.
path(L,X,Y):-member(L,e(X,Z)),path(L,Z,Y).
goal
path ([e(a, b),e(b, c),e(a, f),e(c, d),e(f,d)], a, d).
%path ([e(a, b),e(c, d),e(a, c),e(d, e),e(b, d),e(f, g)], b, g).
```

Пример 76:

Решить задачу из примера 74, используя списочные структуры для представления графа. Граф задается списком списков: в каждом подсписке голова является вершиной, а хвост – списком смежных вершин.

```
domains
edge= e(symbol, symbol)
/* аргументы обозначают имена вершин */
list1=symbol*
list2=edge*
graf = g(list1, list2)
predicates
path(graf, symbol, symbol)
/*Предикат path(graf, symbol, symbol) определяет отношение связанности вершин в графе.*/
clauses
path (g([],[]),_,_).
path (g([X|_],[e(X,Y)|_]),X,Y).
%path (g([X|T],[e(X,_)|T1]),X,Y):-
%path (g([X|T],T1),X,Y).
path (g([X|T],[e(X,Z)|T1]),X,Y):-
path (g(T,T1),Z,Y).
path (g([Z|T],T1),X,Y):-Z<>X,
path (g(T,T1),X,Y).
path (g([X|T],[e(_,_)|T1]),X,Y):-
path (g([X|T],T1),X,Y).
goal
path (g([a, b, c, d],[e(a, b),e(b, c),e(a, d),e(c, e)]), b, e).
```

3.2.26 Поиск пути на графе.

Программы поиска пути на графе относятся к классу так называемых недетерминированных программ с неизвестным выбором альтернативы, то есть в таких программах неизвестно, какое из нескольких рекурсивных правил будет выполнено для доказательства до того момента, когда вычисление будет успешно завершено. По сути дела такие программы представляют собой спецификацию алгоритма поиска в глубину для решения определенной задачи. Программа проверки изоморфности двух бинарных деревьев, приведенная в примере 56 относится к задачам данного класса.

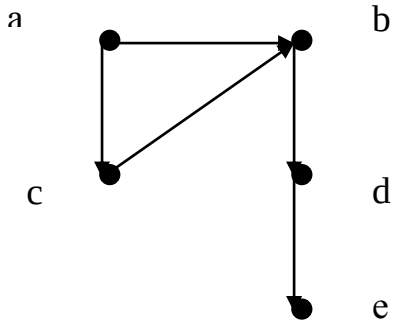
Пример 74:

Определить путь между вершинами графа, представленного на рисунке:

A- переменная обозначающая начало пути

B- вершина в которую нужно попасть

P -ациклический путь на графе (ациклический путь- это путь не имеющий повторяющих вершин).



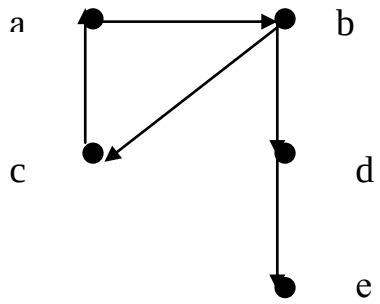

```
domains
top=symbol
listtop=top*
predicates
edge (top, top)
/* аргументы обозначают имена вершин */
path (top, top, listtop)
/*Предикат path( top, top, listtop) создает список из вершин, составляющих путь.*/
clauses
edge (a, b).
edge (b, c).
edge (c, a).
edge (b, d).
edge (d, e).
path (A, A, [A]).
path (A, B, [A\P]):-edge(A, N), path(N, B, P).
```

С помощью поиска в глубину осуществляется корректный обход

любого конечного дерева или ориентированного ациклического графа. Однако, встречаются задачи, где требуется производить обход графа с циклами. В процессе вычислений может образоваться бесконечный программный цикл по одному из циклов графа.

Неприятностей с зацикливанием можно избежать двумя способами: ограничением на глубину поиска или накоплением пройденных вершин. Последний способ можно реализовать при помощи модификации отношения *path*. К аргументам отношения добавляется дополнительный аргумент, используемый для накопления уже пройденных при обходе вершин, для исключения повторного использования одного и того же состояния применяется проверка.

Пример 75: написать программу обхода конечного ориентированного графа, представленного на рисунке.




```
domains
top=symbol
listtop=top*
predicates
edge(top,top)
path (top,top,listtop,listtop)
path (top,top)
member(top,listtop)
reverse(listtop,listtop,listtop)
clauses
edge(a,b).
edge(b,c).
edge(c,a).
edge(b,d).
edge(d,e).
member(A,[A|_]):-!.
member(A,[_|T]):-member(A,T).
reverse([],T2,T2).
reverse([H|T],T1,T2):-reverse(T,[H|T1],T2).
```

```
path(A,B,P,[B|P]):-edge(A,B).
path(A,B,P,P2):-edge(A,N),not(member(N,P)),
                P1=[N|P], path (N,B,P1,P2).
path(A,B):-path(A,B,[A],P),reverse(P,[],Res),write(Res).
goal
path(a,e).
```

3.2.27 Метод “образовать и проверить”

Метод “образовать и проверить” – общий прием, используемый при проектировании алгоритмов и программ. Суть его состоит в том, что один процесс или программа генерирует множество предполагаемых решений задачи, а другой процесс или программа проверяет эти предполагаемые решения, пытаясь найти те из них, которые действительно являются решением задачи.

Используя вычислительную модель Пролога, легко создавать логические программы, реализующие метод “образовать и проверить”. Обычно такие программы содержат конъюнкцию двух целей, одна из которых действует как генератор предполагаемых решений, а вторая проверяет, являются ли эти решения приемлемыми. В Прологе метод “образовать и проверить” рассматривается как метод недетерминированного программирования. В такой недетерминированной программе генератор вносит предположение о некотором элементе из области возможных решений, а затем просто проверяется, корректно ли данное предположение генератора.

Для написания программ недетерминированного выбора конкретного элемента из некоторого списка в качестве генератора обычно используют предикат *member* из примера 36, порождающий множество решений. При задании цели *member (X, [1,2,3,4])* будут даны в требуемой последовательности решения *X=1, X=2, X=3, X=4.*

Пример 76: проверить существование в двух списках одного и того же элемента.

```
domains
list=integer*
predicates
member (integer, list)
intersect(list,list)
clauses
member (Head, [Head |_ ]).
member (Head, [_ | Tail ]):- member (Head, Tail).
intersect (L1, L2):- member(X, L1), member(X, L2).
goal
intersect ([1, 4, 3, 2], [2, 5,6]).
```

Первая подцель *member* в предикате *intersect* генерирует элементы из первого списка, а с помощью второй подцели *member* проверяется, входят ли эти элементы во второй список. Описывая данную программу как

недетерминированную, можно говорить, что первая цель делает предположение о том, что *X* содержится в списке *L1*, а вторая цель проверяет, является ли *X* элементом списка *L2*.

Следующее определение предиката *member* с использованием предиката *append:*

member(X, L):- append(L1, [X|L2], L) само по существу является программой, в которой реализован принцип «образовать и проверить». Однако, в данном случае, два шага метода сливаются в один в процессе унификации. С помощью предиката *append* производится расщепление списка и тут же выполняется проверка, является ли *X* первым элементом второго списка.

Еще один пример преимущества соединения генерации и проверки дает программа для решения задачи об *N* ферзях: требуется разместить *N* ферзей на квадратной доске размером *NxN* так, чтобы на каждой горизонтальной, вертикальной или диагональной линии было не больше одной фигуры. В первоначальной формулировке речь шла о размещении *8* ферзей на шахматной доске таким образом, чтобы они не угрожали друг другу. Отсюда пошло название задачи о ферзях.

Эта задача хорошо изучена в математике. Для *N=2* и *N=3* решения не существует; два симметричных решения при *N=4* показаны на рисунке. Для *N=8* существует *88* (а с учетом симметричных – *92*) решений этой задачи.

		Q	
Q			
			Q
	Q		

	Q		
			Q
Q			
		Q	

Приведенная в *примере 77* программа представляет решение задачи об *N ферзях*. Решение задачи представляется в виде некоторой перестановки

списка от *1 до N*. Порядковый номер элемента этого списка определяет номер вертикали, а сам элемент – номер горизонтали, на пересечении которых стоит ферзь. Так решение *[2, 4, 1, 3]* задачи о четырех ферзях соответствует первому решению, представленному на рисунке, а решение *[3, 1, 4, 2]*- второму решению. Подобное описание решений и программа их генерации неявно предполагают, что в любом решении задачи о ферзях на каждой горизонтали и на каждой вертикали будет находиться по одному ферзю.

Пример 77: программа решения задачи об N ферзях.

```
domains
list=integer*
predicates
range (integer, integer, list)
/* предикат порождает список, содержащий числа в заданном интервале*/
queens (list, list, list)
/* предикат формирует решение задачи о N ферзях в виде списка решений, при этом первый список – текущий вариант списка размещения ферзей, второй список промежуточное решение, третий список - результат*/
select (integer, list, list)
/*предикат удаляет из списка одно вхождение элемента*/
attack (integer, list)
/*предикат преобразует attack, чтобы ввести начальное присваивание разности в номерах горизонталей */
attack (integer, integer, list)
/*предикат проверяет условие атаки ферзя другими ферзями из списка, два ферзя находятся на одной и той же диагонали, на расстоянии M вертикалей друг от друга, если номер горизонтали одного ферзя на M больше или на M меньше номера горизонтали другого ферзя*/
fqueens (integer, list)
clauses
range (M, N, [M|T]):- M<N, M1=M+1, range (M1, N, T).
range (N, N, [N]):-!.
select(X,[X|T1],T1).
select (X, [Y|T1], [Y|T2]):-select (X, T1, T2).
attack1 (X, L):- attack(X, 1, L).
attack( X, N, [Y|T2]):-N=X-Y; N=Y-X.
attack( X, N, [Y|T2]):-N1=N+1, attack (X, N1, T2).
queens (L1, L2, L3):-select (X, L1, L11),
                     not (attack1 (X,L2)),
                     queens (L11, [X|L2], L3).
queens ([], L, L).
fqueens(N,L):-range (1, N, L1),
              queens(L1,[],L).
goal
```

fqueens (4,L),write(L).

При таком задании цели, будет выдано второе решение, представленное на рисунке, если задать внешнюю цель, то будут выданы оба решения.

В данной программе реализован принцип «образоввать и проверить», так как сначала с помощью предиката range генерируется список, содержащий числа от 1 до N. Предикат select перебирает все элементы из полученного списка для размещения очередного ферзя, при этом корректность размещения проверяется при помощи предиката attack. Таким образом, генератором является предикат select, а проверка реализуется при помощи отрицания предиката attack. Чтобы проверить, в безопасном положении находится новый ферзь, необходимо знать позиции ранее размещенных ферзей. В данном случае для хранения промежуточных результатов используется второй параметр предиката queens, так как решение задачи находится на прямом ходе рекурсии, для закрепления результата при выходе из рекурсии используется третий параметр.

4 Основные стратегии решения задач. Поиск решения в пространстве состояний

4.1 Понятие пространства состояния

Пространство состояний – это граф, вершины которого соответствуют ситуациям, встречающимся в задаче («проблемные ситуации»), а решение задачи сводится к поиску путей на этом графе. На самом деле, задача поиска пути на графе и задача о N ферзях - это задачи, использующие одну из стратегий перебора альтернатив в пространстве состояний, а именно – стратегию поиска в глубину.

Рассмотрим другие задачи, для решения которых можно использовать в качестве общей схемы решения пространство состояний.

К таким задачам относятся следующие задачи:

- задача о восьми ферзях;
- переупорядочение кубиков, поставленных друг на друга в виде столбиков;
- головоломка «игра в восемь»;
- головоломка о «ханойской башне»;
- задача о перевозке через реку волка, козы и капусты;
- задача о двух кувшинах;
- задача о коммивояжере;
- другие оптимизационные задачи.

Со всеми задачами такого рода связано два типа понятий:

- проблемные ситуации;
- разрешенные ходы или действия, преобразующие одни проблемные ситуации в другие.

Проблемные ситуации вместе с возможными ходами образуют

направленный граф, называемый *пространством состояний.*

Пространство состояний некоторой задачи определяет «правила игры»: вершины пространства состояний соответствуют ситуациям, а дуги – разрешенным ходам или действиям, или шагам решения задачи. Конкретная задача определяется:

- пространством состояний;
- стартовой вершиной;
- целевым условием или целевой вершиной.

Каждому разрешенному ходу или действию можно приписать его стоимость. Например, в задаче о коммивояжере ходы соответствуют переездам из города в город, ясно, что стоимость хода в данном случае – это расстояние между соответствующими городами.

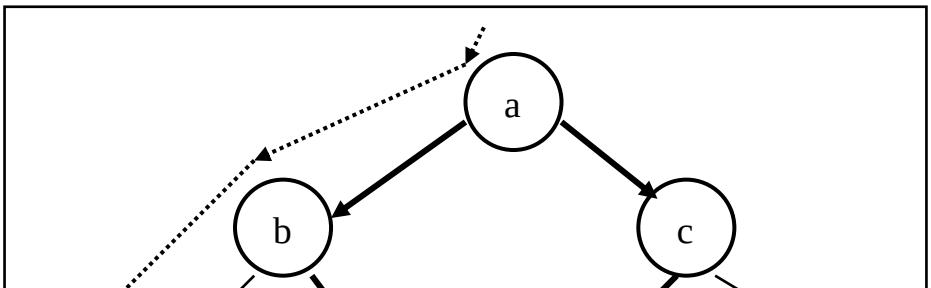
В тех случаях, когда ход имеет стоимость, программист заинтересован в отыскании решения минимальной стоимости. Стоимость решения – это сумма стоимостей дуг, из которых состоит «решающий путь» – путь из стартовой вершины в целевую. Даже если стоимости не заданы, все равно может возникнуть оптимизационная задача: требуется найти кратчайшее решение.

В представленной в *примере 77* программе о *N* ферзях проблемная ситуация (вершина в пространстве состояний) описывается в виде списка из *N X*-координат ферзей, а переход из одной вершины в другую генерирует предикат *queens*, причем начальная ситуация генерируется предикатом *range*, а целевая ситуация определяется при помощи предиката *attack.*

4.2 Основные стратегии поиска решений

4.2.1 Поиск в глубину

Программа решения задачи о N ферзях реализует стратегию поиска в глубину. Под термином «в глубину» имеется в виду тот порядок, в котором рассматриваются альтернативы в пространстве состояний. Всегда, когда алгоритму поиска в глубину надлежит выбрать из нескольких вершин ту, в которую следует перейти для продолжения поиска, он предпочитает самую «глубокую» из них. Самая глубокая вершина – это вершина, расположенная дальше других от стартовой вершины. На следующем рисунке показан пример, который иллюстрирует работу алгоритма поиска в глубину. Этот порядок в точности соответствует результату трассировки процесса вычислений при поиске решения.



Порядок обхода вершин указан пунктирными стрелками, a – начальная вершина, j и f – целевые вершины.

Поиск в глубину наиболее адекватен рекурсивному стилю программирования, принятому в Прологе. Причина этого состоит в том, что обрабатывая цели, Пролог сам просматривает альтернативы именно в глубину.

На Прологе переход от одной проблемной ситуации к другой может быть представлен при помощи предиката *after (X, Y, C)*, который истинен тогда, когда в пространстве состояний существует разрешенный ход из вершины *X* в вершину *Y*, стоимость которого равна *C*. Предикат *after* может быть задан в программе явным образом в виде фактов, однако такой принцип оказывается непрактичным, если пространство состояний является достаточно сложным. Поэтому отношение следования *after* обычно определяется неявно, при помощи правил вычисления вершин, следующих за некоторой заданной вершиной.

Другой проблемой при описании пространства состояний является способ представления самих вершин, то есть самих состояний.

В качестве первого примера решения таких задач рассмотрим задачу о ханойских башнях. Есть три стержня и набор дисков разного диаметра. В начале игры все диски надеты на левый стержень. Цель игры заключается в переносе всех дисков на правый стержень по одному стержню за раз, при этом нельзя ставить диск большего диаметра на диск меньшего диаметра. Для этой игры есть простая стратегия:

1. Один диск перемещается непосредственно;
2. *N* дисков переносятся в 3 этапа:

- Перенести *N-1* диск на средний стержень;
- Перенести последний диск на правый стержень;
- Перенести *N-1* диск со среднего на правый стержень.

В программе на языке Пролог есть 3 предиката:

- *hanoi* – запускающий предикат, указывает сколько дисков надо

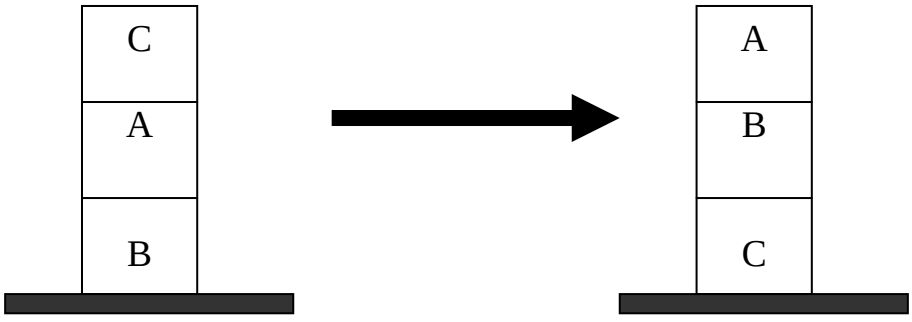
переместить;

- *move* – описывает правила перемещения дисков с одного стержня на другой;
- *inform* – указывает на действие с конкретным диском.

Пример 78: решение задачи о ханойских башнях.

```
domains
loc=right;middle;left
% описывает состояние стержней
predicates
hanoi(integert)
% определяет размерность задачи
move(integer,loc,loc,loc)
% определяет переход из одной вершины пространства состояния в другую, то есть описывает правила перекладывания дисков
inform(loc,loc)
% распечатывает действия с дисками
clauses
hanoi(N):-move(N,left,middle,right).
move(1,A,_,C):-inform(A,C),!.
move(N,A,B,C):-N1=N-1, move(N1,A,C,B),
inform(A,C), move(N1,B,A,C).
inform(Loc1,Loc2):-nl, write("Move a disk from ",Loc1," to ", Loc2).
goal
hanoi(3).
```

В качестве второго примера решения таких задач рассмотрим задачу нахождения плана переупорядочивания кубиков, представленную на следующем рисунке.



На каждом шаге разрешается переставлять только один кубик. Кубик можно взять только тогда, когда его верхняя поверхность свободна. Кубик можно поставить либо на стол, либо на другой кубик. Для того, чтобы получить требумое состояние, необходимо получить последовательность ходов, реализующую данную трансформацию. В качестве примера будет рассмотрен общий случай данной задачи, когда имеется произвольное число кубиков в столбиках. Число столбиков ограничено некоторым максимальным

значением.

Проблемную ситуацию можно представить как список столбиков. Каждый столбик, в свою очередь, представляется списком кубиков, из которых он составлен. Кубики упорядочены в списке таким образом, что самый верхний кубик находится в голове списка. «Пустые» столбики изображаются как пустые списки. Таким образом, исходную ситуацию на рисунке можно записать как терм *[[c, a, b], [], []]*.

Целевая ситуация- это любая конфигурация кубиков, содержащая столбик, составленный из имеющихся кубиков в указанном порядке. Таких ситуаций три:

[[a, b, c], [], []];
[[], [a, b, c], []];
[[], [], [a, b, c]].

Пример 79: решение задачи о перемещении кубиков.

```
domains
list=symbol*
% описывает состояние одного столбика кубиков
sit=list*
% описывает состояние всех столбиков
sits=sit*
% описывает путь из начальной вершины в целевую вершину
predicates
after(sit,sit)
% определяет переход из одной вершины пространства состояния в другую, то есть описывает все возможные правила перекладывания кубиков
solve (sit, sit, sits, sits)
% определяет путь для решения задачи
member (list, sit)
% первый предикат ищет список в списке списков
member1(sit, sits)
% второй предикат ищет список списков в списке списков списков
writesp(sits)
% распечатывает путь
clauses
member(X, [X|_]):-!.
member(X,[_|T]):-member(X,T).
member1(X, [X|_]):-!.
member1(X,[_|T]):-member1(X,T).
after([St11,St12,St13],S):-St13=[H3|T3],S=[St11,[H3|St12],T3].
after([St11,St12,St13],S):-St13=[H3|T3],S=[[H3|St11],St12,T3].
after([St11,St12,St13],S):-St12=[H2|T2],S=[[H2|St11],T2,St13].
after([St11,St12,St13],S):-St12=[H2|T2],S=[St11,T2,[H2|St13]].
after([St11,St12,St13],S):-St11=[H1|T1],S=[T1,[H1|St12],St13].
after([St11,St12,St13],S):-St11=[H1|T1],S=[T1,St12,[H1|St13]].
solve(S,S1,Sp,[S1|Sp]):-after(S,S1), member([a,b,c],S1).
```

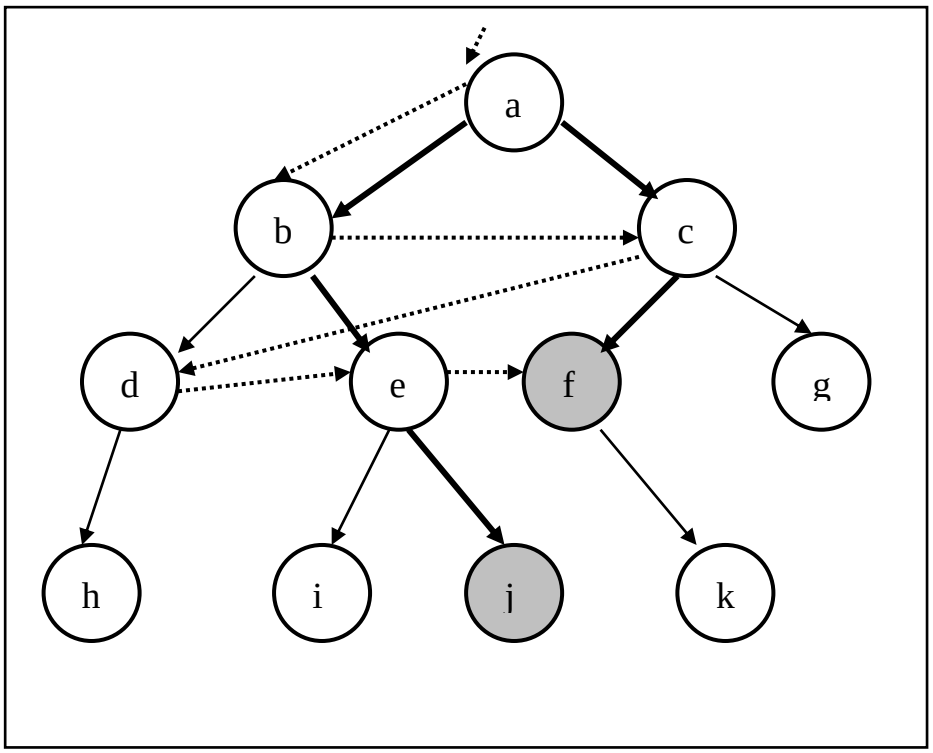
```
solve(S,S2,Sp,Sp2):-after(S,S1), not(member([a,b,c],S1)),
                    not(member1(S1,Sp)),solve(S1,S2,[S1|Sp],Sp2).
writesp([]).
writesp([H|T]):-writesp(T),write(H),nl.
goal
solve([[c,a,b],[],[]],S,[[c,a,b],[],[]],Sp),writesp(Sp).
```

(Вставить другую программу переупорядочения кубиков).

В данном примере реализован усовершенствованный алгоритм поиска в глубину, в котором добавлен алгоритм обнаружения циклов. Предикат *solve* включает очередную вершину в решающий путь только в том случае, если она еще не встречалась раньше.

4.2.2 Поиск в ширину

В противоположность поиску в глубину стратегия поиска в ширину предусматривает переход в первую очередь к вершинам, ближайшим к начальной вершине. На следующем рисунке показан пример, который иллюстрирует работу алгоритма поиска в ширину.



Поиск в ширину программируется не так легко, как поиск в глубину. Причина состоит в том, что приходится сохранять все множество альтернативных вершин-кандидатов, а не только одну вершину как при поиске в глубину. Более того, если при помощи процесса поиска необходимо получит решающий путь, то следует хранить не множество вершин-кандидатов, а множество путей-кандидатов. Для представления множества путей-кандидатов обычно используют списки, однако при таком способе одинаковые участки путей хранятся в нескольких экземплярах. Избежать подобной ситуации можно, если представить множество путей-кандидатов в виде дерева, в котором общие участки путей хранятся в его верхней части без

дублирования. При реализации стратегии поиска в ширину решающие пути порождаются один за другим в порядке увеличения их длин, следовательно, стратегия поиска в ширину гарантирует получение кратчайшего решения первым.

Представленные выше стратегии поиска в глубину и поиска в ширину не учитывают стоимости, приписанной дугам в пространстве состояний. Если критерием оптимальности является минимальная стоимость пути, а не его длина, то в данном случае поиск в ширину не решает поставленную задачу.

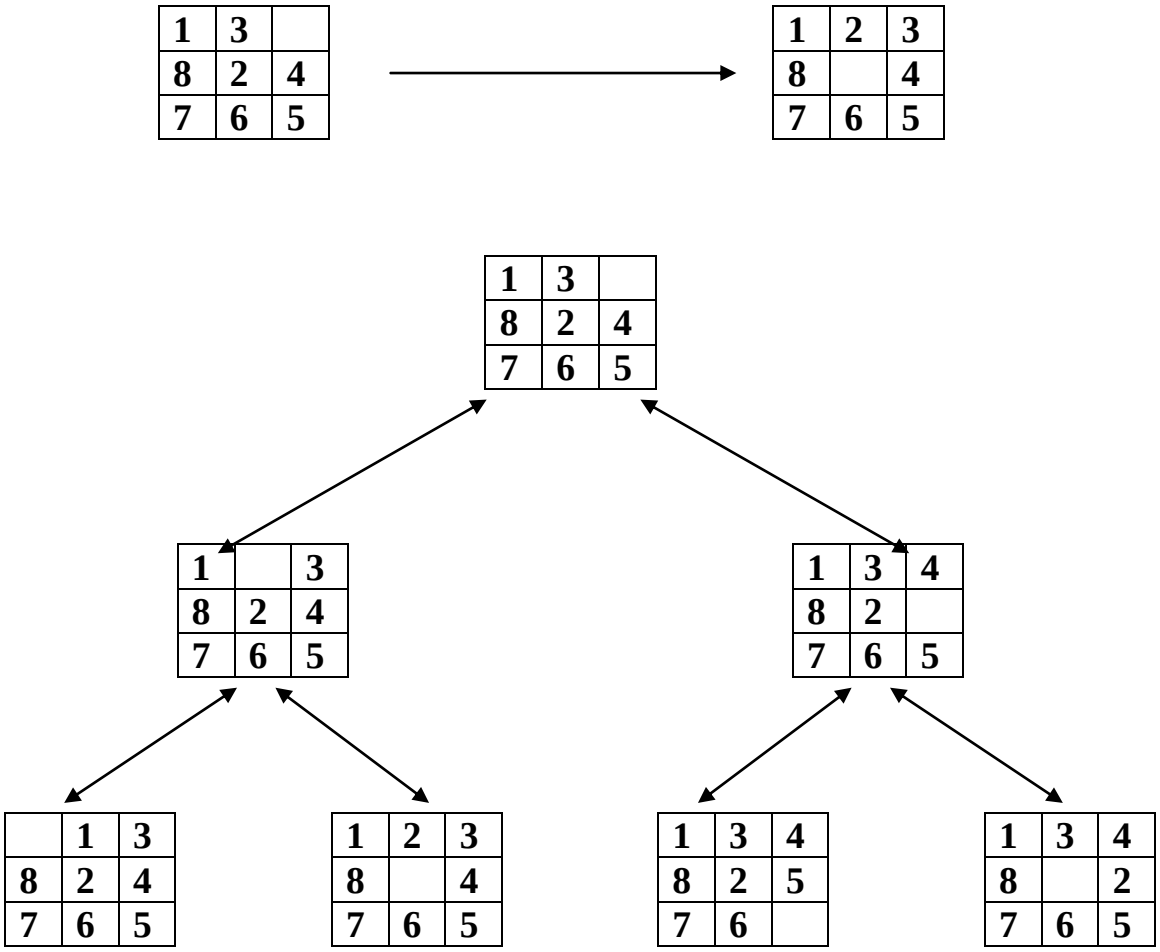
Еще одна проблема, возникающая при решении задачи поиска – это проблема *комбинаторной сложности*. Для сложных предметных областей число альтернатив столь велико, что проблема сложности часто принимает критический характер, так как длина решающего пути (тем более, если их множество, как при реализации поиска в ширину) может привести к *экспоненциальному росту длины в зависимости от размерности задачи*, что приводит к ситуации, называемой *комбинаторным взрывом*. Стратегии поиска в глубину и в ширину недостаточно «умны» для борьбы с такой ситуацией, так как все пути рассматриваются как одинаково перспективные.

По-видимому, процедуры поиска должны использовать какую-либо информацию, отражающую специфику данной задачи, с тем, чтобы на каждой стадии поиска принимать решения о наиболее перспективных путях поиска. В результате процесс будет продвигаться к целевой вершине, обходя бесполезные пути. Информация, относящаяся к конкретной решаемой задаче и используемая для управления поиском, называется *эвристикой*. Алгоритмы поиска, *использующие эвристики*, называют *эвристическими алгоритмами*.

Одним из эвристических алгоритмов решения задач является алгоритм A*, который является усовершенствованным алгоритмом поиска в ширину. Это, так называемый, алгоритм поиска по заданному критерию. Для каждого возможного перехода за один шаг определяется эвристическая оценка и для продолжения поиска решающего пути выбирается наилучшая в соответствии с данной оценкой вершина.

Предполагается, что для всех дуг в пространстве состояний определена функция стоимости перемещения от вершины к вершинам-преемникам. Допустим, для эвристической оценки применяется функция *f(n)* такая, что для каждой вершины *n* она служит для оценки «сложности достижения n». Соответственно наиболее перспективной является та вершина, для которой значение функции *f(n)* является минимальным. Рассмотрим алгоритм A* на примере решения задачи «игра в восемь».

На следующем рисунке представлена задача «игра в восемь» в виде задачи поиска пути в пространстве состояний. В головоломке используется восемь перемещаемых фишек, пронумерованных цифрами от 1 до 8. Фишки располагаются в девяти ячейках, образующих матрицу 3x3. Одна из ячеек всегда пуста, любая смежная с ней фишка может быть передвинута в эту ячейку. Конечная ситуация – это некоторая заранее заданная конфигурация фишек.



При этом можно выделить четыре основных оператора:

1. Перемещение пустой фишки вниз;
2. Перемещение пустой фишки вверх;
3. Перемещение пустой фишки влево;
4. Перемещение пустой фишки вправо.

Оценочная функция *f(n)* формируется как стоимость оптимального пути к цели из начального состояния через *n* вершин дерева поиска. Значение оценочной функции для вершины *n* равно: *f(n)=g(n)+h(n)*, где *g(n)* – стоимость оптимального пути от начальной вершины до *n*-ой, а *h(n)* – стоимость оптимального пути от *n*-ой вершины до целевой. Понятно, что *h(n)* это эвристическая гипотеза, основанная на имеющейся информации о данной конкретной задаче.

При этом *g(n)* принимается равной глубине пройденного пути на дереве поиска от начальной вершины до *n*-ой, а *h(n)* – расстояние Хемминга от *n*-ой вершины до целевой (в данном случае оно равно числу фишек, стоящих не на своих местах). Существует модификация алгоритма A*, в которой представляет сумму манхэттеновских расстояний (считается как сумма расстояний в горизонтальном и вертикальном направлениях) от каждой фишки до её «целевой клетки» плюс утроенное значение «оценки упорядоченности». Оценка упорядоченности определяет степень

упорядоченности фишек текущей позиции по отношению к целевой позиции и высчитывается по следующим правилам:

- Фишка в центре имеет оценку 1;
- Фишка в другой позиции имеет оценку 0, если за ней в направлении по часовой стрелке следует соответствующий ей преемник;
- Фишка в другой позиции имеет оценку 2, если за ней в направлении по часовой стрелке не следует соответствующий ей преемник.

Стратегия выбора следующей вершины в пространстве состояний – минимальное значение оценочной функции.

Формулировка алгоритма:

1. Рассматриваем все варианты перемещения пустой фишки за один шаг и выбираем вариант с минимальной оценкой *h(n).*
2. Переходим в новое состояние.
3. Создаём вершины следующего уровня иерархии.
4. Выбираем состояние с минимальной оценкой *h(n).*
5. Повторяем до тех пор, пока не достигнем цели.

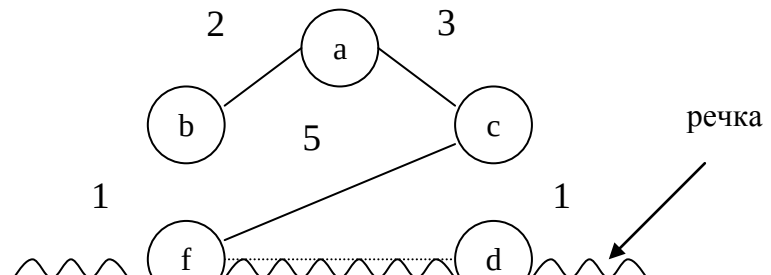
Цель не будет достигнута до тех пор, пока число перемещений меньше числа фишек, находящихся не на своих местах.

Для задачи, изображённой на рисунке, алгоритм находит решение за два шага. От начальной вершины возможен переход в два состояния с оценочной стоимостью f(1)=n+h(1)=1+4+3*(1+2)=14 и f(2)=1+3+3*(1+2+2)=19. Выбираем минимальную стоимость и переходим в состояние 1. Далее генерируем вершины 3 и 4 с оценочными стоимостями соответственно f(3)=2+2+3*(1+2+2)=19 и f(4)=2+0=2. Последняя вершина является целевой.

В соответствии с модифицированным алгоритмом оценочные стоимости вершин на первом шаге равны f(1)=n+h(1)=1+2=3 и f(2)=1+3=4

4.3 Сведение задачи к подзадачам и И/ИЛИ графы.

Для некоторых категорий задач более естественным решением является разбиение задачи на подзадачи. Разбиение на подзадачи дает преимущество в том случае, когда подзадачи взаимно независимы, и, следовательно, решать их можно независимо друг от друга. Проиллюстрируем это на примере решения задачи поиска на карте дорог маршрута между заданными городами как показано на рисунке.



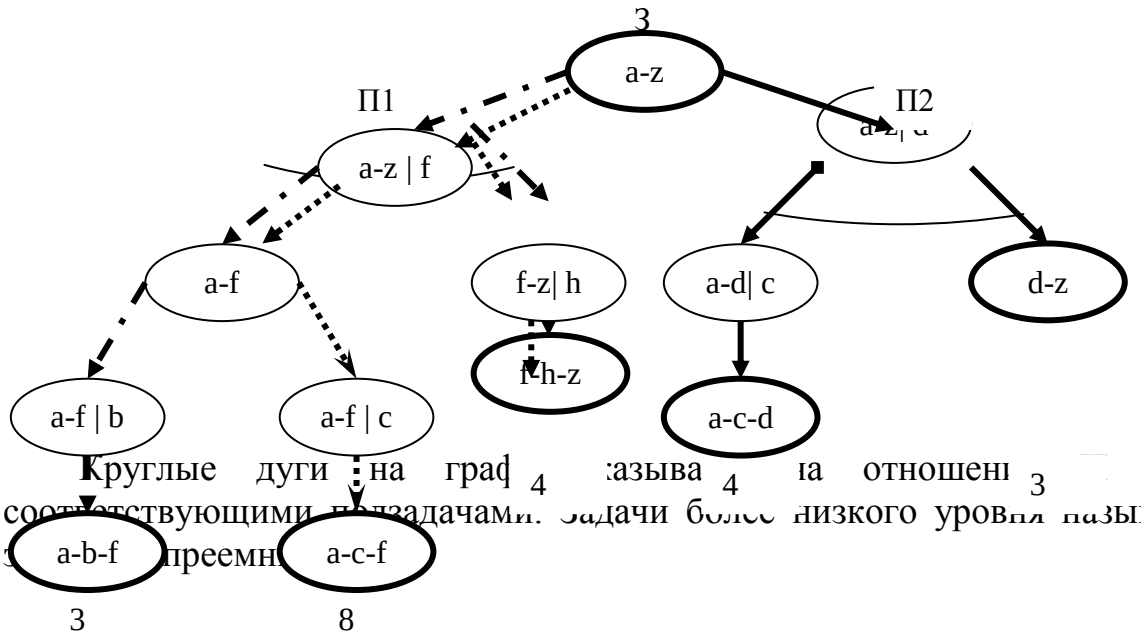
Вершинами a,b,c,d,f,h,z – представлены города. Расстояние между городами обозначено весом дуги из одной вершины графа в другую. На карте есть река. Допустим, что переправиться через реку можно только по двум мостам: один находится в городе f, а второй – в городе d. Очевидно, что искомый маршрут обязательно должен проходить через один из мостов, а значит должен проходить либо через f, либо через d. Таким образом, мы имеем две главные альтернативы:

- путь из a в z, проходящий через f;
- путь из a в z, проходящий через d.

Затем, каждую из этих двух альтернативных задач можно, в свою очередь, разбить следующим образом:

1. для того, чтобы найти путь из a в z через f, необходимо:
- найти путь из a в f и
- найти путь из f в z.
2. для того, чтобы найти путь из a в z через d, необходимо:
- найти путь из a в d и
- найти путь из d в z.

Таким образом, в двух альтернативах мы получили четыре подзадачи, которые можно решать независимо друг от друга. Полученное разбиение исходной задачи можно изобразить в форме И/ИЛИ – графа, представленного на рисунке.



Круглые дуги на граф[illegible]азыва[illegible]ia отношени[illegible] между соответствующими подзадачами. Задачи более низкого уровня называются [illegible] преемн[illegible]

И/ИЛИ- граф- это направленный граф, вершины которого соответствуют задачам, а дуги – отношениям между задачами.

Между дугами также существуют свои отношения – это отношения И и ИЛИ, в зависимости от того, должны ли мы решить только одну из задач-преемников или же несколько из них. В принципе из верщины могут выходить дуги, находящиеся в отношении И вместе с дугами, находящимися в отношении ИЛИ. Тем не менее, будем предполагать, что каждая верщина имеет либо только И-преемников, либо только ИЛИ-преемников, так как в такую форму можно преобразовать любой И/ИЛИ- граф, вводя в него при необходимости вспомогательные ИЛИ-вершины. Вершину, из которой выходят только И- дуги называются *И- вершиной*; вершину, из которой выходят только ИЛИ- дуги,- *ИЛИ- вершиной.*

Решением задачи, представленной в виде И/ИЛИ- графа является решающее дерево, так как решение должно включать в себя все подзадачи И-вершин.

Решающее дерево Т определяется следующим образом:

- исходная задача Р – это корень дерева Т;
- если Р является ИЛИ-вершиной, то в Т содержится только один из ее преемников (из И/ИЛИ-графа) вместе сос своим собственным решающим деревом;
- если Р – это И-вершина, то все ее преемники (из И/ИЛИ-графа) вместе со своими решающими деревьями содержатся в Т.

На представленном выше И/ИЛИ- графе представлены три решающих дерева, обозначенных штих-пунктирной, пунктирной и сплошной линиями. Соответственно, стоимости данных деревьев составляют 7, 12, 7. В данном случае стоимости определены как суммы стоимостей всех дуг дерева. Иногда стоимость решающего дерева определяется сумой стоимостей всех его вершин. В соответствии с заданным критерием, из всех решающих деревьев выбирается оптимальное.

4.4 Решение игровых задач в терминах И/ИЛИ- графа

Такие игры, как шахматы или шашки, естественно рассматривать как задачи, представленные И/ИЛИ- графами. Игры такого рода называются играми двух лиц с полной информацией. Будем считать, что существует только два возможных исхода игры:

- выигрыш;
- проигрыш.

Игры с тремя возможными исходами можно свести к играм с двумя исходами, считая, что есть: *выигрыш и невыигрыш*.

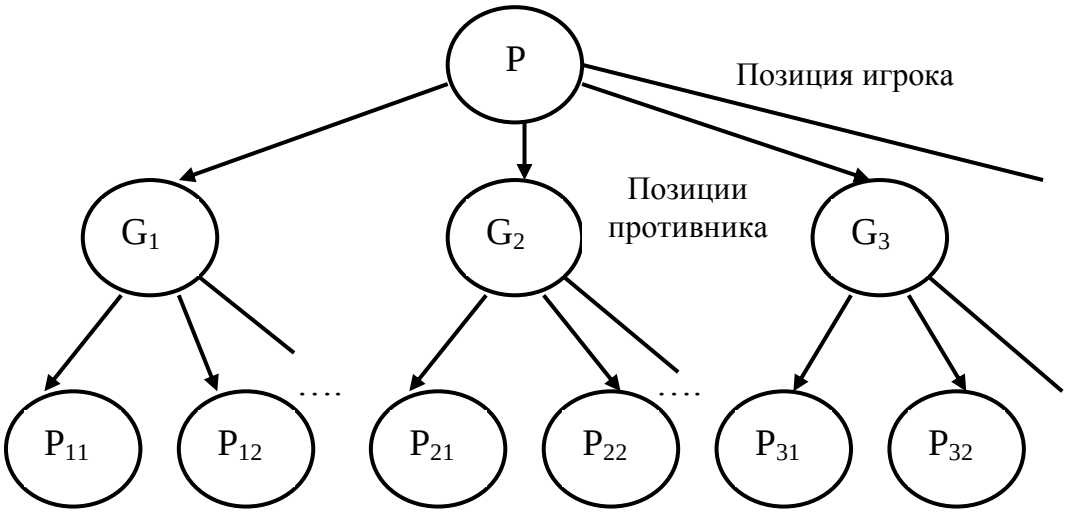
Так как участники игры ходят по очереди, то выделим два вида позиций, в зависимости от того, чей ход:

- позиция игрока;
- позиция противника.

Допустим, что начальная позиция Р – это позиция игрока. Каждый

вариант хода игрока в этой позиции приводит к одной из позиций противника G_1, G_2, G_3 и так далее. Каждый вариант хода противника в позиции G_i приводит к одной из позиций игрока P_{ij}.

В И/ИЛИ- дереве, показанном на рисунке, вершины соответствуют позициям, а дуги – возможным ходам. Уровни позиций игрока чередуются в дереве с уровнями позиций противника. Игрок выигрывает в позиции P, если он выигрывает в G_1, G_2, G_3 и так далее. Следовательно, P – это ИЛИ-вершина. Позиции G_i – это позиции противника, поэтому если в этой позиции выигрывает игрок, то он выигрывает и после каждого варианта хода противника, то есть игрок выигрывает в G_i, если он выигрывает во всех позициях P_{ij}. Таким образом, все позиции противника – это И-вершины. Целевые позиции – это позиции, выигранные согласно правилам игры. Для того, чтобы решить игровую задачу, мы должны построить решающее дерево, гарантирующее победу игрока независимо от ответов противника. Такое дерево задает полную стратегию достижения выигрыша: для каждого возможного продолжения, выбранного противником, в дереве стратегии есть ответный ход, приводящий к победе.



Рассмотрим решение подобных задач на примере игры в «2 лунки».

Игрок или его противник может взять из одной любой лунки любое количество камешков. Проигрывает тот, кто берет последний камешек.

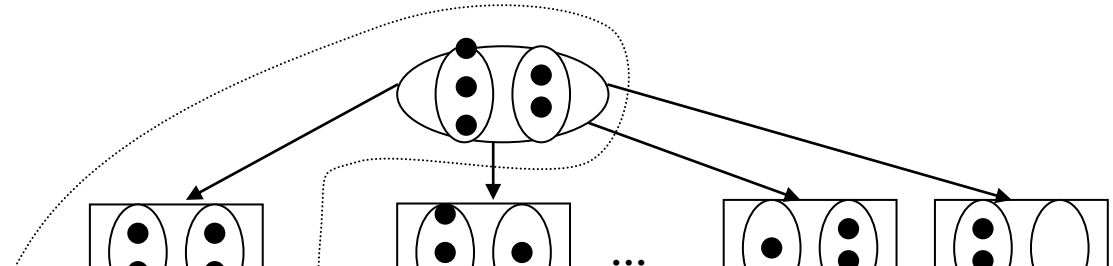
- позиция противника

- позиция игрока

- выигрыш игрока

- проигрыш игрока

Дерево решений этой игры представлено на рисунке.



Пунктирной линией обозначена оптимальная стратегия игрока, которая приведет к выигрышу**.**

4.5 Минимаксный принцип поиска решений

Алгоритмы поиска пути на И/ИЛИ- графах могут использовать стратегии поиска в глубину и ширину, однако, для большинства игр, дерево игры имеет большое количество позиций, что приводит к комбинаторному взрыву при реализации просмотра всех вершин дерева решений.

Основной подход к организации поиска на игровых деревьях использует оценочные функции. Оценочная функция используется для вычисления оценки текущего состояния игры.

Для выбора следующего хода используется простой алгоритм:

- найти всевозможные состояния игры, которые могут быть достигнуты за один ход;
- используя оценочную функцию, вычислить оценки состояний;
- выбрать ход, ведущий к позиции с наивысшей оценкой.

Если оценочная функция была бы совершенной, то есть ее значение отражало бы какие позиции ведут к победе, а какие – к поражению, то достаточно было бы просмотра вперед на один шаг. Обычно совершенная оценочная функция неизвестна, поэтому стратегия выбора хода на основе просмотра на один шаг вперед не дает хорошего результата, поэтому используется стратегия просмотра на несколько шагов вперед.

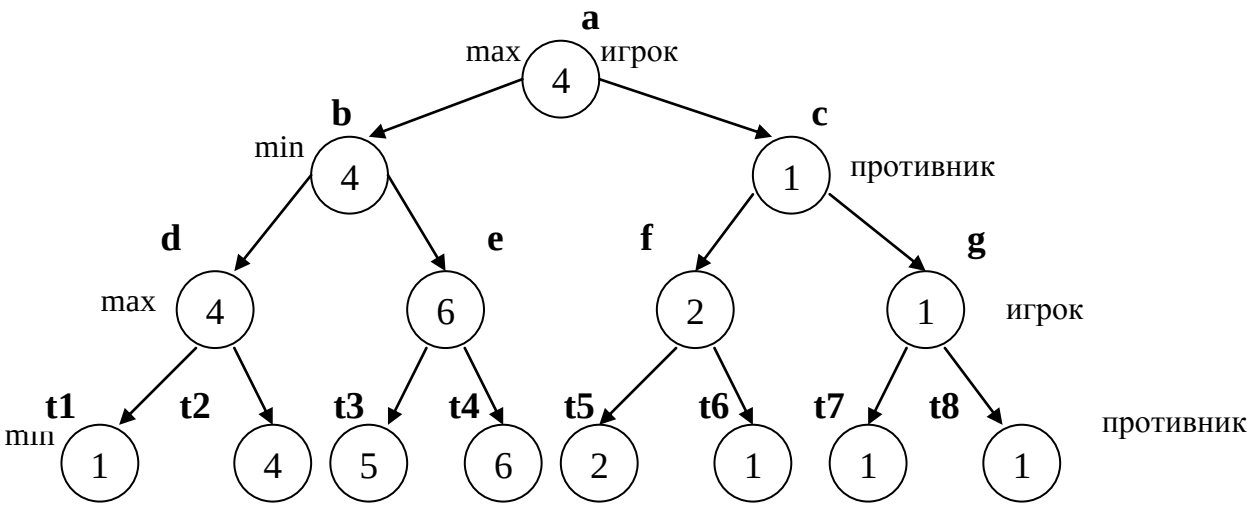
Стандартный метод определения оценки позиции, основанный на просмотре вперед нескольких слоев игрового дерева, называется минимаксным алгоритмом.

Минимаксный алгоритм предполагает, что противник из нескольких возможных ходов сделает выбор, лучший для себя , то есть худший для

игрока. Поэтому целью игрока является выбор такого хода, который даст максимальную оценку позиции, возможной после лучшего хода противника, то есть минимизирующего оценку позиции противника. Отсюда название – минимаксный алгоритм. Число слоев игрового дерева, просматриваемых при поиске, зависит от доступных ресурсов. На последнем слое используется оценочная функция.

В предположении, что оценочная функция выбрана разумно, алгоритм будет давать тем лучшие результаты, чем больше слоев просматривается при поиске.

Пусть мы имеем следующее дерево игры:



Задана некая оценочная функция $\varphi(P_k)$, где P_k- некоторая игровая ситуация.

Предположим, что игрок максимизирует свой выигрыш, а противник минимизирует свой проигрыш. Вариант решения, образованный минимаксной стратегией движения по дереву игры, будем называть основным вариантом решения.

Если существует оценочная функция, то можно ввести внутреннюю функцию $\varphi(P_k)$ такую, что:

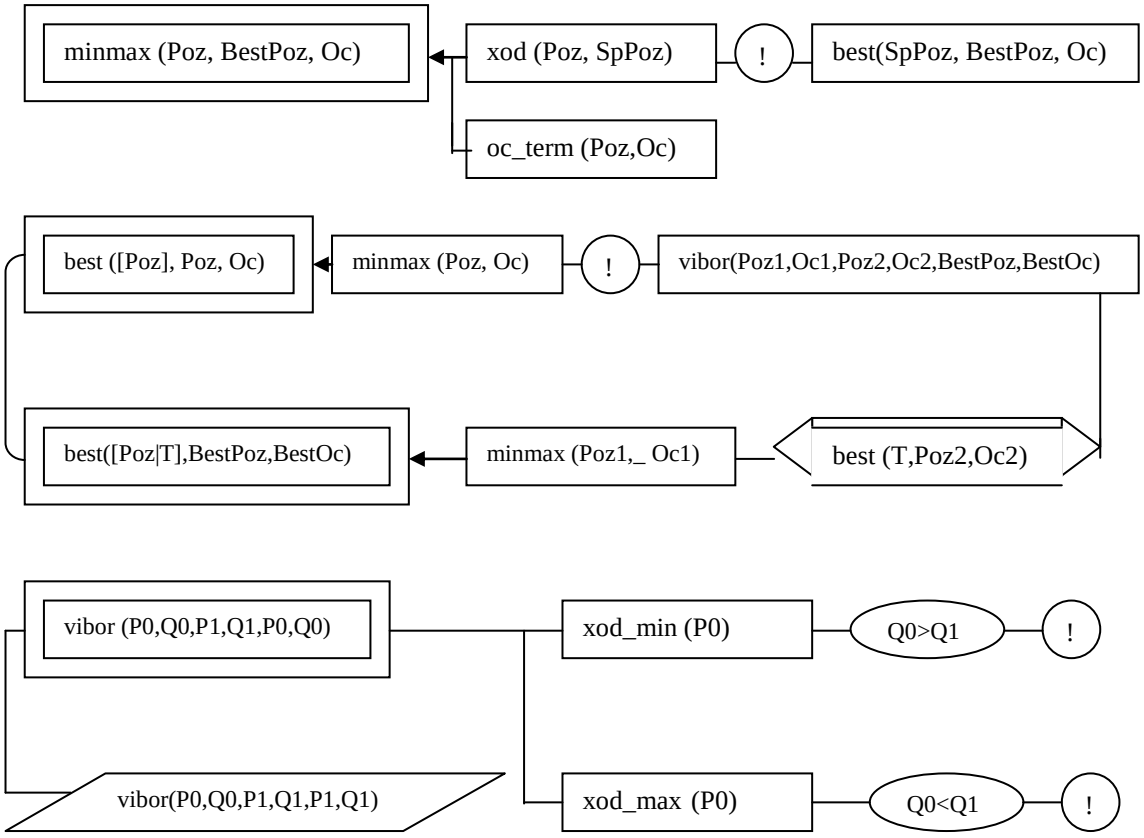
$$\varphi(p_k)=\begin{cases} \max\ \varphi(p_k) \to p_k - \max\ вершина \\ \min\ \varphi(p_k) \to p_k - \min\ вершина \\ \varphi(p_k) \to p_k - терминальная\quad вершина \end{cases}$$

Пример 78:

```
trace
domains
pozic = symbol
spoz = pozic*
database
xod (pozic, spoz)
xod_min (pozic)
```

```
xod_max (pozic)
predicates
minmax (pozic, pozic, integer)
best (spoz, pozic, integer)
oc_term(pozic, integer)
vibor(pozic, integer, pozic, integer, pozic, integer)
clauses
xod (a, [b,c]).
xod (b, [d,e]).
xod (c, [f,g]).
xod (d, [t1,t2]).
xod (e, [t3,t4]).
xod (f, [t5,t6]).
xod (g, [t7,t8]).
xod_max (a).
xod_max (d).
xod_max (e).
xod_max (f).
xod_max (g).
xod_min (b).
xod_min (c).
xod_min (t1).
xod_min (t2).
xod_min (t3).
xod_min (t4).
xod_min (t5).
xod_min (t6).
xod_min (t7).
xod_min (t8).
oc_term (a,4).
oc_term (b,4).
oc_term (c,1).
oc_term (d,4).
oc_term (e,6).
oc_term (f,2).
oc_term (g,1).
oc_term (t1,1).
oc_term (t2,4).
oc_term (t3,5).
oc_term (t4,6).
oc_term (t5,2).
oc_term (t6,1).
oc_term (t7,1).
oc_term (t8,1).
minmax (Poz, BestPoz, Oc):-
```

```
            xod (Poz, SpPoz),!,
            best(SpPoz, BestPoz, Oc);
            oc_term(Poz, Oc).
best ([Poz], Poz, Oc):- minmax (Poz, _ , Oc), !.
best ([Poz1| T], BestPoz, BestOc):-
            minmax (Poz1, _ , Oc1),
            best (T, Poz2, Oc2),
            vibor(Poz1,Oc1,Poz2,Oc2,BestPoz,BestOc).
vibor(Poz0, Oc0, Poz1, Oc1, Poz0, Oc0):-
            xod_min (Poz0), Oc0>Oc1,!;
            xod_max (Poz0), Oc0<Oc1,!.
vibor(Poz0, Oc0, Poz1, Oc1, Poz1, Oc1).
goal
minmax(a,BestPoz,Oc),write(BestPoz),write(Oc).
```



5 Введение в экспертные системы

5.1 Основные понятия

Экспертные системы (ЭС) как самостоятельное направление в искусственном интеллекте (ИИ) сформировалось в конце 1970-х годов. История ЭС началась с сообщения японского комитета по разработке ЭВМ пятого поколения, в котором основное внимание уделялось развитию «интеллектуальных способностей» компьютеров с тем, чтобы они могли оперировать не только данными, но и знаниями.

Область исследования ЭС называется *инженерией знаний*. Этот термин был введён Е. Фейгенбаумом и в его трактовке означает «привнесение принципов и инструментария из области искусственного интеллекта в решение трудных прикладных проблем, требующих знаний экспертов. Не каждую систему, основанную на знаниях, можно рассматривать как экспертную. Экспертная система должна быть способна в определённой степени объяснять своё поведение и свои решения пользователю [1].

Таким образом, экспертные системы предназначены для решения неформализованных задач, то есть задач, решаемых с помощью неточных знаний, которые являются результатом обобщения многолетнего опыта работы и интуиции специалистов.

Неформализованные знания обычно представляют собой эвристические приемы и правила. ЭС обладают следующими особенностями (всеми сразу или частично) [2]:

- задачи не могут быть представлены в числовой форме;
- исходные данные и знания о предметной области обладают неоднозначностью, неточностью, противоречивостью;
- цели нельзя выразить с помощью чётко определённой целевой функции;
- не существует однозначного алгоритмического решения задачи;
- алгоритмическое решение существует, но его нельзя использовать по причине большой размерности пространства решений и ограничений на ресурсы.

ЭС охватывают самые разные предметные области, среди которых преобладают медицина, бизнес, производство, проектирование и системы управления.

Для классификации ЭС используются следующие признаки:

- способ формирования решения;
- способ учёта временного признака;
- вид используемых данных и знаний;
- число используемых источников знаний.

По *способу формирования решения* ЭС можно разделить на анализирующие и синтезирующие. В системах первого типа осуществляется

выбор решения из множества известных решений на основе анализа знаний, в системах второго типа решение синтезируется из отдельных фрагментов знаний.

В зависимости от *способа учёта временного признака* ЭС делятся на статические и динамические. Статические ЭС предназначены для решения задач с неизменяемыми в процессе решения данными и знаниями, а динамические ЭС допускают такие изменения.

По *видам используемых данных и знаний* различают ЭС с детерминированными и неопределёнными знаниями. Под неопределённостью знаний понимаются их неполнота, ненадёжность, нечёткость.

ЭС могут создаваться с использованием одного или нескольких *источников знаний*.

В соответствии с перечисленными признаками можно выделить четыре основных класса ЭС: классифицирующие, доопределяющие, трансформирующие и мультиагентные [2].

Классифицирующие ЭС решают задачи распознавания ситуаций. Основным методом формирования решений в них является дедуктивный логический вывод.

Доопределяющие ЭС используются для решения задач с не полностью определенными данными и знаниями. В таких ЭС возникают задачи интерпретации нечётких знаний и выбора альтернативных направлений поиска в пространстве возможных решений. В качестве методов обработки неопределенных знаний могут использоваться байесовский вероятностный подход, коэффициенты уверенности, нечеткая логика.

Трансформирующие ЭС относятся к синтезирующим динамическим ЭС, которые реализуют повторяющееся преобразование знаний в процессе решения задачи. В ЭС данного класса используются различные способы обработки знаний:

- генерация и проверка гипотез;
- логика предположений и умолчаний;
- использование метазнаний для устранения неопределенности в ситуациях.

Мультиагентные системы – это динамические ЭС, основанные на интеграции разнородных источников знаний, которые обмениваются между собой полученными результатами в процессе решения задач. Системы данного класса имеют следующие возможности:

- реализация альтернативных рассуждений на основе использования различных источников знаний и механизма устранения противоречий;
- распределённое решение задач, разделяемых на параллельно решаемые подзадачи с самостоятельными источниками знаний;
- применение различных стратегий вывода заключений в зависимости от типа решаемой задачи;
- обработка больших массивов информации из базы данных (БД);

- использование математических моделей и внешних процедур для имитации развития ситуаций.

Для формирования полноценной ЭС необходимо, как правило, реализовать в ней следующие функции:

- функции решения задач, позволяющие использовать специальные знания в проблемной области (при этом может потребоваться обеспечить работу в условиях неопределённости или неполноты или недостоверности знаний);
- функции взаимодействия с пользователем, которые, в частности, позволяют объяснить намерения и выводы системы в процессе решения задачи и по завершении этого процесса.

5.2 Проектирование экспертных систем

Первые ЭС были статического типа. Типичная статическая ЭС должна включать следующие компоненты [2]:

- базу знаний (БЗ);
- базу данных (рабочую память);
- решатель (интерпретатор);
- систему объяснений;
- компоненты приобретения знаний;
- интерфейс с пользователем.

БЗ ЭС предназначена для хранения долгосрочных данных, описывающих рассматриваемую область, и правил, описывающих целесообразные преобразования данных этой области.

БД ЭС служит для хранения текущих данных решаемой задачи.

Решатель формирует последовательность применения правил и осуществляет их обработку, используя данные из рабочей памяти и знания из БЗ.

Система объяснений показывает, каким образом система получила решение задачи, и какие знания при этом использовались. Это облегчает тестирование системы и повышает доверие пользователя к полученному результату.

Компоненты приобретения знаний необходимы для заполнения ЭС знаниями в диалоге с пользователем-экспертом, а также для добавления и модификации заложенных в систему знаний.

К разработке ЭС привлекаются специалисты из разных предметных областей, а именно:

- эксперты той проблемной области, к которой относятся задачи, решаемые ЭС;
- инженеры по знаниям, являющиеся специалистами по разработке интеллектуальных информационных систем (ИИС);
- программисты, осуществляющие реализацию ЭС.

Любая ЭС должна иметь, по крайней мере, два режима работы:

- режим приобретения знаний;

- режим консультаций.

В *режиме приобретения знаний* эксперт наполняет ЭС знаниями, которые позволят ЭС в дальнейшем решать конкретные задачи из описанной проблемной области. Эксперт описывает проблемную область в виде совокупности данных об объектах и правил, определяющих взаимные связи между данными, и способы манипулирования данными.

В *режиме консультаций* пользователь ЭС сообщает системе конкретные данные о решаемой задаче и стремится получить с её помощью результат. При этом входные данные о задаче поступают в рабочую память. Решатель на основе данных из БД и правил из БЗ формирует решение.

Динамические ЭС, наряду с компонентами статических ЭС, должны содержать:

- подсистему моделирования внешнего мира;
- подсистему связи с внешним окружением.

Подсистема моделирования необходима для прогнозирования, анализа и адекватной оценки состояния внешней среды. Изменения окружения решаемой задачи требуют изменения хранимых в ЭС знаний, для того чтобы отразить временную логику происходящих в реальном мире событий.

5.3 Типы решаемых задач

ЭС могут решать следующие задачи [2]:

- анализа и синтеза. В задаче *анализа* задана модель сущности и требуется определить неизвестные характеристики модели. В задаче *синтеза* задаются условия, которым должны удовлетворять характеристики «неизвестной» модели сущности, требуется построить модель этой сущности;
- статические или динамические. Если ЭС явно не учитывает фактор времени и / или не изменяет в процессе решения знания об окружающем мире, то ЭС решает *статические* задачи, в противном случае – *динамические* (работающие в реальном масштабе времени). Обычно выделяют следующие системы реального времени: *псевдореального* времени, *«мягкого»* реального времени и *«жёсткого»* реального времени. Системы псевдореального времени получают и обрабатывают данные, поступающие из внешних источников.

ЭС могут решать следующие типы задач:

- *интерпретации данных* – процесса определения смысла данных;
- *диагностики* – процесса соотнесения объекта с некоторым классом объектов и / или обнаружения неисправностей;
- *мониторинга* – непрерывной интерпретации данных в реальном масштабе времени и контроле допуска их параметров;
- *проектирования* – создания ранее не существовавшего объекта и подготовки спецификаций на создание объектов с заранее определёнными свойствами;

- *прогнозирования* – предсказания последствий некоторых событий или явлений на основе анализа имеющихся данных;
- *планирования* – построения планов действий объектов, способных выполнять некоторые функции;
- *обучения* каким-либо дисциплинам или предметам;
- *управления* – поддержки определённого режима деятельности системы;
- *поддержки принятия решений*.

Задачи интерпретации данных, диагностики, поддержки принятия решений относятся к задачам анализа, задачи проектирования, планирования и управления – к задачам синтеза, остальные задачи – комбинированного типа.

5.4 Инструментальные средства разработки экспертных систем

Классификация инструментальных средств разработки ЭС обычно производится по следующим параметрам [2]:

- уровень используемого языка;
- парадигмы программирования и механизмы реализации;
- способ представления знаний;
- механизмы вывода и моделирование;
- средства приобретения знаний;
- технологии разработки.

Уровень используемого языка:

- традиционные (в том числе и объектно-ориентированные) языки программирования;
- специальные языки программирования (LISP, PROLOG, РЕФАЛ);
- инструментальные средства, содержащие часть компонентов ЭС (предназначены для разработчиков ЭС);
- среды разработки общего назначения, содержащие все компоненты ЭС, но не имеющие описания конкретных проблемных сред;
- проблемно-ориентированные среды разработки (для решения определённого класса задач или имеющие знания о типах предметных областей).

Парадигмы программирования:

- процедурное программирование;
- программирование, ориентированное на данные;
- программирование, ориентированное на правила;
- объектно-ориентированное программирование;
- логическое программирование.

Способ (модели) представления знаний:

- продукционные правила;
- фреймы (объекты);
- логические формулы;

- семантические сети;
- нейронные сети.

Механизмы вывода и моделирования:

1. Моделирование процесса получения решения:

- построение дерева вывода на основе обучающей выборки и выбор маршрута на дереве вывода в режиме решения задачи;
- компиляция сети вывода из специфических правил в режиме приобретения знаний и поиск решения на сети в режиме решения задачи;
- генерация сети вывода и поиск решения в режиме решения задачи, при этом генерация сети вывода осуществляется в ходе выполнения сопоставления, определяющей пары «правило-совокупность данных», на которых условия этого правила удовлетворяются;
- в режиме решения задачи ЭС осуществляет выработку правдоподобных предположений (при отсутствии достаточной информации для решения), выполнение рассуждений по обоснованию предположений, генерацию альтернативных сетей вывода, поиск решения в сетях вывода;
- построение сети вывода на основе обучающей выборки и поиск решения на выходах сети в режиме решения задачи;

2. Механизмы поиска решения:

- двунаправленный поиск, поиск от данных к целям, поиск от целей к данным;
- «поиск в ширину», «поиск в глубину».

3. Механизмы генерации предположений и сети вывода:

- генерация в режиме приобретения знаний, генерация в режиме решения задачи;
- операция сопоставления применяется ко всем правилам и всем типам сущностей в каждом цикле механизма вывода, используются различные средства сокращения правил и/ или сущностей.

Механизм вывода для динамических сред дополнительно содержит планировщик, управляющий деятельностью ЭС в соответствии с приоритетами; средства получения оптимального решения в условиях ограниченности ресурсов; систему поддержания истинности значений переменных, изменяющихся во времени.

Средства приобретения знаний:

1. Уровень приобретения знаний:

- формальный язык;
- ограниченный естественный язык;
- язык пиктограмм и изображений;
- естественный язык и язык изображений;

2. Тип приобретаемых знаний:

- данные в виде таблиц, содержащих значения входных и выходных атрибутов, по которым индуктивными методами строится дерево вывода;
- специализированные правила;
- общие и специализированные правила;
- данные в виде таблиц, содержащих значения входных и выходных векторов, по которым строится сеть вывода.

3. Тип приобретаемых данных:

- атрибуты и значения;
- объекты;
- классы структурированных объектов и их экземпляры, получающие значения атрибутов путём наследования.

5.5 Нечёткие знания в экспертных системах

При разработке ИИС знания о конкретной предметной области, для которой создаётся система, редко бывают полными и абсолютно достоверными. Знания, которыми заполняются ЭС, получаются в результате опроса экспертов, мнения которых субъективны. Даже числовые данные, полученные в ходе экспериментов, имеют статистические оценки достоверности, надёжности, значимости и так далее.

Смысл термина *нечёткость* многозначен. Основными компонентами нечётких знаний можно считать следующие понятия[2]:

- недетерминированность выводов;
- многозначность;
- ненадёжность;
- неполнота;
- неточность.

Недетерминированность выводов - это характерная черта большинства систем ИИ. Недетерминированность означает, что заранее путь решения конкретной задачи в пространстве её состояний определить невозможно. Поэтому методом проб и ошибок выбирается некоторая цепочка логических заключений, а в случае если она не приводит к успеху, организуется перебор с возвратом для поиска другой цепочки. Для решения подобных задач предложено множество эвристических алгоритмов, например, алгоритм A*.

Многозначность интерпретации – обычное явление в задачах распознавания графических образов, понимания естественного языка. Устранение многозначности достигается с помощью циклических операций фильтрации.

Ненадёжность знаний и выводов означает, что для оценки их достоверности нельзя применить двухбалльную шкалу (1 – абсолютно достоверные; 0 – недостоверные знания). Для более тонкой оценки применяется вероятностный подход, основанный на теореме Байеса, использование коэффициентов уверенности, использование нечётких выводов на базе нечёткой логики.

Неполнота знаний и немонотонная логика. При добавлении знаний в БЗ возникает опасность получения противоречивых выводов, если система знаний не является полной. Как известно, формальная логическая система, основанная на логике предикатов первого порядка, является *полной*, при этом новые факты не нарушают истинность ранее полученных выводов. Это свойство логических выводов называется *монотонностью*. К сожалению, реальные знания в ЭС редко бывают полными, поэтому в качестве средств обработки неполных знаний, для которых необходимы немонотонные выводы, разрабатываются методы *немонотонной логики*. Известна немонотонная логика Макдермотта и Доула, логика умолчания Рейтера, немонотонная логика Маккарти. Для организации логических выводов в интеллектуальных системах с неполными знаниями вместо традиционной дедукции применяется *абдукция.* Абдукцией называется процесс формирования объясняющей гипотезы на основе заданной теории и имеющихся наблюдений (фактов).

Неточность знаний. Числовые данные могут быть неточными, при этом существуют оценки такой неточности (доверительный интервал, уровень значимости, степень адекватности и так далее). Лингвистические знания тоже могут быть неточными. Для учёта неточности лингвистических знаний используются нечёткая логика и нечёткие выводы, основанные на теории нечётких множеств, предложенной Л.Заде в 1965 году.

5.6 Продукционные правила для представления знаний.

Рассмотрим пример создания экспертной системы на основе использования продукционных правил. Правила типа «если-то», называемые продукциями, являются одним из наиболее популярных формализмов представления знаний. Каждое такое правило есть условное утверждение, однако, существуют различные варианты их интерпретации:

1) ЕСЛИ <условие P> ТО <заключение C>;
2) ЕСЛИ <ситуация S> ТО <действие A>;
3) ЕСЛИ <выполняются условия C1 и C2> ТО <не выполняется условие C3>.

Продукции обладают следующими свойствами:

- Модульность: каждое правило описывает небольшой, относительно независимый фрагмент знаний;
- Возможность инкрементного наращивания: добавление новых правил в базу знаний независимо от существующих правил;
- Удобство модификации (следствие модульности): старые правила можно изменять и заменять на новые независимо от других правил;
- Прозрачность системы как следствие применения правил.

Последнее свойство - это способность системы к объяснению принятых решений и полученных результатов. Применение «если - то» правил облегчает получение ответов на вопросы типа: «как?» и «почему?».

«Если - то»- правила часто применяют для определения логических

отношений между понятиями предметной области. Про чисто логические отношения можно сказать, что они принадлежат к «категорическим знаниям», то есть соответствующие им отношения абсолютно верны. Однако, в некоторых предметных областях преобладают вероятностные («мягкие») знания. Эти знания являются «мягкими» в том смысле, что говорить об их применимости к любым практическим ситуациям можно только до некоторой степени. В таких случаях используют модифицированные «если - то» – правила, дополняя их логическую интерпретацию вероятностной оценкой или использовать нечёткие продукционные правила.

Например, *если* условие A, *то* заключение B *с вероятностью* F.

Проиллюстрируем использование правил типа «если - то» на примере «игрушечной» базы знаний, помогающей идентифицировать животных по их основным признакам в предположении, что задача идентификации ограничена только небольшим числом разных животных.

Правило 1: *если*
Животное «имеет» шерсть
или
Животное «кормит детенышей» молоком
то
Животное - млекопитающее.

Правило 2: *если*
Животное «имеет» перья
или
Животное «летает » и
Животное «откладывает яйца»
то
Животное - птица.

Правило 3: *если*
Животное это млекопитающее и
Животное «ест» мясо
то
Животное - хищник.

Правило 4: *если*
Животное это хищник и
Животное «имеет» «рыжий цвет» и
Животное «имеет» «темные пятна»
то
Животное - «гепард».

Правило 5: *если*
Животное это хищник и
Животное «имеет» «рыжий цвет»
и
Животное «имеет» «черные полосы»
то
Животное - «тигр».

Правило 6: *если*

Животное это птица и
Животное «не может» «летать»
и
Животное «плавает»
то
Животное - «пингвин».

Правило 7: *если*

Животное это птица и
Животное «летает»
то
Животное - «альбатрос».

Факт: X это животное: - принадлежит (X, [гепард, тигр, пингвин, альбатрос]).

можно_спросить (Животное, «кормит детенышей», Чем).
можно_спросить (Животное, «откладывает яйца»).
можно_спросить (Животное, «ест», Что).
можно_спросить (Животное, «не может», Что делать).
можно_спросить (Животное, «плавает»).
можно_спросить (Животное, «летает»).

Если переписать данные правила в виде настоящих прологовских правил, то они примут вид:

Животное это млекопитающее :- Животное «имеет» «шерсть»,
Животное «кормит детенышей» «молоком».
Животное это хищник :- Животное это млекопитающее,
Животное «ест» «мясо».
Животное это тигр:- животное это хищник, животное «имеет» «рыжий цвет», животное «имеет» «черные полосы».

Добавим некоторые факты:

«Пушок» имеет «шерсть».
«Пушок» ленив.
«Пушок» имеет «рыжий цвет».
«Пушок» имеет «черные полосы».
«Пушок» ест «мясо».
?-«Пушок» это тигр.
Yes.
?-«Пушок» это гепард.
No.

Напишем на языке Пролог нашу базу знаний:

```
domains
list=string*
predicates
животное(string)
животное_млекопитающее(string)
животное_птица(string)
```

```
животное_хищник(string)
животное_гепард(string)
животное_тигр(string)
животное_пингвин(string)
животное_альбатрос(string)
животное_кормит_детёнышей(string, string)
животное_имеет(string, string)
животное_ест(string, string)
животное_ленив(string)
животное_откладывает_яйца(string)
животное_летает(string)
животное_плавает(string)
принадлежит(string,list)

clauses
животное_летает("альбатрос").
животное_плавает("пингвин").
животное_ест("тигр", "мясо").
животное_ест("гепард", "мясо").
животное_ест("альбатрос", "рыбу").
животное_ест("пингвин", "рыбу").
животное_ест("Пушок", "мясо").
животное_откладывает_яйца("пингвин").
животное_откладывает_яйца("альбатрос").
животное_кормит_детёнышей("тигр", "молоком").
животное_кормит_детёнышей("гепард", "молоком").
животное_кормит_детёнышей("Пушок", "молоком").
животное_имеет("гепард", "шерсть").
животное_имеет("гепард", "рыжий цвет").
животное_имеет("гепард", "тёмные пятна").
животное_имеет("тигр", "шерсть").
животное_имеет("тигр", "рыжий цвет").
животное_имеет("тигр", "чёрные полосы").
животное_имеет("пингвин", "перья").
животное_имеет("альбатрос", "перья").
животное_имеет("Пушок", "шерсть").
животное_имеет("Пушок", "рыжий цвет").
животное_имеет("Пушок", "чёрные полосы").
животное_ленив("Пушок").
животное(X):-принадлежит(X,["пингвин", "альбатрос", "тигр",
"перья","Пушок"]).
животное_млекопитающее(X):-животное(X),животное_имеет(X,
"шерсть"),
        животное_кормит_детёнышей(X,"молоком").
животное_птица(X):-животное(X),животное_имеет(X, "перья"),
```

животное_откладывает_яйца(X).
животное_хищник(X):-животное_млекопитающее(X), животное_ест(X, "мясо"),
животное_имеет(X, "шерсть").
животное_гепард(X):-животное_хищник(X), животное_имеет(X, "рыжий цвет"),
животное_имеет(X, "тёмные пятна").
животное_тигр(X):-животное_хищник(X), животное_имеет(X, "рыжий цвет"),
животное_имеет(X, "чёрные полосы").
животное_пингвин(X):- животное_птица(X), животное_плавает(X), животное_ест(X, "рыбу"),
not(животное_летает(X)).
животное_альбатрос(X):- животное_птица(X), животное_летает(X), животное_ест(X, "рыбу").
принадлежит(A,[A|_]).
принадлежит(A,[_|T]):-принадлежит(A,T).

goal
%животное_кормит_детёнышей(X, Y).
%животное_откладывает_яйца(X).
%животное_ест(X, Y).
%животное_летает(X).
%животное_плавает(X).
%животное_тигр("Пушок").
животное_гепард("Пушок").

Наша пролог-система отвечает на вопросы, используя базу знаний, однако, нельзя сказать, что ее поведение вполне соответствует поведению эксперта. Это происходит по двум причинам:

- Система не может объяснить свой ответ: как она установила, что «Пушок» тигр, а не гепард;
- В систему необходимо ввести всю необходимую информацию в виде фактов, при этом возможен ввод лишней информации («Пушок» ленив) или потеря необходимой информации.

Для того, чтобы исправить эти два недостатка, в системе должен быть реализован более совершенный способ взаимодействия между пользователем и системой во время и после завершения рассуждений, то есть необходимо создать свое средство интерпретации в виде специальной надстройки над пролог-системой.

Система должна взаимодействовать с пользователем в режиме диалога следующим образом:

Пожалуйста, спрашивайте:
«Пушок» это тигр?
Это правда: «Пушок» имеет шерсть?
Да.

Это правда: «Пушок» ест мясо?
Да.
«Пушок» это тигр.
Хотите узнать, как?
«Пушок» это хищник
выведено из
«Пушок» это млекопитающее
выведено из
«Пушок» имеет шерсть
было сказано
и
«Пушок» ест мясо
было сказано.

Как видно из диалога, система задает пользователю вопросы, касающиеся «примитивной» информации, например:

Это правда: «Пушок» ест мясо?

На подобные вопросы пользователь может отвечать двумя способами:

- сообщив системе в качестве ответа на вопрос необходимую информацию;
- спросив систему, *почему* эта информация необходима.

Последняя возможность позволяет пользователю заглянуть внутрь системы и увидеть ее намерения. Из объяснений системы пользователь поймет, стоит ли информация, которую запрашивает система, тех дополнительных усилий, которые необходимо приложить для ее приобретения. Для того, чтобы заглянуть внутрь системы, следует создать специальную надстройку над пролог - системой, которая будет включать в себя средства взаимодействия с пользователем.

Такая надстройка будет принимать вопрос и искать на него ответ. Язык правил допускает, чтобы в условной части правила была И/ИЛИ комбинация условий. Вопрос также может быть представлен в виде И/ИЛИ комбинаций подвопросов. Поэтому процесс поиска ответов на эти вопросы будет аналогичен процессу поиска в И/ИЛИ – графах.

Ответ на заданный вопрос можно найти несколькими способами в соответствии со следующими принципами:

- если *В* найден в базе знаний в виде факта, то *Отв – это «В это правда»*;
- если в базе знаний существует правило вида «если *Условие* то *В*», то для получения ответа *Отв* рассмотрите *Условие*;
- если вопрос *В* можно задавать пользователю, спросите пользователя об истинности *В*;
- если *В* имеет вид *В1 и В2*, то рассмотрите *В1*, а затем, если *В1* ложно, то положите *Отв* равным *«В это ложь»*, в противном случае рассмотрите *В2* и получите *Отв* как соответствующую комбинацию ответов на вопросы *В1* и *В2*;

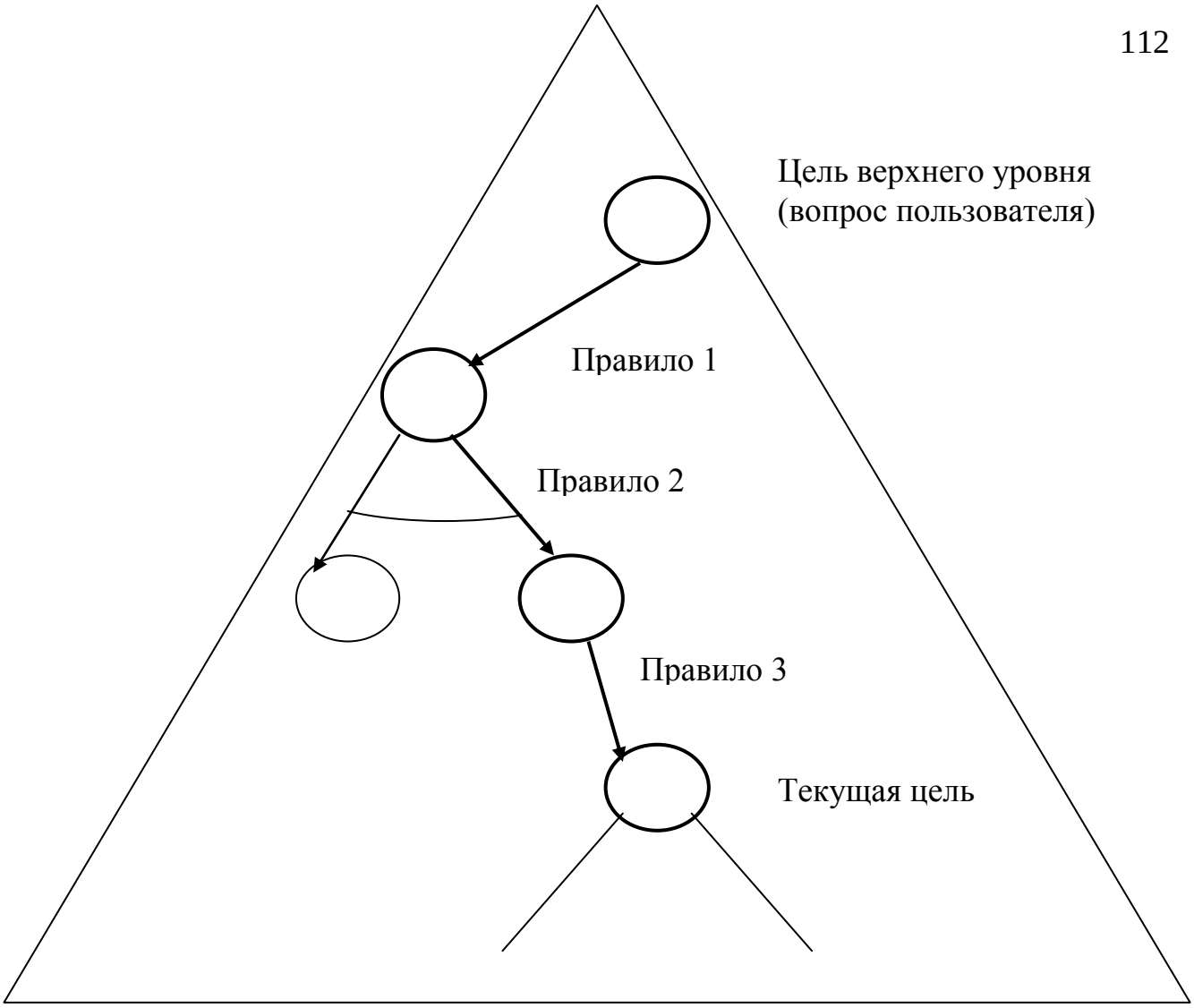
- если *B* имеет вид *B1* или *B2,* то рассмотрите *B1,* а затем, если *B1* истинно, то положите *Отв* равным *«B это правда»,* в противном случае рассмотрите *B2* и получите *Отв* как соответствующую комбинацию ответов на вопросы *B1* и *B2*.

5.7 Формирование ответа на вопрос «почему»

Объяснение экспертной системы в этом случае должно выглядеть примерно так:

потому, что
Я могу использовать ***a,***
чтобы проверить по ***правилу 1****, что* ***b****, и*
Я могу использовать ***b,***
чтобы проверить по ***правилу 2****, что* ***c****, и*
Я могу использовать ***c,***
чтобы проверить по ***правилу 3****, что* ***d****, и*
…
Я могу использовать ***y,***
чтобы проверить по ***правилу n****, что* ***z****, и*
z *– это Ваш исходный вопрос.*

Объяснение – это демонстрация того, как система намерена использовать информацию, которую она хочет получить от пользователя. Намерения системы демонстрируются в виде цепочки правил и целей, соединяющей эту информацию с исходным вопросом. Такая цепочка называется *трассой.* Трассу можно представить как цепочку правил, соединяющую в И/ИЛИ – дереве вопросов текущую цель с целью самого верхнего уровня так, как это показано на рисунке. Таким образом, для формирования ответа на вопрос «почему» нужно двигаться в пространстве поиска от текущей цели вверх вплоть до самой верхней цели. Для того, чтобы суметь это сделать, нам придется в процессе рассуждений сохранять трассу в явном виде.



5.8 Формирование ответа на вопрос «как»

Один из известных способов ответить на вопрос «как» – это представить доказательство, то есть те правила и подцели, которые использовались для достижения полученного заключения. Это доказательство имеет вид решающего И/ИЛИ – дерева. Поэтому наша машина логического вывода будет не просто отвечать на вопрос, соответствующий цели самого верхнего уровня, а будет выдавать в качестве ответа решающее И/ИЛИ – дерево, составленное из имен правил и подцелей. Затем это дерево можно отобразить на выходе системы в качестве объяснения типа *«как».* Объяснению можно придать удобную для восприятия форму, например:

«Пушок» это хищник
было выведено по правилу 3 из
«Пушок» это млекопитающее
было выведено по правилу 1 из
«Пушок» имеет шерсть
было сказано
и «Пушок» ест мясо
было сказано.

5.9 Работа с неопределенностью

Описанная выше оболочка экспертной системы может работать только с такими вопросами (утверждениями), которые либо истинны, либо ложны. Правила базы знаний - «категорические импликации», однако многие области экспертных знаний не являются категорическими. Поэтому, как данные, относящиеся к конкретной задаче, так и импликации, содержащиеся в правилах, могут быть не вполне определенными. Неопределенность можно продемонстрировать, приписывая утверждениям некоторые характеристики, отличные от *«истина»* и *«ложь»*. Характеристики могут иметь свое внешнее выражение в форме дескрипторов, таких как, например, *верно, весьма вероятно, вероятно, маловероятно, невозможно*. Другой способ представления – степень вероятности может выражаться в форме действительного числа, заключенного в некотором интервале, например между 0 и 1 или между –5 и +5. Такую характеристику называют по-разному- «коэффициент определенности», «степень доверия», «субъективная уверенность». Более естественным было бы использовать вероятности в математическом смысле слова, но попытки применить их на практике приводят к трудностям по следующим причинам:

- экспертам неудобно мыслить в терминах вероятностей. Их оценки правдоподобия не вполне соответствуют математическому определению вероятности;
- работа с вероятностями, корректная с точки зрения математики, потребовала бы каких-либо упрощающих допущений, не вполне оправданных с точки зрения практического приложения.

Поэтому, даже если выбранная мера правдоподобия лежит в интервале от 0 до 1, более правильным будет называть ее «субъективной уверенностью», подчеркивая этим, что имеется в виду оценка, данная экспертом. Вычисления над такими оценками могут отличаться от вычислений теории вероятностей, однако, они могут служить вполне адекватной моделью того, как человек оценивает достоверность своих выводов.

Для работы в условиях неопределенности было придумано множество различных механизмов, мы рассмотрим одну простую модель, которая не лишена недостатков, но была использована на практике в экспертных системах минералогической разведки и локализации неисправностей.

В данной системе достоверность событий моделируется с помощью действительных чисел, заключенных в интервале от 0 до 1. Отношения между событиями можно представить графически в виде «сети вывода», на которой события изображаются прямоугольниками, а отношения между ними стрелками. Овалами изображены комбинации событий (И, ИЛИ, НЕ). Отношения между событиями являются «мягкими импликациями».

Пусть имеются два события E и H, и пусть информация о том, что имело место событие E, оказывает влияние на нашу уверенность в том, что произошло событие H. Данному отношению можно приписать некоторую

«силу», с которой оно действует:

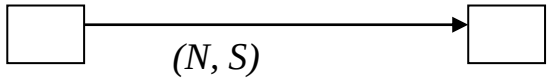
если E, то H с силой S.

В данной системе сила моделируется при помощи двух параметров:

N =<коэффициент необходимости>;

S=<коэффициент достаточности>.

В сети вывода это изображается так:



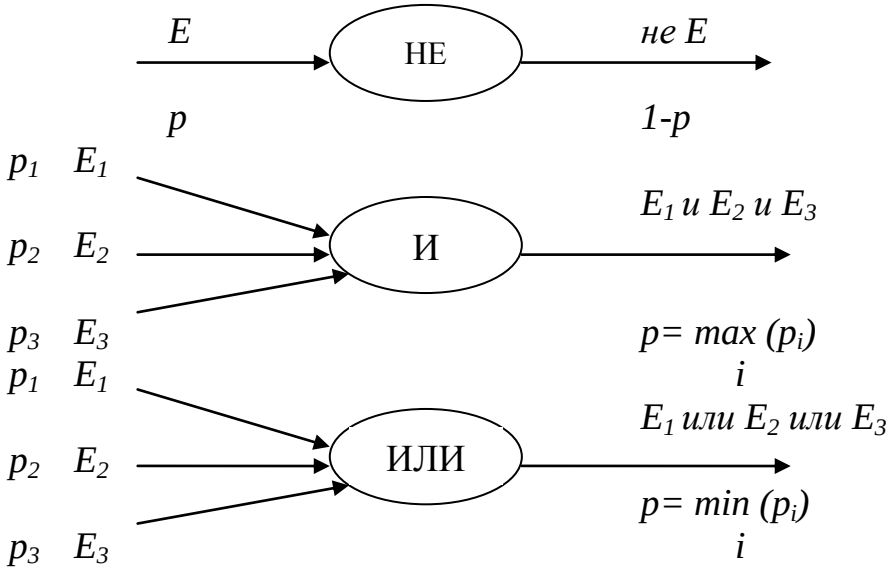
$p_0(E)$ $p_0(H)$, p_0 – *априорная вероятность;*

$p(E)$ $p(H|E)$, p – *апостериорная вероятность.*

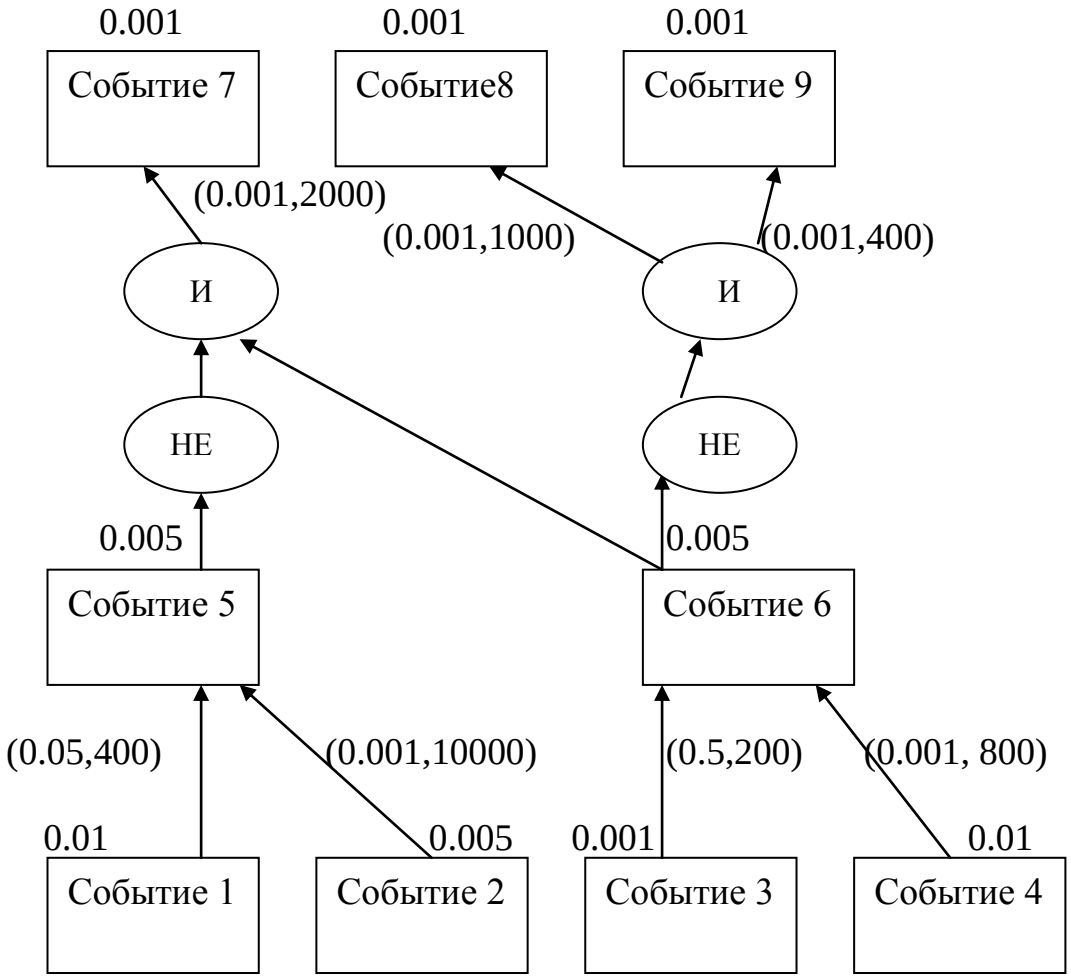
Два события, участвующие в отношении, часто называют “фактом” и “гипотезой”. Тогда, необходимо найти такой факт *E*, который мог бы подтвердить или опровергнуть гипотезу *H. S* показывает, в какой степени *достаточно факта E* для подтверждения *гипотезы H, N* – насколько *необходим* факт *E* для подтверждения гипотезы *H*. Если факт *E* имел место, то чем больше *S*, тем больше уверенности в *H*. С другой стороны, если не верно, что имел место факт *E*, то чем больше *N*, тем менее вероятно, что гипотеза *H* верна.

Для каждого события *H* сети вывода существует априорная вероятность $p_0(H)$- безусловная вероятность события *H* в состоянии, когда неизвестно ни одного положительного или отрицательного факта. Если становится известным какой-нибудь факт *E*, то вероятность *H* меняет свое значение с $p_0(H)$ на $p(H|E)$. Величина изменения зависит от «силы» стрелки, ведущей из *E в H.*

Таким образом, проверка гипотез начинается, исходя из априорных вероятностей. В дальнейшем происходит накопление информации о фактах, что находит свое отражение в изменении вероятностей событий сети. Эти изменения распространяются по сети событий в соответствии со связями между событиями. Логические комбинации отношений можно изобразить следующим образом:



На следующем рисунке представлен пример представления сети событий.



ть это избыточная информация.

Литература:

1. Адаменко А.Н., Кучуков А. Логическое программирование и Visual Prolog – Спб.: БХВ – Петербург, 2003.
2. Андрейчиков А.В., Андрейчикова О.Н.. Интеллектуальные информационные системы: Учебник. – М.: Финансы и статистика, 2006. – 424 с., ил.
3. Братко И. Алгоритмы искусственного интеллекта на языке Prolog. М.: Вильямс, 2004. – 637 с.
4. Марселлус Д. Программирование экспертных систем на Турбо Прологе: Пер. с англ. – М.: Финансы и статистика, 1994. – 256 с., ил.
5. Осовский С. Нейронные сети для обработки информации / Пер. с пол. И.Д. Рудинского. – М.: Финансы и статистика, 2002. – 344 с.: ил.
6. Солдатова О.П. Основы нейроинфоматики: учеб. пособие - Самара: Изд-во Самар. гос. аэрокосм. ун-та, 2006- 132 с. ил.
7. Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог: Пер. с англ. М.: Мир.1990.
8. Хайкин С. Нейронные сети: Полный курс: Пер. с англ. - 2-е изд. – М.: Вильямс, 2006. – 1104 с.: ил.

Учебное пособие по дисциплине «Системы искусственного интеллекта»

Содержание

Предисловие 4
1 Логическое программирование и аксиоматические системы 4
1.1 Общие положения 4
1.2 Автоматизация доказательства в логике предикатов. 6
1.2.1 История вопроса 6
1.2.2 Скулемовские стандартные формы. 7
1.2.3 Метод резолюций в исчислении высказываний. 10
1.2.4 Правило унификации в логике предикатов. 12
1.2.5 Метод резолюций в исчислении предикатов 14
2 Введение в язык логического программирования ПРОЛОГ. 16
2.1 Общие положения 16
2.2 Основы языка программирования Пролог 16
2.3 Использование дизъюнкции и отрицания. 20
2.4 Унификация в Прологе. 20
2.5 Вычисление цели. Механизм возврата. 21
2.6 Управление поиском решения. 23
2.7 Процедурность Пролога. 24
2.8 Структура программ Пролога. 25
2.9 Использование составных термов 28
2.10 Использование списков 29
2.11 Применение списков в программах 31
2.11.1 Поиск элемента в списке 32
2.11.2 Объединение двух списков 33
2.11.3 Определение длины списка 33
2.11.4 Поиск максимального и минимального элемента в списке 34
2.11.5 Сортировка списков 35
2.11.6 Компоновка данных в список 37
2.12 Повторение и рекурсия в Прологе 38
2.12.1 Механизм возврата 38
2.12.2 Метод возврата после неудачи 39
2.12.3 Метод повтора, использующий бесконечный цикл 41
2.13 Методы организации рекурсии 42
2.14 Создание динамических баз данных 45
2.15 Использование строк в Прологе. 49
2.16 Преобразование данных в Прологе 51
2.17 Представление бинарных деревьев 52
2.18 Представление графов в языке Пролог 54
2.19 Поиск пути на графе. 57
2.20 Метод “образовать и проверить” 59
3 Основные стратегии решения задач. Поиск решения в пространстве состояний 62
3.1 Понятие пространства состояния 62
3.2 Основные стратегии поиска решений в пространстве состояний 63

3.2.1 Поиск в глубину 63
3.2.2 Поиск в ширину 67
3.3 Сведение задачи к подзадачам и И/ИЛИ графы. 70
3.4 Решение игровых задач в терминах И/ИЛИ- графа 73
3.5 Минимаксный принцип поиска решений 75
Литература 79

Предисловие

Язык Пролог был создан как язык программирования для решения задач искусственного интеллекта. Языку Пролог посвящались многие книги и статьи в журналах, однако все они описывали разные версии языка, и только одна монография была посвящена языку Visual Prolog– книга А.Н. Адаменко и А.М.Кучукова «Логическое программирование и Visual Prolog» – вышла в 2003 году и в настоящее время стала библиографической редкостью.

За последние годы в стандарты многих специальностей и направлений подготовки бакалавров и магистров, связанных с информационными технологиями, введены курсы, освоение которых предполагает получение навыков логического программирования. Для студентов, изучающих логическое программирование, и предназначено данное учебное пособие.

В пособии содержится краткое описание математических основ логического программирования, введение в язык программирования Visual Prolog и изложение наиболее известных методов и алгоритмов решения интеллектуальных задач. В пособии приведено множество примеров программ на языке Visual Prolog, иллюстрирующих описываемые методы и алгоритмы.

1 Логическое программирование и аксиоматические системы

1.1 Общие положения

Теория формальных систем и, в частности, математическая логика являются формализацией человеческого мышления и представления наших знаний. Если предположить, что можно аксиоматизировать наши знания и можно построить алгоритм, позволяющий реализовать процесс вывода ответов на запрос из знаний, то в результате можно получить формальный метод для получения неформальных результатов.

Логическое программирование возникло в эру ЭВМ как естественное желание автоматизировать процесс логического вывода, поэтому оно является ветвью теории формальных систем.

Логическое программирование (в широком смысле) представляет собой семейство таких методов решения задач, в которых используются приемы логического вывода для манипулирования знаниями, представленными в декларативной форме [1]. Как писал Джордж Робинсон в 1984 году, в основе идеи логического программирования лежит описание задачи совокупностью утверждений на некотором формальном логическом языке и получение решения с помощью вывода в некоторой формальной (аксиоматической) системе. Такой аксиоматической системой являются исчисление предикатов первого порядка, поэтому в узком смысле логическое программирование понимается как использование исчисления предикатов

первого порядка в качестве основы для описания предметной области и осуществления резолюционного логического вывода.

Аксиоматической системой называется способ задания множества путем указания исходных элементов (аксиом исчисления) и правил вывода, каждое из которых описывает, как строить новые элементы из исходных элементов.

Любая аксиоматическая система должна удовлетворять следующим требованиям:

1. Непротиворечивость: невозможность вывода отрицания уже доказанного выражения (которое считается общезначимым);
2. Независимость (минимальность): система не должна содержать бесполезных аксиом и правил вывода. Некоторое выражение *независимо* от аксиоматической системы, если его нельзя вывести с помощью этой системы. В минимальной системе каждая аксиома независима от остальной системы, то есть, не выводима из других аксиом.
3. Полнота (взаимность адекватности): любая тавтология выводима из системы аксиом. В адекватной системе аксиом любая выводимая формула есть тавтология, то есть верно, что $\vdash P \rightarrow \models P$. Соответственно в *полной системе верно:* $\models P \rightarrow \vdash P$.

Исчислениями называют наиболее важные из аксиоматических логических систем – исчисление высказываний и исчисление предикатов. Исчисление высказываний и исчисление предикатов первого порядка являются *полными аксиоматическими системами.*

Под *аксиоматическим методом* [1] понимают способ построения научной теории, при котором за ее основу берется ряд основополагающих, не требующих доказательств положений этой теории, называемыми аксиомами или постулатами.

Аксиоматический метод зародился в работах древнегреческих геометров. Вплоть до начала XIX века единственным образцом применения этого метода была геометрия Евклида.

В начале XIX века Н.И.Лобачевский и Я.Больяй, независимо друг от друга, открыли новую неевклидову геометрию, заменив пятый постулат о параллельных прямых на его отрицание. Их открытие стало отправной точкой для развития аксиоматического метода, который лег в основу теории формальных систем.

Формальная теория строится как четко определенный класс выражений, формул, в котором некоторым точным способом выделяется подкласс теорем данной формальной системы. При этом формулы формальной системы непосредственно не несут в себе никакого содержательного смысла, они строятся из произвольных знаков или символов, исходя лишь из соображений удобства.

Формальная теория (система), задаётся четверкой вида $M = <T, S, A, B>$. Множество T есть множество *базовых элементов*, например слов из

некоторого словаря, или деталей из некоторого набора. Для множества *T* существует некоторый способ определения принадлежности или непринадлежности произвольного элемента к данному множеству. Процедура такой проверки может быть любой, но она должна давать ответ на вопрос, является ли *x* элементом множества *T* за конечное число шагов. Обозначим эту процедуру *P(T).*

Множество S есть множество синтаксических правил. С их помощью из элементов T образуют синтаксически правильные совокупности. Например, из слов словаря строятся синтаксически правильные фразы, а из деталей собираются конструкции. Существует некоторая процедура P(S), с помощью которой за конечное число шагов можно получить ответ на вопрос, является ли совокупность X синтаксически правильной.

Во множестве синтаксически правильных совокупностей выделяется некоторое подмножество A. Элементы A называются аксиомами. Как и для других составляющих формальной системы, должна существовать процедура P(A), с помощью которой для любой синтаксически правильной совокупности можно получить ответ на вопрос о принадлежности ее к множеству A.

Множество B есть множество правил вывода. Применяя их к элементам A, можно получать новые синтаксически правильные совокупности, к которым снова можно применять правила из B. Так формируется множество выводимых в данной формальной системе совокупностей. Если имеется процедура P(B), с помощью которой можно определить для любой синтаксически правильной совокупности, является ли она выводимой, то соответствующая формальная система называется *разрешимой*. Это показывает, что именно правила вывода являются наиболее сложной составляющей формальной системы.

Исчисление высказываний является *разрешимой формальной системой,* а исчисление предикатов первого порядка – *неразрешимой формальной системой.*

С накоплением опыта построения формальных теорий и попытками аксиоматизации арифметики, предпринятыми Дж. Пеано, возникла *теория доказательств*. Теория доказательств – это раздел современной математической логики и предшественница логического программирования.

1.2 Автоматизация доказательства в логике предикатов.

1.2.1 История вопроса

Поиск общей разрешающей процедуры для проверки общезначимости формул был начат Лейбницем в XVII веке. Затем исследования продолжились в XX веке, а в 1936 году Черч и Тьюринг независимо друг от друга доказали, что не существует никакой общей разрешающей процедуры, никакого алгоритма, проверяющего общезначимость формул в логике предикатов первого порядка.

Тем не менее, существуют алгоритмы поиска доказательства, которые могут подтвердить, что формула общезначима, если она на самом деле общезначима (для необщезначимых формул эти алгоритмы, вообще говоря, не заканчивают свою работу).

Очень важный подход к автоматическому доказательству теорем был дан Эрбраном в 1930 году. По определению общезначимая формула есть формула, которая истинна при всех интерпретациях. Эрбран разработал алгоритм нахождения интерпретации, которая опровергает данную формулу. Однако, если данная формула действительно общезначима, то никакой интерпретации не существует и алгоритм заканчивает работу за конечное число шагов. Метод Эрбрана служит основой для большинства современных автоматических алгоритмов поиска доказательства.

Гилмор в 1959 году одним из первых реализовал процедуру Эрбрана. Его программа была предназначена для обнаружения противоречивости отрицания данной формулы, так как формула общезначима тогда и только тогда, когда ее отрицание противоречиво. Однако программа Гилмора оказалась неэффективной и в 1960 году метод Гилмора был улучшен Девисом и Патнемом. Но их улучшение оказалось недостаточным, так как многие общезначимые формулы логики предикатов все еще не могли быть доказаны на ЭВМ за разумное время.

Главный шаг вперед сделал Робинсон в 1965 году, который ввел так называемый метод резолюций, который оказался много эффективней, чем любая описанная ранее процедура. После введения метода резолюций был предложен ряд стратегий для увеличения его эффективности. Такими стратегиями являются *семантическая резолюция, лок-резолюция, линейная резолюция, стратегия предпочтения единичных и стратегия поддержки.*

1.2.2 Скулемовские стандартные формы.

Процедуры доказательства по Эрбрану или методу резолюций на самом деле являются процедурами опровержения, то есть вместо доказательства общезначимости формулы доказывается, что ее отрицание противоречиво. Кроме того, эти процедуры опровержения применяются к некоторой стандартной форме, которая была введена Девисом и Патнемом. По существу они использовали следующие идеи:

1. Формула логики предикатов может быть сведена к ПНФ, в которой матрица не содержит никаких кванторов, а префикс есть последовательность кванторов.
2. Поскольку матрица не содержит кванторов, она может быть сведена к конъюнктивной нормальной форме.
3. Сохраняя противоречивость формулы, в ней можно исключить кванторы существования путем использования скулемовских функций.

Алгоритм преобразования формулы в ПНФ известен. При помощи законов эквивалентных преобразований логики высказываний можно свести матрицу к КНФ.

Алгоритм преобразования формул в ДНФ и КНФ.

Шаг 1. Используем законы эквивалентных преобразований исчисления высказываний для того, чтобы исключить логические связки импликации и эквивалентности.

Шаг 2. Многократно используем закон двойного отрицания, и законы де Моргана, чтобы внести знак отрицания внутрь формулы.

Шаг 3. Несколько раз используем дистрибутивные законы и другие законы, чтобы получить НФ.

Алгоритм преобразования формулы $(K_1x_1)\ldots(K_nx_n)\ (M)$, где каждое $(K_ix_i),\ i = 1,\ldots,n,$ есть или $(\forall x_i)$ или $(\exists x_i)$, и M есть *КНФ* в скулемовскую нормальную форму (СНФ) приведен ниже.

Алгоритм преобразования ПНФ в ССФ.

Шаг 1. Представим формулу в ПНФ $(K_1x_1)\ldots(K_nx_n)\ (M)$, где M есть *КНФ*. Пусть K_r есть квантор существования в префиксе $(K_1x_1)\ldots(K_nx_n)$, $1<=r<=n$.

Шаг 2. Если никакой квантор всеобщности не стоит левее K_r – выберем новую константу c, отличную от других констант, входящих в M, заменим все x_r в M на c и вычеркнем K_rx_r из префикса. Если $K_1,\ldots K_i$ – список всех кванторов всеобщности, встречающихся в M левее K_r, $1<\ i<r$, выберем новый i –местный функциональный символ f, отличный от других функциональных символов, заменим все x_r в M на $f(x_1, x_2,\ldots x_i)$ и вычеркнем K_rx_r из префикса.

Шаг 3. Применим шаг 2 для всех кванторов существования в префиксе. Последняя из полученных формул есть *скулемовская стандартная форма* формулы. Константы и функции, используемые для замены переменных квантора существования, называются *скулемовскими* функциями.

Пример 1. Получить ССФ для формулы $(\exists x)(\forall y)(\forall z)(\exists u)(\forall v)(\exists w)\ (P(x, y, z, u, v, w)$.

В этой формуле левее $(\exists x)$ *нет никаких кванторов всеобщности, левее* $(\exists u)$ *стоят* $(\forall y)$ *и* $(\forall z)$, *а левее* $(\exists w)$ *стоят* $(\forall y)$, $(\forall z)$ *и* $(\forall v)$. *Следовательно, мы заменим переменную x на константу a, переменную u - на двухместную f(y, z), переменную w - на трехместную функцию g(y, z, v). Таким образом, мы получаем следующую стандартную форму написанной выше формулы:*

$(\forall y)(\forall z)(\forall v)(P(a, y, z, f(y, z), g(y, z, v))$.

Определение 1: *Дизъюнктом называется дизъюнкция литералов. Дизъюнкт, содержащий r литералов, называется r- литеральным дизъюнктом. Однолитеральный дизъюнкт называется единичным дизъюнктом. Если дизъюнкт не содержит никаких литералов, то он*

называется пустым дизъюнктом- □ *. Так как пустой дизъюнкт не содержит литералов, которые могли бы быть истинными при любых интерпретациях, то пустой дизъюнкт всегда ложен.*

Определение 2: *Множество дизъюнктов S есть конъюнкция всех дизъюнктов из S , где каждая переменная в S считается управляемой квантором всеобщности.*

Вследствие последнего определения, ССФ может быть представлена множеством дизъюнктов.

Пример 2. Получить скулемовскую стандартную форму формулы

(∀x) (P(x) ∧ (∀y)(¬ (∀z)Q(x, y)→ (∃u)R(u, x, y))) и представить её в виде множества дизъюнктов.

1. Исключим связки импликации:

(∀x) (P(x) ∧ (∀y)(¬ ¬(∀z)Q(x, y)∨ (∃u)R(u, x, y))).

2. Удалим бесполезные кванторы:

(∀x) (P(x) ∧ (∀y)(¬ ¬Q(x, y)∨ (∃u)R(u, x, y))).

3. Применим правило двойного отрицания:

(∀x) (P(x) ∧ (∀y)(Q(x, y)∨ (∃u)R(u, x, y))).

4. Переместим кванторы в начало формулы:

(∀x)(∀y)(∃u) (P(x) ∧ (Q(x, y)∨ R(u, x, y))),

Получим ПНФ.

(∀x)(∀y)(∃u)(P(x) ∧ (Q(x, y)∨ R(u, x, y))), у которой матрица находится в КНФ.

Избавимся от кванторов существования в префиксе:

Так как перед (∃u) есть (∀x),(∀y) то переменная и заменяется двухместной функцией f(x, y). Таким образом, мы получаем следующую стандартную форму:

(∀x)(∀y)(P(x) ∧ (Q(x, y) ∨ R(f(x, y), x, y))).

Получим ССФ.

Отбросим кванторы всеобщности и заменим конъюнкцию на перечисление:

{P(x), (Q(x, y) ∨ R(f(x, y), x, y))}.

Получим множество из двух дизъюнктов.

Пример 3. Получить скулемовскую стандартную форму формулы

(∀x)(∃y)(∃z)((¬ P(x, y) ∧ Q(x, z)) ∨ R(x, y, z) ∧ (¬ Q(x, z) ∧ P(x, y))).

Сначала сведем матрицу к КНФ:

((¬ P(x, y) ∨ R(x, y, z))∧ (Q(x, z) ∨ R(x, y, z)) ∧ (¬ Q(x, z) ∧ P(x, y))).

Затем избавимся от кванторов существования в префиксе:

Так как перед (∃y)(∃z) есть (∀x), то переменные y, z заменяются соответственно одноместными функциями f(x), g(x). Таким образом, мы получаем следующую стандартную форму:

(∀x)(((¬ P(x, f(x)) ∨ R(x, f(x), g(x)))∧ (Q(x, g(x)) ∨ R(x, f(x), g(x)) ∧ (¬Q(x, g(x)) ∧ P(x, f(x))))).

Представим полученную ССФ в виде множества дизъюнктов:

{¬P(x, f(x))∨ R(x, f(x), g(x)), Q(x, g(x)) ∨R(x, f(x), g(x)), ¬Q(x, g(x)) ∧ P(x, f(x))}.

Теорема 1. Пусть S – множество дизъюнктов, которые представляют ССФ формулы F. Тогда F противоречива в том и только в том случае, когда S противоречиво.

Теорема 2. Пусть даны формулы F_1, F_2,…, F_n и формула G. G есть логическое следствие F_1, F_2,…, F_n тогда и только тогда, когда формула $((F_1 \wedge F_2 \wedge \ldots \wedge F_n) \rightarrow G)$ общезначима.

Теорема 3. Пусть даны формулы F_1, F_2,…, F_n и формула G. G есть логическое следствие F_1, F_2,…, F_n тогда и только тогда, когда формула $(F_1 \wedge F_2 \wedge \ldots \wedge F_n \wedge \neg G)$ противоречива.

Замечание.

Для того чтобы доказать, что данная формула является тавтологией, достаточно доказать, что ее отрицание является противоречием:

$\neg((F_1 \wedge F_2 \wedge \ldots \wedge F_n) \rightarrow G) =$

$\neg(\neg(F_1 \wedge F_2 \wedge \ldots \wedge F_n) \vee G) =$

$(F_1 \wedge F_2 \wedge \ldots \wedge F_n) \wedge \neg G).$

На основании теорем 1 и 3 можно сделать вывод, что формула G является логическим следствием формулы F тогда, когда противоречива конъюнкция множества S и формулы ¬G, то есть противоречива формула $S_1 \wedge S_2 \wedge \ldots S_n \wedge \neg G$. Таким образом, если в множество S добавить негативный литерал ¬G и доказать, что полученное множество противоречиво, то тем самым можно доказать выводимость G из множества S.

1.2.3 Метод резолюций в исчислении высказываний.

Основная идея метода резолюций состоит в том, чтобы проверить, содержит ли множество дизъюнктов пустой дизъюнкт. Если множество содержит пустой дизъюнкт, то оно противоречиво (невыполнимо). Если множество не содержит пустой дизъюнкт, то проверяется следующий факт: может ли пустой дизъюнкт быть получен из данного множества. Множество содержит пустой дизъюнкт, тогда и только тогда, когда оно пустое. Если множество можно свести к пустому, то тем самым можно доказать его противоречивость. В этом и состоит метод резолюций, который часто рассматривают как специальное правило вывода, используемое для порождения новых дизъюнктов из данного множества.

Определение 3:Если A атом, то литералы A и ¬A контрарны друг другу, и множество { A, ¬A } называется контрарной парой.

Отметим, что дизъюнкт есть тавтология, если он содержит контрарную пару.

Определение 4: Правило резолюций состоит в следующем:

Для любых двух дизъюнктов C_1 и C_2, если существует литерал L_1 в C_1, который контрарен литералу L_2 в C_2, то вычеркнув L_1 и L_2 из C_1 и C_2 соответственно и построив дизъюнкцию оставшихся дизъюнктов, получим резолюцию (резольвенту) C_1 и C_2.

Пример 4: рассмотрим следующие дизъюнкты:

C_1: $P \vee R$,

C_2: $\neg P \vee Q$.

Дизъюнкт C_1 имеет литерал P, который контрарен литералу $\neg P$ в C_2. Следовательно, вычеркивая P и $\neg P$ из C_1 и C_2 соответственно, построим дизъюнкцию оставшихся дизъюнктов R и Q и получим резольвенту $R \vee Q$.

Важным свойством резольвенты является то, что любая резольвента двух дизъюнктов C_1 и C_2 есть логическое следствие C_1 и C_2. Это устанавливается в следующей теореме.

Теорема 4. Пусть даны два дизъюнкта C_1 и C_2. Тогда резольвента C дизъюнктов C_1 и C_2 есть логическое следствие C_1 и C_2.

Если есть два единичных дизъюнкта, то их резольвента, если она существует, есть пустой дизъюнкт □. Более существенно, что для невыполнимого множества дизъюнктов многократным применением правила резолюций можно породить □.

Определение 5: Пусть S – множество дизъюнктов. Резолютивный вывод C из S есть такая конечная последовательность C_1, C_2,…, C_k дизъюнктов, что каждый C_i или принадлежит S или является резольвентой дизъюнктов, предшествующих C_i, и C_k=C. Вывод □ из S называется опровержением (или доказательством невыполнимости) S.

Пример 5. Рассмотрим множество S:

1. $\neg P \vee Q$,

2. $\neg Q$,

3. P.

Из 1 и 2 получим резольвенту

4. $\neg P$.

Из 4 и 3 получим резольвенту

5. □.

Так как □ получается из S применениями правила резолюций, то согласно теореме 4 □ есть логическое следствие S, следовательно, S невыполнимо.

Метод резолюций является наиболее эффективным в случае применения его к множеству Хорновских дизъюнктов.

Определение 6: Фразой называется дизъюнкт, у которого негативные литералы размещаются после позитивных литералов в конце дизъюнкта.

Пример 6: $P_1 \vee P_2 \vee \ldots P_n \vee \neg N_1 \vee \neg N_2 \ldots \vee \neg N_m$

Определение 28: Фраза Хорна это фраза, содержащая только один позитивный литерал.

Пример 7: преобразовать фразу Хорна в обратную импликацию.

$P \vee \neg N_1 \vee \neg N_2 \ldots \vee \neg N_m$

$\neg N_1 \vee \neg N_2 \ldots \vee \neg N_m = \neg (N_1 \wedge N_2 \wedge \ldots \wedge N_m)$

$P \leftarrow (N_1 \wedge N_2 \wedge \ldots \wedge N_m)$

$P \leftarrow N_1, N_2, \ldots N_m$

При представлении дизъюнктов фразами Хорна негативные литералы соответствуют гипотезам, а позитивный литерал представляет заключение. Единичный позитивный дизъюнкт представляет некоторый факт, то есть заключение, не зависящее ни от каких гипотез. Часто задача состоит в том, что надо проверить некоторую формулу, называемую целью, логически выведенную из множества правил и фактов. Резолюция является методом доказательства от противного: исходя из фактов, правил и отрицания цели, приходим к противоречию (пустому дизъюнкту).

1.2.4 Правило унификации в логике предикатов.

Правило резолюций предполагает нахождение в дизъюнкте литерала, контрарного литералу в другом дизъюнкте. Для дизъюнктов логики высказываний это очень просто. Для дизъюнктов логики предикатов процесс усложняется, так как дизъюнкты могут содержать функции, переменные и константы.

Пример 8. Рассмотрим дизъюнкты:

C_1: $P(y) \vee Q(y)$,

C_2: $\neg P(f(x)) \vee R(x)$.

Не существует никакого литерала в C_1, контрарного какому-либо литералу в C_2. Однако, если подставить f(a) вместо y в C_1 и a вместо x в C_2, то исходные дизъюнкты примут вид:

C_1': $P(f(a)) \vee Q(f(a))$,

C_2': $\neg P(f(a)) \vee R(a)$.

Так как P(f(a)) контрарен $\neg P(f(a))$, то можно получить резольвенту

C_3': $Q(f(a)) \vee R(a)$.

В общем случае, подставив f(x) вместо y в C_1, получим

C_1'': $P(f(x)) \vee Q(f(x))$.

Литерал $P(f(x))$ в C_1'' контрарен литералу $\neg P(f(x))$ в C_2. Следовательно, можно получить резольвенту

C_3: $Q(f(x)) \vee R(x)$.

Таким образом, если подставлять подходящие термы вместо переменных в исходные дизъюнкты, можно порождать новые дизъюнкты. Отметим, что дизъюнкт C_3 из примера 8 является наиболее общим дизъюнктом в том смысле, что все другие дизъюнкты, порожденные правилом резолюции будут частным случаем данного дизъюнкта.

Определение 7: Подстановка θ– это конечное множество вида $\{t_1/v_1, \ldots, t_n/v_n\}$, где каждая v_i – переменная, каждый t_i – терм, отличный от v_i, все v_i различны.

Определение 8: Подстановка θ называется унификатором для множества $\{E_1,\ldots, E_k\}$ тогда и только тогда, когда $E_1\theta=E_2\theta=\ldots E_k\theta$. Множество $\{E_1,\ldots, E_k\}$ унифицируемо, если для него существует унификатор.

Прежде чем применить правило резолюции в исчислении предикатов переменные в литералах необходимо унифицировать.

Унификация производится при следующих условиях:

1. Если термы константы, то они унифицируемы тогда и только тогда, когда они совпадают.
2. Если в первом дизъюнкте терм переменная, а во втором константа, то они унифицируемы, при этом вместо переменной подставляется константа.
3. Если терм в первом дизъюнкте переменная и во втором дизъюнкте терм тоже переменная, то они унифицируемы.
4. Если в первом дизъюнкте терм переменная, а во втором - употребление функции, то они унифицируемы, при этом вместо переменной подставляется употребление функции.
5. Унифицируются между собой термы, стоящие на одинаковых местах в одинаковых предикатах.

Пример 9. Рассмотрим дизъюнкты:

1. Q(a, b, c) и Q(a, d, l). Дизъюнкты не унифицируемы.

2. Q(a, b, c) и Q(x, y, z). Дизъюнкты унифицируемы. Унификатор - Q(a, b, c).

Определение 9: Унификатор σ для множества $\{E_1,\ldots, E_k\}$ будет наиболее общим унификатором тогда и только тогда, когда для каждого унификатора θ для этого множества существует такая подстановка λ, что $\theta=\sigma\circ\lambda$, то есть θ является композицией подстановок σ и λ.

Определение 10: Композицией подстановок σ и λ есть функция $\sigma\circ\lambda$, определяемая следующим образом $(\sigma\circ\lambda)[t]=\sigma[\lambda[t]]$, где t – терм, σ и λ - подстановки, а $\lambda[t]$ – терм, который получается из t путем применения к нему подстановки λ.

Определение 11: Множество рассогласований непустого множества дизъюнктов $\{E_1,\ldots, E_k\}$ получается путем выявления первой (слева) позиции, на которой не для всех дизъюнктов из E стоит один и тот же символ, и выписывания из каждого дизъюнкта терма, который начинается с символа, занимающего данную позицию. Множество термов и есть множество рассогласований в E.

Пример 10. Рассмотрим дизъюнкты:

{P(x, f(y, z)), P(x, a), P(x, g(h(k(x))))}.

Множество рассогласований состоит из термов, которые начинаются с пятой позиции и представляет собой множество {f(x, y), a, g(h(k(x)))}.

Алгоритм унификации для нахождения наиболее общего унификатора.

Пусть E – множество дизъюнктов, D – множество рассогласований, k – номер итерации, σ_k наиболее общий унификатор на k-ой итерации.

Шаг 1. Присвоим k=0, σ_k=e (пустой унификатор), E_k=E.

Шаг 2. Если для E_k не существует множества рассогласований D_k, то остановка: σ_k – наиболее общий унификатор для E. Иначе найдем множество рассогласований D_k.

Шаг 3. Если существуют такие элементы v_k и t_k в D_k, что v_k переменная, не входящая в терм t_k, то перейдем к шагу 4. В противном случае остановка: E не унифицируемо.

Шаг 4. Пусть $\sigma_{k+1}=\sigma_k \{ t_k / v_k\}$, заменим во всех дизъюнктах E_k t_k на v_k.

Шаг5. K=k+1. Перейти к шагу 2.

Пример 11. Рассмотрим дизъюнкты:

E={P(f(a), g(x)), P(y, y)}.

1. *E_0=E, k=0, σ_0=e.*
2. *D_0={f(a),y}, v_0=y, t_0=f(a).*
3. *σ_1=={f(a)/y}, E_1={P(f(a), g(x)), P(f(a), f(a))}.*
4. *D_1={g(x),f(a)}.*
5. *Нет переменной в множестве рассогласований D_1.*

Следовательно, алгоритм унификации завершается, множество E – не унифицируемо.

1.2.5 Метод резолюций в исчислении предикатов

С помощью унификации можно распространить правило резолюций на исчисление предикатов. При унификации возникает одна трудность: если один из термов есть переменная x, а другой терм содержит x, но не сводится к x, унификация невозможна. Проблема решается путем переименования переменных таким образом, чтобы унифицируемые дизъюнкты не содержали одинаковых переменных.

Определение 12: Если два или более литерала (с одиниковым знаком) дизъюнкта C имеют наиболее общий унификатор σ, то $C\sigma$ - называется склейкой C.

Пример 12. Рассмотрим дизъюнкты:

Пусть $C= P(x) \vee P(f(y)) \vee \neg Q(x)$.

Тогда 1 и 2 литералы имеют наиболее общий унификатор σ = {f(y)/x}. Следовательно, $C\sigma=P(f(y)) \vee \neg Q(f(y))$ есть склейка C.

Определение 13: Пусть C_1 и C_2 – два дизъюнкта, которые не имеют никаких общих переменных. Пусть L_1 и L_2 - два литерала в C_1 и C_2 соответственно. Если L_1 и $\neg L_2$ имеют наиболее общий унификатор σ, то дизъюнкт $(C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma)$ называется резольвентой C_1 и C_2.

Пример 13. Пусть $C_1= P(x) \vee Q(x)$ и $C_2= \neg P(a) \vee R(x)$. Так как x входит в C_1 и C_2, то мы заменим переменную в C_2 и пусть $C_2= \neg P(a) \vee R(y)$. Выбираем $L_1= P(x)$ и $L_2=\neg P(a)$. L_1 и L_2 имеют наиболее общий унификатор σ={a/x}. Следовательно, $Q(a) \vee R(y)$ – резольвента C_1 и C_2.

Пример 14. Доказать, что формула R(a, z) выводима из множества дизъюнктов $S=\{\neg P(x, f(x)) \vee R(x, f(x), g(x)), Q(x, g(x)) \vee R(x, f(x), g(x)), \neg Q(x, g(x)) \wedge P(x, f(x))\}$.

Пусть $C_1=\neg P(x, f(x)) \vee R(x, f(x)$, $C_2= Q(x, g(x)) \vee R(x, f(x)$, $C_3=\neg Q(x, g(x)) \vee P(x, f(x))$. *Добавим к множеству S единичный дизъюнкт* $C_4=\neg R(a, z)$.

Так как в дизъюнктах C_1 , C_2 , C_3 *есть одинаковая переменная x, то заменим её в* C_2 *на y, а в* C_3 *на u. Таким образом, решение исходной задачи сводится к доказательству противоречивости следующего множества дизъюнктов:*

$C_1=\neg P(x, f(x)) \vee R(x, f(x))$;

$C_2= Q(y, g(y)) \vee R(y, f(y))$;

$C_3=\neg Q(u, g(u)) \vee P(u, f(u))$;

$C_4=\neg R(a, z)$.

Найдём резольвенту C_5 *дизъюнктов* C_1 *и* C_3 *и добавим её к множеству дизъюнктов, для чего произведём унификацию переменных x и u, таким образом, что u заменим на x, и получим* $C_5= R(x, f(x)) \vee \neg Q(x, g(x))$.

Найдём резольвенту C_6 *дизъюнктов* C_2 *и* C_5 *и добавим её к множеству дизъюнктов, для чего произведём унификацию переменных y и x, таким образом, что y заменим на x, и получим* $C_6= R(x, f(x)) \vee R(x, f(x))= R(x, f(x))$.

Найдём резольвенту C_7 *дизъюнктов* C_2 *и* C_6 *и добавим её к множеству дизъюнктов, для чего произведём замену переменной x на константу a, а переменную z заменим на функцию f(a), таким образом получим* $C_7= \square$, *то есть пустой дизъюнкт.*

Алгоритм метода резолюций.

Шаг 1. Если в S есть пустой дизъюнкт, то множество невыполнимо, иначе перейти к шагу 2.

Шаг 2. Найти в исходном множестве S такие дизъюнкты или склейки дизъюнктов C_1 *и* C_2, *которые содержат унифицируемые литералы* $L_1 \in C_1$ *и* $L_2 \in C_2$. *Если таких дизъюнктов нет, то исходное множество выполнимо, иначе перейти к шагу 3.*

Шаг 3. Вычислить резольвенту C_1 *и* C_2 *и добавить ее в множество S. Перейти к шагу 1.*

2 Введение в язык логического программирования ПРОЛОГ.

2.1 Общие положения

Язык Пролог объединяет два подхода: логический и процедурный.

По мнению Дж. Робинсона, в основе идеи логического программирования лежит принцип описания задачи при помощи совокупности утверждений на некотором формальном логическом языке и получение решения при помощи вывода в некоторой формальной системе. Основой языка Пролог является логика предикатов первого порядка.

Программа на Прологе включает в себя постановку задачи в виде множества фраз Хорна (раздел clauses) и описание цели (раздел goal), - формулировку теоремы, которую нужно доказать, исходя из множества правил и фактов, содержащихся в этой постановке.

Процесс поиска доказательства основан на методе линейной резолюции (дизъюнкты подбираются в порядке их следования в тексте программы).

Проиллюстрируем принцип логического программирования на простом примере: запишем известный метод вычисления наибольшего общего делителя двух натуральных чисел – алгоритм Евклида в виде Хорновских дизъюнктов. При этом примем новую форму записи фразы Хорна, например $S \vee \neg P \vee \neg Q \vee \neg R$ будем записывать как $S \leftarrow P \wedge Q \wedge R$. Тогда алгоритм Евклида можно записать в виде трех фраз Хорна:

1. $NOD\ (x, x, x) \leftarrow$.
2. $NOD\ (x, y, z) \leftarrow B\ (x, y) \wedge NOD\ (f\ (x, y), y, z)$.
3. $NOD\ (x, y, z) \leftarrow B\ (y, x) \wedge NOD\ (x, f\ (y, x), z)$.

Предикат *NOD* – определяет наибольший общий делитель *z* для натуральных чисел *x* и *y*, предикат *B* – определяет отношение «больше», функция *f* – определяет операцию вычитания. Если мы заменим предикат *B* и функцию *f* обычными символами, то фразы примут вид:

1. *NOD (x, x, x): -.*
2. *NOD (x, y, z): - x>y, u=(x- y), NOD (u, y, z).*
3. *NOD (x, y, z): -y>x, u=(y- x), NOD (x, u, z).*

Для вычисления наибольшего общего делителя двух натуральных чисел, например 4 и 6, добавим к описанию алгоритма четвертый дизъюнкт:

4. *□: - NOD (4, 6, z).*

Последний дизъюнкт – это цель, которую мы будем пытаться вывести из первых трех дизъюнктов.

2.2 Основы языка программирования Пролог

Программа на языке Пролог не является последовательностью действий – она представляет собой набор фактов и правил, обеспечивающих

получение логических заключений из данных фактов. Поэтому Пролог считается декларативным языком программирования.

Пролог базируется на *фразах (предложениях) Хорна*, являющихся подмножеством формальной системы, называемой *логикой предикатов [2]*.

Пролог использует упрощенную версию синтаксиса логики предикатов, он прост для понимания и очень близок к естественному языку.

Одной из важнейших особенностей Пролога является то, что он ищет не только ответ на поставленный вопрос, но и все возможные альтернативные решения. Вместо обычной работы программы на процедурном языке от начала и до конца, Пролог может возвращаться назад и просматривать все остальные пути при решении всех частей задачи.

Программист на Прологе описывает объекты и отношения, а также правила, при которых эти отношения являются истинными.

Объекты рассуждения в Прологе называются *термами* – синтаксическими объектами одной из следующих категорий:

- константы,
- переменные,
- функции (составные термы или структуры), состоящие из имени функции и списка аргументов-термов, имена функций начинаются со строчной буквы.

Константа в Прологе служит для обозначения имен собственных и начинается *со строчной буквы*.

Переменная в Прологе служит для обозначения объекта рассуждения, на который нельзя сослаться *по имени*.

Переменные в Прологе инициализируются при сопоставлении с константами в фактах и правилах.

До инициализации переменная свободна, после присвоения ей значения она становится связанной. Переменная остается связанной только то время, которое необходимо для получения решения по запросу, затем Пролог освобождает ее и ищет другое решение.

Переменные в Прологе предназначены для установления соответствия между термами предикатов, действующих в пределах одной фразы (предложения), а не местом памяти для хранения данных. Переменная начинается с прописной буквы или знаков подчеркивания.

В Прологе программист свободен в выборе имен констант, переменных, функций и предикатов. Исключения составляют резервированные имена и числовые константы. Переменные от констант отличаются первой буквой имени: у констант она строчная, у переменных – заглавная буква или символ подчеркивания.

Область действия имени представляет собой часть программы, где это имя имеет один и тот же смысл:

- для переменной областью действия является предложение (факт, правило или цель), содержащее данную переменную;

- для остальных имен (констант, функций или предикатов) – вся программа.

Специальным знаком «_» обозначается анонимная переменная, которая используется тогда, когда конкретное значение переменной не существенно для данного предложения. Анонимные переменные не отличаются от обычных при поиске соответствий, но не принимают значений и не появляются в ответах. *Различные вхождения знака подчеркивания означают различные анонимные переменные.*

Отношения между объектами в Прологе называются фактами. Факт соответствует фразе Хорна, состоящей из одного положительного литерала.

Факт – это простейшая разновидность предложения Пролога.

Любой факт имеет соответствующее значение истинности и определяет отношение между термами.

Факт является простым предикатом, который записывается в виде функционального терма, состоящего из имени отношения и объектов, заключенных в круглые скобки, например:

мать(мария, анна).

отец(иван, анна).

Точка, стоящая после предиката, указывает на то, что рассматриваемое выражение является фактом.

Вторым типом предложений Пролога является вопрос или *цель. Цель* – это средство формулировки задачи, которую должна решать программа. Простой вопрос (цель) синтаксически является разновидностью факта, например:

Цель: мать (мария, юлия).

В данном случае программе задан вопрос, является ли мария матерью юлии. Если необходимо задать вопрос, кто является матерью юлии, то цель будет иметь следующий вид:

Цель: мать(X, юлия).

Сложные цели представляют собой конъюнкцию простых целей и имеют следующий вид:

Цель: Q_1, Q_2,…,Q_n, где запятая обозначает операцию конъюнкции, а Q_1, Q_2,…,Q_n – подцели главной цели.

Конъюнкция в Прологе истинна только при истинности всех компонент, однако, в отличие от логики, в Прологе учитывается *порядок оценки истинности компонент* (слева направо).

Пример 15.

Пусть задана семейная БД при помощи перечисления родительских отношений в виде списка фактов:

мать(мария, анна).

мать(мария, юлия).

мать(анна, петр).

отец(иван, анна).

отец(иван, юлия).

Тогда вопрос, является ли иван дедом петра, можно задать в виде следующей цели:

Цель: отец(иван, X), мать(X, петр).

На самом деле БД Пролога включает не только факты, но и правила. Факты и правила представляют собой не множество, а список. Для получения ответа БД просматривается по порядку, то есть в порядке следования фактов и предикатов в тексте программы.

Цель достигнута, если в БД удалось найти факт или правило, который (которое) удовлетворяет предикату цели, то есть превращает его в истинное высказывание. В нашем примере первую подцель удовлетворяют факты *отец(иван, анна). и отец(иван, юлия).* Вторую подцель удовлетворяет факт *мать(анна, петр).* Следовательно, главная цель удовлетворена, переменная *X* связывается с константой *анна*.

Третьим типом предложения является *правило*. Правило позволяет вывести один факт из других фактов. Иными словами, правило – это заключение, для которого известно, что оно истинно, если одно или несколько других найденных заключений или фактов являются истинными.

Правила – это предложения вида

H: - P_1, P_2,..., P_n.

Символ «: -» читается как «если», предикат H называется заключением, а последовательность предикатов P_1, P_2,..., P_n называется посылками. Приведенное правило является аналогом хорновского дизъюнкта $H \vee \neg P_1 \vee \neg P_2, \ldots, \vee \neg P_n$. Заключение истинно, если истинны все посылки. В посылках переменные связаны квантором существования, а в заключении - квантором всеобщности.

Пример 16.

Добавим в БД примера 15 правила, задающие отношение «дед»:

мать(мария, анна).

мать(мария, юлия).

мать(анна, петр).

отец(иван, анна).

отец(иван, юлия).

дед (X, Y): - отец(X, Z), мать(Z, Y).

дед (X, Y): - отец(X, Z), отец(Z, Y).

Тогда вопрос, является ли иван дедом петра, можно задать в виде следующей цели:

Цель: дед(иван, петр).

Правила - самые общие предложения Пролога, факт является частным случаем правила без правой части, а цель – правило без левой части.

Все предложения для одного предиката связаны между собой отношением «или».

Очень часто правила в Прологе являются рекурсивными. Например, для нашей семейной БД предикат «предок» определяется рекурсивно:

предок(x, y): - мать(x, y).

предок(x, y): - отец(x, y).
предок(x, y): - мать(x, z), предок(z, y).
предок(x, y): - отец (x, z), предок(z, y).

Рекурсивное определение предиката обязательно должно содержать нерекурсивную часть, иначе оно будет логически некорректным и программа зациклится. Чтобы избежать зацикливания, следует также позаботиться о порядке выполнения предложений, поэтому практически полезно, а порой и необходимо придерживаться принципа: *«сначала нерекурсивные выражения»*.

Программа на Прологе - это конечное множество предложений.

2.3 Использование дизъюнкции и отрицания.

Чистый Пролог разрешает применять в правилах и целях только конъюнкцию, однако, язык, используемый на практике, допускает применение дизъюнкции и отрицания в телах правил и целях. Для достижения цели, содержащей дизъюнкцию, Пролог–система сначала пытается удовлетворить левую часть дизъюнкции, а если это не удается, то переходит к поиску решения для правой части дизъюнкции. Аналогичные действия производятся при выполнении тела правил, содержащих дизъюнкцию. Для обозначения дизъюнкции используется символ « ; ».

В Прологе отрицание имеет имя *«not»* и для представления отрицания какого-либо предиката *P* используется запись *not(P). Цель not(P) достижима тогда и только тогда, когда не удовлетворяется предикат (цель) P.* При этом переменным значения не присваиваются. В самом деле, если достигается *P,* то не достигается *not(P),* значит надо стереть все присваивания, приводящие к данному результату. Наоборот, если *P* не достигается, то переменные не принимают никаких значений.

2.4 Унификация в Прологе.

Общий принцип выполнения программ на Прологе прост: производится поиск ответа на вопросы, задаваемые БД, состоящей из фактов и правил, то есть проверяется соответствие предикатов вопроса предложениям из БД. Это частный случай метода резолюций.

Отметим, что в Прологе не проводится синтаксического различия между предикатом и функцией (составным термом), а также между нечисловой константой и функцией без аргументов. Следовательно, суть действия состоит в том, что ищется попарное соответствие между термами, один из которых является целью, а другой принадлежит БД.

Установление соответствия между термами является основной операцией при вычислении цели. Она осуществляется следующим образом: на каждом шаге выбирается очередной терм и отыскивается соответствующее выражение в БД. При этом переменные получают или теряют значения. Этот процесс можно описать в терминах текстуальных

подстановок: «подставить терм t вместо переменной Y». Свободными переменными в Прологе называются переменные, которым не были присвоены значения, а все остальные переменные называются связанными переменными. Переменная становится связанной только во время унификации, переменная вновь становится свободной, когда унификация оказывается неуспешной или цель оказывается успешно вычисленной. В Прологе присваивание значений переменным выполняется внутренними подпрограммами унификации. Переменные становятся свободными, как только для внутренних подпрограмм унификации отпадает необходимость связывать некоторое значение с переменной для выполнения доказательства подцели.

Правила унификации.

1. Если x и y-константы, то они унифицируемы, только если они равны.
2. Если x- константа или функция, а Y-переменная, то они унифицируемы, при этом Y принимает значение x .
3. Если x и y -функции, то они унифицируемы тогда и только тогда, когда у них одинаковые имена функций (функторы) и набор аргументов и каждая пара аргументов функций унифицируемы.

Есть особый предикат «=», который используется в Прологе для отождествления двух термов. Использование оператора «=» поможет лучше понять процесс означивания переменной. В Прологе оператор «=» интерпретируется как оператор присваивания или как оператор проверки на равенство в зависимости от того, являются ли значения термов свободными или связанными.

Пример 17: X=Y, если X и Y – связанные переменные, то производится проверка на равенство, например: если X=5 и Y=5, то результат ДА (истина); если X=6 а Y=5, то результат НЕТ(ложь). Если одна из переменных X или Y – свободная, то ей будет присвоено значение другой переменной, для Пролога несущественно слева или справа от знака «=» стоит связанная переменная.

Оператор «=» ведет себя точно так, как внутренние подпрограммы унификации при сопоставлении целей или подцелей с фактами и правилами программы.

2.5 Вычисление цели. Механизм возврата.

Каноническая форма цели (вопроса) является конъюнкцией атомарных предикатов, то есть последовательностью подцелей, разделенных запятыми [1]:

$Q=Q_1, Q_2, \ldots, Q_n$.

Пролог пытается вычислить цель при помощи унификации термов предикатов подцелей с соответствующими элементами в фактах и заголовках правил. Поиск ответа на вопрос напоминает поиск пути в лабиринте: следует

поворачивать налево в каждой развилке лабиринта до тех пор, пока не попадете в тупик. В этом случае следует вернуться к последней развилке и повернуть направо, после чего опять следует повернуть налево и так далее. Унификация выполняется слева направо, как и поиск пути в лабиринте.

Некоторые подцели при унификации с некоторыми фактами или правилами могут оказаться неуспешными, поэтому Прологу требуется способ запоминания точек отката, в которых он может продолжить альтернативные поиски решения. Прежде чем реализовать один из возможных путей вычисления подцели, Пролог фактически помещает в программу указатель, который определяет точку, в которую может быть выполнен возврат, если текущая попытка поиска цели будет неудачной.

Если некоторая подцель оказывается неуспешной, то Пролог возвращается влево к ближайшей точке возврата. С этой точки Пролог начинает попытку найти другое решение для неуспешной цели. До тех пор, пока следующая цель на данном уровне не будет успешной, Пролог будет повторять возврат к ближайшей точке возврата. Эти попытки выполняются внутренними подпрограммами унификации и механизмом возврата.

Замечание: единственным способом освободить переменную, связанную в предложении является откат при поиске с возвратом.

Алгоритм вычисления цели – частный случай правила резолюции применительно к дизъюнктам Хорна. Вопрос Q является правилом без заголовка, аналогом выражения $\neg Q$. Пусть D – база данных (множество дизъюнктов). На вопрос Q, следует найти такую подстановку σ, для которой множество $\sigma[D \cup (\neg Q)]$ невыполнимо. Стратегия выбора очередной пары дизъюнктов для резолюции здесь очень проста: подцели и предложения просматриваются в текстуальном порядке.

Пример 18: пусть есть БД семья:

1. *мать(мария, анна).*
2. *мать(мария, юлия).*
3. *мать(анна, петр).*
4. *отец(иван, анна).*
5. *отец(иван, юлия).*
6. *дед (X, Y): - отец(X, Z), мать(Z, Y).*
7. *дед (X, Y): - отец(X, Z), отец(Z, Y).*
8. *бабка (X, Y): - мать(X, Z), мать(Z, Y).*
9. *бабка (X, Y): - мать(X, Z), отец(Z, Y).*

Зададим сложную цель:

Q1, Q2 = отец(X, Y), мать(мария, Y).

Подцель Q1= отец(X,Y) соответствует четвертому предложению БД :отец (иван,анна). Это дает подстановку σ_1*={X=иван,Y=анна}. Затем найденная подстановка применяется к* $\sigma_1[Q2]$*= мать(мария, анна). Этой подцели соответствует 1 предложение БД, что дает пустую подцель по правилу резолюции, следовательно ответ найден: X= иван, Y= анна.*

Для получения нового ответа в БД ищется новая унификация для $\sigma_1[Q_2]$. *Так как в БД нет соответствующего предложения, то вычисления прекращаются, система вновь рассматривает последовательность* Q_1, Q_2 *и для* Q_1 *ищется новая унификация в БД, начиная с пятого предложения. Это и есть возврат. Пятое предложение сразу же дает желаемую унификацию с подстановкой* σ_2*={X=иван,Y=юлия}. Вновь найденная подстановка применяется к* $\sigma_1[Q_2]$*= мать(мария, юлия). Этой подцели соответствует второе предложение БД. Далее указанная процедура повторяется и в итоге имеем:*

Цель: отец(X, Y), мать(мария, Y).

2 решения: X=иван, Y=анна

X=иван, Y=юлия.

Это описание объясняет, как работает утилита TestGoal в Visual Prolog.

При реализации механизма возврата выполняются следующие правила:

1. Подцели вычисляются слева-направо.
2. Предложения при вычислении подцели проверяются в текстуальном порядке, то есть сверху-вниз.
3. Если подцель сопоставима с заголовком правила, то должно быть вычислено тело этого правила, при этом тело правила образует новое подмножество подцелей для вычисления.
4. Цель считается успешно вычисленной, когда найден соответствующий факт для каждой подцели.

Если в Visual Prolog создать программу для автономного исполнения (то есть создать свой project-файл для разрабатываемой программы), то поиск цели в ней будет вестись так же, как для внутренней цели в PDC Prolog, то есть до первого успешного решения.

2.6 Управление поиском решения.

Встроенный в Пролог механизм поиска с возвратом может привести к поиску ненужных решений, в результате чего снижается эффективность программы в случае, если надо найти только одно решение. В других случаях бывает необходимо продолжить поиск, даже если решение найдено.

Пролог обеспечивает два встроенных предиката, которые дают возможность управлять механизмом поиска с возвратом: предикат *fail* – используется для инициализации поиска с возвратом и предикат *отсечения !* – используется для запрета возврата.

Предикат fail всегда имеет ложное значение!

Отсечение так же, как и *fail* помещается в тело правила. ***Однако, в отличие от fail предикат отсечения имеет всегда истинное значение.***

При этом выполняется обращение к другим предикатам в теле правила, следующим за отсечением. Следует иметь в виду, что невозможно произвести возврат к предикатам, расположенным в теле правила перед

отсечением, а также невозможен возврат к другим правилам данного предиката.

Существует только два случая применения предиката отсечения:

1. Если заранее известно, что определенные посылки никогда не приведут к осмысленным решениям – это так называемое «зеленое отсечение».
2. Если отсечения требует сама логика программы для исключения альтернативных подцелей – это так называемое «красное отсечение».

Пример 19: Использование предикатов ! и fail. Для примера 16 можно добавить правила для печати всех пар «мать-ребёнок», которые есть в БД:

1.печать_1(X,Y):-мать(X,Y), write(X,” есть мать”,Y),nl.
goal
печать_1(X,Y).
В результате будет выдано 3 решения:
X=мария, Y= анна.
X=мария, Y= юлия.
X=анна, Y= петр.
2.печать_2:-мать(X,Y), write(X,” есть мать”,Y),nl,fail.
goal
печать_2.
В результате будет выдано 3 решения:
X=мария, Y= анна.
X=мария, Y= юлия.
X=анна, Y= петр.
3.печать_3(X,Y):-мать(X,Y), write(X,” есть мать”,Y),nl,!.
goal
печать_3(X,Y).
В результате будет выдано 1 решение:
X=мария, Y= анна.

2.7 Процедурность Пролога.

Пролог – декларативный язык. Описывая задачу в терминах фактов и правил, программист предоставляет Прологу самому искать способ решения. В процедурных языках программист должен сам писать процедуры и функции, которые подробно «объясняют» компьютеру, какие шаги надо сделать для решения задачи.

Тем не менее, рассмотрим Пролог с точки зрения процедурного программирования:

1. Факты и правила можно рассматривать как определения процедур.
2. Использование правил для условного ветвления программы. Правило, в отличие от процедуры, позволяет задавать множество

альтернативных определений одной и той же процедуры. Поэтому, правило можно считать аналогом оператора *case* в Паскале.

3. В правиле может быть выполнено сравнение, как в условных операторах.
4. Отсечение можно считать аналогом *go to*.
5. Возврат вычисленного значения производится аналогично процедурам. В Прологе это делается путем связывания свободных переменных при сопоставлении цели с фактами и правилами.

2.8 Структура программ Пролога.

Программа, написанная на Прологе, состоит из шести основных разделов: раздел описания доменов, раздел базы данных, раздел описания предикатов, раздел описания предложений и раздел описания цели. Ключевые слова *domains, constants, facts (database), predicates, clauses и goal* отмечают начала соответствующих разделов. Назначение этих разделов таково:

- раздел *domains* содержит объявления доменов, которые описывают различные типы данных, используемых в программе, если в программе не требуется объявления доменов, то этот раздел может быть опущен;
- раздел *constants* используется для объявления символических констант, используемых в программе, если в программе не требуется объявления символических констант, то этот раздел может быть опущен;
- раздел *facts (database)* содержит описания предикатов внутренней базы данных Пролога, если программа такой базы данных не требует, то этот раздел может быть опущен;
- раздел *predicates* служит для описания предикатов, не принадлежащих внутренней базе данных, если в программе не требуется объявления предикатов, то этот раздел может быть опущен;
- в раздел *clauses* заносятся факты и правила самой программы, если в программе используются только встроенные предикаты, то этот раздел может быть опущен;
- в разделе *goal* на языке Пролог формулируется назначение создаваемой программы. ***Данный раздел программы является обязательным.*** Составными частями при этом могут являться некие подцели, из которых формируется единая цель программы.

В Visual Prolog разрешает объявление разделов *domains, facts, predicates, clauses* как глобальных разделов, то есть с ключевым словом *global*.

Пролог имеет следующие встроенные типы доменов:

Тип данных	**Ключевое слово**	**Диапазон значений**	**Примеры использования**
Символы	char	Все возможные символы	‘a’, ’b’, ‘#’, ‘B’, ‘%’
Целые числа	integer byte word dword	От –32768 до 32767 От 0 до 255 От 0 до 65535 От 0 до	-63, 84, 2349
Действительные числа	real short ushort long ulong unsigned	От +1E-307 до +1E308 16 битов со знаком 16 битов без знака 32 бита со знаком 32 бита без знака 16 или 32 бита без знака	360, - 8324, 1.25E23, 5.15E-9
Строки	string	Последовательность символов (не более 250)	«today», «123», «school_day»
Символические имена	symbol	1.Последовательность букв, цифр, символов подчеркивания; первый символ – строчная буква. 2. Последовательность любых символов, заключенная в кавычки.	flower, school_day «string and symbol»
Ссылочный тип	ref		
Файлы	file	Допустимое в DOS имя файла	mail.txt, LAB.PRO

Если в программе необходимо использовать новые домены данных, то они должны быть описаны в разделе *domains*.

Пример 20:

```
domains
number=integer
name, person=symbol.
```

Различие между symbol и string - в машинном представлении и выполнении, синтаксически они не различимы.

Visual Prolog выполняет автоматическое преобразование типов между доменами string и symbol. Однако, по принятому соглашению, символическую строку в двойных кавычках нужно рассматривать как string, а без кавычек – как symbol:

Symbol - имена, начинающиеся с символа нижнего регистра и содержащие только символы, цифры, и символы подчеркивания.

String – в двойных кавычках могут содержать любую комбинацию символов, кроме #0, который отмечает конец строки.

Visual Prolog поддерживает и другие типы стандартных доменов данных, например, для работы с внешними БД или объектами.

Предикаты описываются в разделе predicates. Предикат представляет собой строку символов, первым из которых является строчная буква. Предикаты могут не иметь аргументов, например «go» или «repeat». Если предикаты имеют аргументы, то они определяются при описании предикатов в разделе predicates:

Пример 21:

predicates

mother (symbol, symbol)

father (symbol, symbol).

Факты и правила определяются в разделе clauses, а вопрос к программе задается в разделе goal – в этом случае цель называется внутренней целью. В Visual Prolog раздел goal в тексте программы является обязательным. Разница в режимах исполнения программы состоит в разном использовании утилиты Test Goal. Если утилита создается для запуска любой программы, то при этом ищутся все решения, если утилита создается для автономного запуска программы – то ищется одно решение.

2.9 Использование составных термов

В Прологе функциональный терм или предикат можно рассматривать как структуру данных, подобную записи в языке Паскаль. Терм, представляющий совокупность термов, называется составным термом или структурой. Составные структуры данных в Прологе объявляются в разделе domains. Если термы структуры относятся к одному и тому же типу доменов, то этот объект называется однодоменной структурой данных. Если термы структуры относятся к разным типам доменов, то такая структура данных называется многодоменной структурой данных. Использование доменной структуры упрощает структуру предиката.

Аргументами составного терма данных могут быть простые типы данных, составные термы или списки.

Синтаксически составной терм выглядит так же, как и предикат: у терма есть функтор и список аргументов, заключенных в круглые скобки.

Составной терм может быть унифицирован с простой переменной или составным объектом (при этом переменные могут быть использованы как часть внутренней структуры терма). Это означает, что составной объект можно использовать для того, чтобы передавать целый набор значений, как единый объект, а затем применять унификацию для их разделения.

Пример 22: Необходимо создать БД, содержащую сведения о книгах из личной библиотеки. Зададим составной терм с именем personal_library, имеющим следующую структуру: personal_library= book (title, author, publisher, year), и предикат collection (collector, personal_library). Терм book называется функтором структуры данных. Пример программы, использующей составные термы для описания личной библиотеки и поиска информации о книгах, напечатанных после 1990 года, выглядит следующим образом:

```
domains
collector, title, author, publisher = symbol
year = integer
book = book (title, author, publisher, year)
predicates
collection (collector, book)
q1(title, year)
q2(book)
clauses
collection (irina, book («Using Turbo Prolog», «Yin with Solomon», «Moscow, World», 1993)).
collection (petr, book («The art of Prolog», «Sterling with Shapiro», »Moscow, World», 1990)).
collection (anna, book («Логическое программирование и Visual Prolog», «Адаменко А.Н., Кучуков А.М.», «СПб.: БХВ-Петербург», 2003)).
q1(N, Y):- collection (_, book( N,_, _, Y)),Y>1990.
```

q2(Z):- collection (_, Z), Z=book(_,_, _, Y),Y>1990.

goal

Для первого запроса цель будет выглядеть следующим образом: *q1(N, Y).* В данном случае переменные *N* и *Y* используются для унификации части составного терма *book.*

Для второго запроса цель надо задать в виде: *q2(Z).*

Простая переменная Z унифицируется с составным термом *book.*

В Прологе структуры могут иметь аргументы в виде списков. Модифицируем предыдущий пример таким образом, чтобы вместо года издания в структуре *book* был задан список лет переизданий книги.

Пример 23: Модифицированный пример 22.

domains

collector, title, author, publisher = symbol

year = integer

*list_year=year**

book = book (title, author, publisher, list_year)

predicates

collection (collector, book)

q1(year)

search(title, year, list_year)

clauses

collection (irina, book («Using Turbo Prolog», «Yin with Solomon», «Moscow, World», [1993,1995])).

collection (petr, book («The art of Prolog», «Sterling with Shapiro», »Moscow, World», [1990,1998])).

collection (anna, book («Логическое программирование и Visual Prolog», «Адаменко А.Н., Кучуков А.М.», «СПб.: БХВ-Петербург», [2003])).

q1(Y):- collection (_, book(N,_, _, L)),search(N,Y,L), fail.

search(_,_,[]).

search(N,Y,[H|T]):-H>Y, write(N,' ',H), nl, search(N,Y,T).

search(N,Y,[H|T]):-H<=Y, search(N,Y,T).

goal

q1(1990).

2.10 Использование списков

Список является составной рекурсивной структурой данных, хотя явно и не объявляется как рекурсивная структура. Список – это упорядоченный набор объектов одного и того же типа. Элементами списка могут быть целые числа, действительные числа, символы, строки, символические имена и структуры. Порядок расположения элементов в списке играет важную роль: те же самые элементы списка, упорядоченные иным способом, представляют уже совсем другой список [4].

Совокупность элементов списка заключается в квадратные скобки ([]), элементы друг от друга отделяются запятыми. Список может содержать

произвольное число элементов, единственным ограничением является объем оперативной памяти. Количество элементов в списке называется его длиной. Список может содержать один элемент и даже не содержать ни одного элемента. Список, не содержащий элементов, называется пустым или нулевым списком.

Непустой список можно рассматривать как список, состоящий из двух частей: головы – первого элемента списка; и хвоста – остальной части списка. Голова является элементом списка, хвост является списком. Голова списка – это неделимое значение, хвост представляет собой список, составленный из того, что осталось от исходного списка в результате «отделения головы». Этот новый список обычно можно делить и дальше. Если список состоит из одного элемента, то его можно разделить на голову, которой будет этот самый элемент, и хвост, являющийся пустым списком.

Пустой список нельзя разделить на голову и хвост!

Операция деления списка на голову и хвост обозначается при помощи вертикальной черты (|):

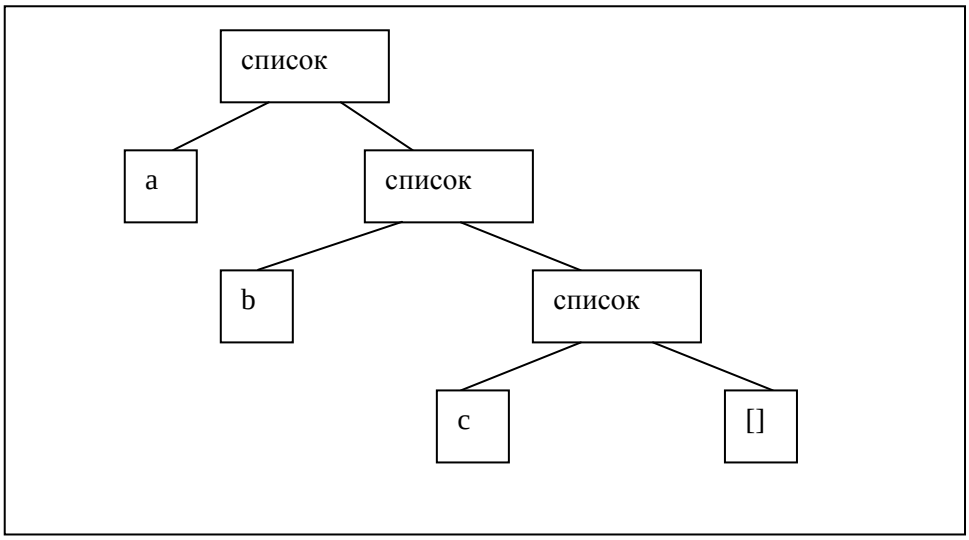
[Head | Tail].

Голова списка всегда имеет тип элемента списка, хвост списка – тип списка!

Head здесь является переменной для обозначения головы списка, переменная Tail обозначает хвост списка (для имен головы и хвоста списка пригодны любые допустимые Прологом имена).

Данная операция также присоединяет элемент в начало списка, например, для того, чтобы присоединить X к списку S следует написать [X|S].

В концептуальном плане, список имеет структуру дерева, как и другие составные термы. Так, например, список *[a,b,c]* можно представить в виде структуры:



Отличительной особенностью описания списков является наличие звездочки (*) после имени домена элементов.

Пример 24: объявление списков, состоящих из элементов стандартных типов доменов или типа структуры.

```
domains
list1=integer*
list2=char*
list3=string*
list4=real*
list5=symbol*
personal_library = book (title, author, publisher, year)
list6= personal_library*
list7=list1*
list8=list5*
```

В первых пяти объявлениях списков в качестве элементов используются стандартные домены данных, в шестом типе списка в качестве элемента используется домен структуры personal_library, в седьмом и восьмом типе списка в качестве элемента используется ранее объявленный список.

Пример 25: демонстрация разделения списков на голову и хвост.

Список	*Голова*	*Хвост*
[1, 2, 3, 4, 5]	*1*	*[2, 3, 4, 5]*
[6.9, 4.3]	*6.9*	*[4.3]*
[cat, dog, horse]	*Cat*	*[dog, horse]*
[‘S’, ‘K’, ‘Y’]	*‘S’*	*[‘K’, ‘Y’]*
[«PIG»]	*«PIG»*	*[]*
[]	*Не определена*	*Не определен*

Visual Prolog допускает объявление и использвание составных списков. Составной список – это список, в котром используется более чем один тип элемента. Для создания такого списка, необходимо использовать структуры, так как домен может содержать более одного типа данных только для структуры.

Пример 26: объявление списка, который может содержать символы, целые числа или строки:

```
domains
llist=i(integer);c(char);s(string)
list=llistl*
```

2.11 Применение списков в программах

Для применения списков в программах на Прологе необходимо описать домен списка в разделе *domains*, предикаты, работающие со

списками необходимо описать в разделе *predicates*, задать сам список можно либо в разделе *clauses* либо в разделе *goal*.

Над списками можно реализовать различные операции: поиск элемента в списке, разделение списка на два списка, присоединение одного списка к другому, удаление элементов из списка, сортировку списка, создание списка из содержимого БД и так далее.

2.11.1 Поиск элемента в списке

Поиск элемента в списке является очень распространенной операцией. Поиск представляет собой просмотр списка на предмет выявления соответствия между объектом поиска и элементом списка. Если такое соответствие найдено, то поиск заканчивается успехом, в противном случае поиск заканчивается неуспехом. Стратегия поиска при этом будет состоять в рекурсивном выделении головы списка и сравнении ее с объектом поиска.

Пример 27: поиск элемента в списке.

```
domains
list=integer*
predicates
member (integer, list)
clauses
member (Head, [Head |_ ]).
member (Head, [_ | Tail ]):- member (Head, Tail).
goal
member (3, [1, 4, 3, 2]).
```

Правило поиска может сравнить объект поиска и голову текущего списка, эта операция записана в виде факта предиката member. Этот вариант предполагает наличие соответствия между объектом поиска и головой списка. Отметим, что хвост списка в данном случае не важен, поэтому хвост списка присваивается анонимной переменной. Если объект поиска и голова списка различны, то в результате исполнения первого предложения будет неуспех, происходит возврат и поиск другого правила или факта, с которыми можно попытаться найти соответствие. Для этой цели служит второе предложение, которое выделяет из списка следующий по порядку элемент, то есть выделяет голову текущего хвоста, поэтому текущий хвост представляется как новый список, голову которого можно сравнить с объектом поиска. В случае исполнения второго предложения, голова текущего списка ставится в соответствие анонимной переменной, так как значение головы с писка в данном случае не играет никакой роли.

Процесс повторяется до тех пор, пока первое предложение даст успех, в случае установления соответствия, либо неуспех, в случае исчерпания списка. В представленном примере предикат find находит все совпадения объекта поиска с элементами списка. Для того, чтобы найти только первое

совпадение следует модифицировать первое предложение следующим образом:

member (Head, [Head |_]):- !.

Отсечение отменяет действие механизма возврата, поэтому поиск альтернативных успешных решений реализован не будет.

2.11.2 Объединение двух списков

Слияние двух списков и получение, таким образом, третьего списка принадлежит к числу наиболее полезных при работе со списками операций. Обозначим первый список *L1*, а второй список - *L2*. Пусть *L1 = [1, 2, 3]*, а *L2 = [4, 5]*. Предикат *append* присоединяет *L2* к *L1* и создает выходной список *L3*, в который он должен переслать все элементы *L1* и *L2*. Весь процесс можно представить следующим образом:

1. Список *L3* вначале пуст.
2. Элементы списка *L1* пересылаются в *L3*, теперь значением *L3* будет *[1, 2, 3]*.
3. Элементы списка *L2* пересылаются в *L3*, в результате чего тот принимает значение *[1, 2, 3, 4, 5]*.

Тогда программа на языке Пролог имеет следующий вид:

Пример 28: объединение двух списков.

```
domains
list=integer*
predicates
append (list, list, list)
clauses
append ( [], L2, L2).
append ([H|T1], L2, [H|T3 ]):- append (T1, L2, T3).
goal
append ( [1, 2, 3], [4, 5], L3).
```

Основное использование предиката *append* состоит в объединении двух списков, что делается при помощи задания цели вида *append ([1, 2, 3], [4, 5], L3)*. Поиск ответа на вопрос типа: *append (L1, [3, 4, 5], [1, 2, 3, 4, 5])* – сводится к поиску такого списка *L1=[1, 2]*, который при слиянии со списком *L2 = [3, 4, 5]* даст список *L3 = [1, 2, 3, 4, 5]*. При обработки *цели append (L1, L2, [1, 2, 3, 4, 5])* ищутся такие списки *L1* и *L2*, что их объединение даст список *L3 = [1, 2, 3, 4, 5]*.

2.11.3 Определение длины списка

Число элементов в списке можно подсчитать при помощи рекурсивных предикатов *count_list1 и count_list2*. Предикат *count_list1* ведёт подсчёт числа элементов в списке на прямом ходе рекурсии, начиная от головы списка, при этом первый параметр типа *byte* является текущим счётчиком, а второй - необходим для возвращения результата при выходе из рекурсии. Предикат *count_list2* ведёт подсчёт числа элементов в списке на обратном ходе

рекурсии, начиная с последнего элемента, при этом параметр типа *byte* является текущим счётчиком и результатом одновременно. Два варианта решения задачи приводятся в примере 29.

Пример 29: определение длины списка.

```
domains
list=integer*
predicates
count_list1(list,byte,byte)
count_list2(list,byte)
clauses
count_list1([],N,N).
count_list1([_|T],N,M):- N1=N+1, count_list1(T,N1,M).
count_list2([],0).
count_list1([_|T],N):- count_list1(T,N1), N=N1+1.
goal
count_list1([0,-1,2,6,-9],0,N), count_list2([4,3,-8],K).
```

2.11.4 Поиск максимального и минимального элемента в списке

Найти максимальный или минимальный элемент в списке можно при помощи рекурсивных предикатов *max_list и min_list.* Предикат *max_list* ищет максимальный элемент в списке на прямом ходе рекурсии, начиная от головы списка, при этом первый параметр типа *integer* является текущим максимумом, а второй - необходим для возвращения результата при выходе из рекурсии. Предикат *min_list* ищет минимальный элемент в списке на обратном ходе рекурсии, начиная с последнего элемента, при этом параметр типа *integer* является текущим минимумом и результатом одновременно. Два варианта решения задачи приводятся в примере 30.

Пример 30: поиск максимального и минимального элемента в списке.

```
domains
list=integer*
predicates
max_list(list, integer, integer)
min_list(list, integer)
clauses
max_list ([],M,M).
max_list ([H|T],N,M):- H>N, max_list(T,H,M).
max_list ([H|T],N,M):- H<=N, max_list(T,N,M).
min_list ([H|[]],H).
min_list ([H|T],M):- min_list(T,M1), H<=M1, M=H.
min_list ([H|T],M):- min_list(T,M1), H>M1, M=M1.

goal
L=[1,5,3,-6,8,-4],L=[H|T],max_list(T,H,Max), min_list([4,3,-8,6],Min).
```

2.11.5 Сортировка списков

Сортировка представляет собой переупорядочение элементов списка определенным образом. Назначением сортировки является упрощение доступа к нужным элементам. Для сортировки списков обычно применяются три метода:

- метод перестановки,
- метод вставки,
- метод выборки.

Также можно использовать комбинации указанных методов.

Первый метод сортировки заключается в перестановке элементов списка до тех пор, пока он не будет упорядочен. Второй метод осуществляется при помощи неоднократной вставки элементов в список до тех пор, пока он не будет упорядочен. Третий метод включает в себя многократную выборку и перемещение элементов списка.

Второй из методов, метод вставки, особенно удобен для реализации на Прологе.

Пример 31: сортировка списков методом перестановки (пузырька).

```
domains
list=integer*
predicates
puz(list,list)
perest(list,list)
clauses
puz(L1,L2):-perest(L1,L3),!,puz(L3,L2).
puz(L1,L1).
perest([X,Y|T],[Y,X|T]):-X>Y.
perest([Z|T],[Z|T1]):-perest(T,T1).
goal
puz([1,3,4,5,2],L).]
```

Пример 32: сортировка списков методом вставки.

```
domains
list=integer*
predicates
insert_sort (list, list)
insert (integer, list, list)
asc_order (integer, integer)
clauses
insert_sort ( [], []).
insert_sort ([H1|T1], L2):- insert_sort (T1, T2),
                                    insert(H1, T2, L2).
insert (X, [H1| T1], [H1| T2]) :- asc_order (X, H1), !,
                                  insert (X, T1, T2).
insert (X, L1, [X| L1]).
```

asc_order (X, Y):- X>Y.

goal

insert_sort ([4, 7, 3, 9], L).

Для удовлетворения первого правила оба списка должны быть пустыми. Для того, чтобы достичь этого состояния, по второму правилу происходит рекурсивный вызов предиката *insert_sort,* при этом значениями *H1* последовательно становятся все элементы исходного списка, которые затем помещаются в стек. В результате исходный список становится пустым и по первому правилу выходной список также становится пустым.

После того, как произошло успешное завершение первого правила, Пролог пытается выполнить второй предикат *insert,* вызов которого содержится в теле второго правила. Переменной *H1* сначала присваивается первое взятое из стека значение *9,* а предикат принимает вид *insert (9, [], [9]).*

Так как теперь удовлетворено все второе правило, то происходит возврат на один шаг рекурсии в предикате *insert_sort.* Из стека извлекается 3 и по третьему правилу вызывается предикат *asc_order,* то есть происходит попытка удовлетворения пятого правила *asc_order (3, 9):- 3>9.* Так как данное правило заканчивается неуспешно, то неуспешно заканчивается и третье правило, следовательно, удовлетворяетсячетвертое правило и *3* вставляется в выходной список слева от *9*: *insert (3, [9], [3, 9]).*

Далее происходит возврат к предикату *insert_sort*, который принимает следующий вид: *insert_sort ([3, 9], [3, 9]).*

На следующем шаге рекурсии из стека извлекается 7 и по третьему правилу вызывается предикат *asc_order* в виде *asc_order (7, 3):- 7>3.* Так как данное правило заканчивается успешно, то элемент 3 убирается в стек и *insert* вызвается рекурсивно еще раз, но уже с хвостом списка – *[9]: insert (7, [9], _).* Так как правило *asc_order (7, 9):- 7>9* заканчивается неуспешно, то выполняется четвертое правило, происходит возврат на предыдущие шаги рекурсии сначала *insert,* затем *insert_sort.*

В результате *7* помещается в выходной список между элементами *3* и *9*: *insert (7, [3, 9], [3, 7, 9]).*

При возврате еще на один шаг рекурсии из стека извлекается 4 и по третьему правилу вызывается предикат *asc_order* в виде *asc_order (4, 3):- 4>3.* Так как данное правило заканчивается успешно, то элемент 3 убирается в стек и *insert* вызвается рекурсивно еще раз, но уже с хвостом списка – *[7, 9]: insert (4, [7, 9], _).* Так как правило *asc_order (4, 7):- 4>7* заканчивается неуспешно, то выполняется четвертое правило, происходит возврат на предыдущие шаги рекурсии сначала *insert,* затем *insert_sort.*

В результате *4* помещается в выходной список между элементами *3* и *7*:

insert (4, [3, 7, 9], [3, 4, 7, 9]).

insert_sort [4, 7, 3, 9], [3, 4, 7, 9]).

2.11.6 Компоновка данных в список

Иногда при программировании определенных задач возникает необходимость собрать данные из фактов БД в список для последующей их обработки. Пролог содержит встроенный предикат *findall*, который позволяет выполнить данную операцию. Описание предиката *findall* выглядит следующим образом:

Findall (Var_, Predicate_, List_), где Var_ обозначает имя для терма предиката Predicate_, в соответствии с типом которого формируются элементы списка List_.

Пример 33: использование предиката findall.

```
domains
d=integer
predicates
decimal (d)
write_decimal
clauses
decimal (0)
decimal (1)
decimal (2)
decimal (3)
decimal (4)
decimal (5)
decimal (6)
decimal (7)
decimal (8)
decimal (9)
write_decimal:- findall(C, decimal (C), L), write (L).
goal
write_decimal.
```

Пример 34: использование предиката findall для предиката с несколькими параметрами.

```
domains
name, address=string
age, sum=word
lista=age*
listn=name*
predicates
person(name, address, age)
sumlist( lista, byte, sum)
clauses
sumlist( [],0, 0).
sumlist( [H|T],N, Sum):- sumlist( T, N1 Sum1), Sum=H+Sum1, N=N1+1.
```

person(“Иванов Пётр”, “Московское шоссе, 34”, 20).
person(“Сидоров Иван”, “улица Гая, 43”, 22).
person(“Петров Сидор”, “улица Врубеля, 15”, 20).
goal
findall(Age, person(_,_,Age), List), sumlist(List, N, Sum),
Ave=Sum/N,
findall(Who, person(Who,_,20), L).

2.12 Повторение и рекурсия в Прологе

Очень часто в программах необходимо выполнить одну и ту же операцию несколько раз. В программах на Прологе повторяющиеся операции обычно выполняются при помощи правил, которые используют возврат и рекурсию [3]. Существует четыре способа построения итеративных и рекурсивных правил:

- механизм возврата;
- метод возврата после неудачи;
- правило повтора, использующее бесконечный цикл;
- обобщенное рекурсивное правило.

Правила повторений и рекурсии должны содержать средства управления их выполнением. Встроенные предикаты Пролога *fail* и *cut (!)* используются для управления возвратами, а условия завершения используются для управления рекурсией. Правила выполняющие повторения, используют возврат, а правила, выполняющие рекурсию, используют самовызов.

2.12.1 Механизм возврата

Возврат является автоматически инициируемым системой процессом, если не используются специальные средства управления им. Если в предикате цели есть хотя бы одна свободная переменная, то механизм возврата переберет все возможные способы связывания этой переменной, то есть реализует итеративный цикл.

Пример 34: распечатать все десятичные цифры.
domains
d=integer
predicates
decimal (d)
write_decimal (d)
clauses
decimal (0).
decimal (1).
decimal (2).
decimal (3).

```
decimal (4).
decimal (5).
decimal (6).
decimal (7).
decimal (8).
decimal (9).
write_decimal (C):- decimal (C), write (C), nl.
goal
write_decimal (C).
```

2.12.2 Метод возврата после неудачи

Метод возврата после неудачи может быть использован для управления вычислением внутренней цели при поиске всех возможных решений. Данный метод либо использует внутренний предикат Пролога *fail*, либо условие, которое порождает ложное значение.

Пример 35: распечатать все десятичные цифры.

```
domains
d=integer
predicates
decimal (d)
write_decimal
clauses
decimal (0)
decimal (1).
decimal (2).
decimal (3).
decimal (4).
decimal (5).
decimal (6).
decimal (7).
decimal (8).
decimal (9).
write_decimal:- decimal (C), write (C), nl, fail.
goal
write_decimal.
```

В программе есть 10 предикатов, каждый из которых является альтернативным предложением для предиката decimal (C). Во время попытки вычислить цель внутренние подпрогрммы унификации связывают переменную C с термом первого предложения, то есть с цифрой 0. Так как существует следующее предложение, которое может обеспечить вычисление подцели decimal (C), то помещается указатель возврата, значение 0 выводится на экран. Предикат fail вызывает неуспешное завершение правила, внутренние подпрограммы унификации выполняют

возврат и процесс повторяется до тех пор, пока не будет обработано последнее предложение.

Пример 36: подсчитать значения квадратов всех десятичных цифр.

```
domains
d =integer
predicates
decimal (d)
s (d, d)
cikl
clauses
decimal (0).
decimal (1).
decimal (2).
decimal (3).
decimal (4).
decimal (5).
decimal (6).
decimal (7).
decimal (8).
decimal (9).
s( X, Z):- Z=X*X.
cikl:-decimal (I), s(I , S), write (S), nl, fail.
goal
not(cikl).
```

Пример 37: необходимо выдать десятичные цифры до 5 включительно.

```
domains
d=integer
predicates
decimal (d)
write_decimal.
make_cut (d)
clauses
decimal (0).
decimal (1).
decimal (2).
decimal (3).
decimal (4).
decimal (5).
decimal (6).
decimal (7).
decimal (8).
decimal (9).
write_decimal:- decimal (C), write (C), nl, make_cut (C),!.
```

make_cut (C):-C=5.
goal
write_decimal.

Предикат ! используется для того, чтобы выполнить отсечение в указанном месте. Неуспешное выполнение предиката make_cut порождает предикат fail, который используется для возврата и доступа к цифрам до тех пор, пока цифра не будет равна 5.

Пример 38: необходимо выдать из БД первую цифру, равную 5.

```
domains
d=integer
predicates
decimal (d)
write_decimal
clauses
decimal (0).
decimal (5).
decimal (2).
decimal (3).
decimal (4).
decimal (5).
decimal (6).
decimal (5).
decimal (8).
decimal (9).
write_decimal:- decimal (C), C=5, write (C), nl, !.
goal
write_decimal.
```

Если из тела правила убрать предикат !, то будут найдены все три цифры 5, что является результатом применения метода возврата после неудачи. При внесении отсечения будет выдана единственная цифра 5.

2.12.3 Метод повтора, использующий бесконечный цикл

Вид правила повторения, порождающего бесконечный цикл:

repeat.
repeat:- repeat.

Первый *repeat* является предложением, объявляющим предикат *repeat* истинным. Однако, поскольку имеется еще один вариант для данного предложения, то указатель возврата устанавливается на первый *repeat*. Второй *repeat* – это правило, которое использует само себя как компоненту (третий *repeat*). Второй *repeat* вызывает третий *repeat*, и этот вызов вычисляется успешно, так как первый *repeat* удовлетворяет подцели *repeat*. Предикат *repeat* будет вычисляться успешно при каждой новой попытке его вызвать после возврата. Факт будет использоваться для выполнения всех подцелей. Таким образом, *repeat* это рекурсивное правило, которое никогда

не бывает неуспешным. Предикат *repeat* широко используется в качестве компонента других правил, который вызывает повторное выполнение всех следующих за ним компонентов.

Пример 39: ввести с клавиатуры целые числа и вывести их на экран. Программа завершается при вводе числа 0.

```
domains
d=integer
predicates
repeat
write_decimal
do_echo
check (d)
clauses
repeat.
repeat:- repeat.
write_decimal:-nl, write( «Введите, пожалуйста, цифры»), nl,
write(«Для останова, введите 0»), nl.
do_echo:- repeat, readln (D), write(D), nl, check (D), !.
check (0):- nl, write («-OK!»).
check(_):- fail.
goal
write_decimal, do_echo.
```

Правило *do_echo* – это конечное правило повтора, благодаря тому, что предикат *repeat* является его компонентом и вызывает повторение предикатов *readln, write, и check.* Предикат *check* описывается двумя предложениями. Первое предложение будет успешным, если вводимая цифра 0, при этом курсор сдвигается на начало следующей строки и на экране появляется сообщение *«OK!»* и процесс повторения завершается, так как после предиката *check* в теле правила следует предикат отсечения. Если введенное значение отличается от 0, то результатом выполнения предиката *check* будет *fail* в соответствии со вторым предложением. В этом случае произойдет возврат к предикату *repeat. Repeat* повторяет посылки в правой части правила, находящиеся правее *repeat* и левее условия выхода из бесконечного цикла.

2.13 Методы организации рекурсии

Рекурсивная процедура – это процедура, которая вызывает сама себя. В рекурсивной процедуре нет проблемы запоминания результатов ее выполнения потому, что любые вычисленные значения можно передавать из одного вызова в другой, как аргументы рекурсивного предиката.

Например, в приведенной ниже программе вычисления факториала (пример 48), приведен пример написания рекурсивного предиката. При этом Пролог создает новую копию предиката *factorial* таким образом, что он становится способным вызвать сам себя как самостоятельную процедуру.

При этом не происходит копирования кода выполнения, но все аргументы и промежуточные переменные копируются в стек, который создается каждый раз при вызове рекурсивного правила.

Когда выполнение правила завершается, занятая стеком память освобождается и выполнение продолжается в стеке правила-родителя.

Пример 40: написать программу вычисления факториала.

```
predicates
factorial (byte, word)
clauses
factorial (0, 1).
factorial (1, 1):-!.
factorial (N, R):- N1=N-1, factorial (N1, R1), R=N*R1.
goal
f (7, Result).
```

Для вычисления факториала используется последовательное вычисление произведения ряда чисел. Его значение образуется после извлечения значений соответствующих переменных из стека, используемых как список параметров для последнего предиката в теле правила. Этот последний предикат вызывается после завершения рекурсии.

Пример 41: написать программу, генерирующую числа Фибоначчи до заданного значения.

```
predicates
f (byte, word)
clauses
f (1, 1).
f (2, 1).
f(N, F):- N1=N-1, f (N1, F1), N2=N1-1, f (N2,F2), F=F1+F2.
goal
f (10, Fib).
```

У рекурсии есть три основных преимущества:

- логическая простота по сравнению с итерацией;
- широкое применение при обработке списков;
- возможность моделирования алгоритмов, которые нельзя эффективно выразить никаким другим способом (например, описания задач, содержащих в себе подзадачи такого же типа).

У рекурсии есть один большой недостаток – использование большого объема памяти. Всякий раз, когда одна процедура вызывает другую, информация о выполнении вызывающей процедуры должна быть сохранена для того, чтобы вызывающая процедура могла, после завершения вызванной процедуры, возобновить выполнение на том месте, где остановилась.

Рассмотрим специальный случай, когда процедура может вызвать себя без сохранения информации о своем состоянии.

Предположим, что процедура вызывается последний раз, то есть после вызванной копии, вызывающая процедура не возобновит свою работу, при

этом стек вызывающей процедуры должен быть заменен стеком вызванной копии. Тогда аргументам процедуры просто присваиваются новые значения, и выполнение возвращается на начало вызывающей процедуры. С процедурной точки зрения этот процесс напоминает обновление управляющих переменных в цикле.

Эта операция в Visual Prolog называется оптимизацией хвостовой рекурсии или оптимизацией последнего вызова.

Создание хвостовой рекурсии в программе на Прологе означает, что:

- вызов рекурсивной процедуры является самой последней посылкой в правой части правила;
- до вызова рекурсивной процедуры в правой части правила не было точек отката.

Приведем примеры хвостовой рекурсии.

Пример 42: рекурсивный счетчик с оптимизированной хвостовой рекурсией.

```
count(100).
count(N):-write(N),nl,N1=N+1,count(N1).
goal
nl, count(0).
```

Модифицируем этот пример так, чтобы хвостовой рекурсии не стало.

Пример 43: рекурсивный счетчик без хвостовой рекурсии.

```
count1(100).
count1(N):-write(N),N1=N+1,count1(N1),nl.
goal
nl, count1(0).
```

Из-за вызова предиката *nl* после вызова рекурсивного предиката должен сохраняться стек.

Пример 44: рекурсивный счетчик без хвостовой рекурсии.

```
count2(100).
count2(N):-write(N),nl,N1=N+1,count2(N1).
count2(N):-N<0, write("N – отрицательное число").
goal
nl, count2(0).
```

Здесь есть непроверенная альтернатива до вызова рекурсивного предиката (третье правило), которое должно проверяться, если второе правило завершится неудачно. Таким образом, стек должен быть сохранен.

Пример 45: рекурсивный счетчик без хвостовой рекурсии.

```
count3(100).
count3(N):-write(N),nl,N1=N+1,check(N1),count3(N1).
check(Z):-Z>=0.
check(Z):-Z<0.
goal
nl, count3(0).
```

Здесь тоже есть непроверенная альтернатива до вызова рекурсивного предиката (предикат *check)*. Случаи в примерах 44 и 45 хуже, чем в примере 42, так как они генерируют точки возврата.

Очень просто сделать рекурсивный вызов последним в правой части правила, но как избежать альтернатив? Для этого следует использовать предикат отсечения, который предотвращает возвраты в точки, левее предиката отсечения. Модифицируем 44 и 45 примеры так, чтобы была хвостовая рекурсия.

Пример 46: рекурсивный счетчик из примера 41 с хвостовой рекурсией.

```
count4(100).
count4(N):-N>0,!,write(N),N1=N+1,count4(N1).
count4(N):- write("N – отрицательное число").
goal
nl, count4(0).
```

Пример 47: рекурсивный счетчик из примера 42 с хвостовой рекурсией.

```
count5(100).
count5(N):-write(N),N1=N+1 check(N1),!, count5(N1).
check(Z):-Z>=0.
check(Z):-Z<0.
goal
nl, count5(0).
```

2.14 Создание динамических баз данных

В Прологе существуют специальные средства для организации внутренних и внешних баз данных. Эти средства рассчитаны на работу с реляционными базами данных. Внутренние подпрограммы унификации осуществляют автоматическую выборку фактов из внутренней (динамической) базы данных с нужными значениями известных параметров и присваивают значения неопределенным параметрам.

Раздел программы *facts* в *Visual Prolog* предназначен для описания предикатов динамической (внутренней) базы данных. База данных называется динамической, так как во время работы программы из нее можно удалять любые факты, а также добавлять новые факты. В этом состоит ее отличие от статических баз данных, где факты являются частью кода программы и не могут быть изменены во время исполнения.

Иногда бывает полезно иметь часть информации базы данных в виде фактов статической БД - эти данные заносятся в динамическую БД сразу после активизации программы. В общем случае, предикаты статической БД имеют другое имя, но ту же самую форму представления данных, что и предикаты динамической БД. Добавление латинской буквы d к имени предиката статической БД - обычный способ различать предикаты динамической и статической БД.

Следует отметить два ограничения, объявленные в разделе *facts* :

- в динамической базе данных Пролога могут содержаться только факты;
- факты базы данных не могут содержать свободные переменные.

Допускается наличие нескольких разделов *facts* , тогда в описании каждого раздела *facts* нужно явно указать его имя, например *facts – mydatabase*. В двух различных разделах facts нельзя использовать одинаковые имена предикатов. Также нельзя использовать одинаковые имена предикатов в разделах *facts* и *predicates*. Если имя базы данных не указывается, то ей присваивается стандартное имя *dbasedom*. Программа может содержать локальные безымянные разделы фактов, если она состоит из единственного модуля, который не объявлен как часть проекта. Среда разработки компилирует программный файл как единственный модуль только при использовании утилиты *TestGoal*. Иначе безымянный раздел фактов должен быть объявлен глобальным, то есть как *global facts*.

В Прологе есть специальные встроенные предикаты для работы с динамической базой данных:

- *assert;*
- *asserta;*
- *assertz;*
- *retract;*
- *retractall;*
- *save;*
- *consult.*

Предикаты *assert, asserta, assertz,* - позволяют занести факт в БД, а предикаты *retract, retractall* - удалить из нее уже имеющийся факт.

Предикат assert заносит новый факт в БД в произвольное место, предикат *asserta* добавляет новый факт перед всеми уже внесенными фактами данного предиката, *assertz* добавляет новый факт после всех фактов данного предиката.

Предикат *retract* удаляет из БД первый факт, который сопоставляется с заданным фактом, предикат *retractall* удаляет из БД все факты, которые сопоставляются с заданным фактом.

Предикат *save* записывает все факты динамической БД в текстовый файл на диск, причем в каждую строку файла заносится один факт. Если файл с заданным именем уже существует, то старый файл будет затерт.

Предикат consult записывает в динамическую БД факты, считанные из текстового файла, при этом факты из файла дописываются в имеющуюся БД. Факты, содержащиеся в текстовом файле должны быть описаны в разделе *domains*.

Пример 48: Написать программу, генерирующую множество 4-разрядных двоичных чисел и записывающих их в динамическую БД.

```
facts
dbin (byte, byte, byte, byte)
predicates
```

cifra (byte)
bin (byte, byte, byte, byte)
clauses
cifra (0).
cifra (1).
bin (A, B, C, D):- cifra (A), cifra (B), cifra (C), cifra (D),
assert (bin (A, B, C, D)).
goal
bin (A, B, C, D).

Пример 49: Написать программу, подсчитывающую число обращений к программе.

facts
dcount (word)
predicates
modcount
clauses
dcount (0).
modcount:- dcount (N), M=N+1, retract (dcount (N)),asserta (dcount (M)).
goal
modcount.

Пример 50: Написать программу, определяющую родственные отношения.

facts
dsisters(symbol,symbol)
dbrothers(symbol,symbol)
predicates
parents(symbol,symbol)
pol(symbol,symbol)
sisters(symbol,symbol)
brothers(symbol,symbol)
clauses
parents (anna, olga).
parents (petr, olga).
parents (anna, irina).
parents (petr, irina).
parents (anna, ivan).
parents (petr, ivan).
pol(olga, w).
pol(anna ,w).
pol(petr, m).
pol(irina, w).
pol(ivan, m).
sisters (X, Y):-dsisters(X, Y).
sisters (X, Y):- parents (Z, X), parents (Z,Y),pol(X,w),

pol(Y,w),not(X=Y),not(dsisters(X,Y)),
asserta(dsisters(X, Y)).
brothers (X ,Y):-dbrothers(X, Y).
brothers (X, Y):- parents (Z,X), parents l(Z,Y),pol(X,m),
pol(Y,m),not(X=Y),not(dbrothers(X,Y)),
asserta(dbrothers(X,Y)).
goal
sisters (X, Y), save ("mybase.txt").

Пример 51: Для базы данных, содержащей сведения о книгах из личной библиотеки, создадим внутреннюю базу данных, куда будем записывать результаты запросов:

domains
collector, title, author, publisher = symbol
year = integer
personal_library = book (title, author, publisher, year)
predicates
collection (collector, personal_library)
q1(collector, title, year)
q2(year)
facts
dq1(collector, title, year)
count(year,byte)
clauses
collection (irina, book («Using Turbo Prolog», «Yin with Solomon», «Moscow, World», 1993)).
collection (irina, book («The art of Prolog», «Sterling with Shapiro», »Moscow, World», 1990)).
collection (petr, book («The art of Prolog», «Sterling with Shapiro», »Moscow, World», 1990)).
collection (petr, book («Prolog: a relation language and its applications», «John Malpas», »Moscow, Science», 1990)).
collection (anna, book («Prolog: a relation language and its applications», «John Malpas», »Moscow, Science», 1990)).
q1(C,T,Y):-dq1(C,T,Y), write(;',C,' ',T,' ',Y),nl, fail.*
q1(C,T,Y):- not(dq1(C,T,_)),collection (C, book(T,_, _, 1990)), assert(dq1(C,T,Y)), write(C,' ',T,' ',Y),nl.
count(2100,0).
q2(Y):-collection (_, book(_, _, _, Y)), count(Yold,Nold), N=Nold+1, assert(count(Y,N)), write(Y,' ',N),nl, fail.
goal
q1(C,T,1990);q2(1990);count(Y,N).

2.15 Использование строк в Прологе.

Строка – это набор символов. При программировании на Прологе символы могут быть «записаны» при помощи алфавитно-цифрового представления или при помощи их *ASCII-кодов*. Обратный *слэш (*\), за которым непосредственно следует *ASCII-код (N)* символа, интерпретируется как символ. Для представления одиночного символа выражение *\N* должно быть заключено в апострофы *('\N')*. Для представления строки символов *ASCII-коды* помещаются друг за другом и вся строка заключается в кавычки *(«\N\N\N»)*.

Операции, обычно выполняемые над строками, включают:

- объединение строк для образования новой строки;
- разделение строки для создания двух новых строк, каждая из которых содержит некоторые из исходных символов;
- поиск символа или подстроки внутри данной строки.

Для удобства работы со строками Пролог имеет несколько встроенных предикатов, манипулирующих со строками:

- *str_len* – предикат для нахождения длины строки;
- *concat* – предикат для объединения двух строк;
- *frontstr* – предикат для разделения строки на две подстроки;
- *frontchar* – предикат для разделения строки на первый символ и остаток;
- *fronttoken* – предикат для разделения строки на лексему и остаток.

Синтаксис предиката *str_len*:

str_len (Str_value, Srt_length), где первый терм имеет тип *string*, а второй терм имеет тип *integer*.

Пример 52:

str_len («Today», L)- в данном случае перменная L получит значение 5;

str_len («Today», 5) – в данном случае будет выполнено сравнение длины строки «Today» и 5. Так как они совпали, то предикат выполнится успешно, если бы длина строки не была равна 5, то предикат вылонился бы неуспешно.

Синтаксис предиката *concat*:

concat (Str1, Str2, Str3), где все термы имеют тип string.

Пример 53:

concat («Today», «Tomorrow», S3)- в данном случае перменная S3 получит значение «TodayTomorrow»;

concat (S1, «Tomorrow», «TodayTomorrow») – в данном случае S1 будет присвоено значение «Today»;

concat («Today», S2, «TodayTomorrow») – в данном случае S2 будет присвоено значение «Tomorrow»;

concat («Today», «Tomorrow», «TodayTomorrow»)- будет проверена возможность склейки строк «Today» и «Tomorrow» в строку «TodayTomorrow».

Синтаксис предиката *frontstr*:

frontstr (Number, Str1, Str2, Str3), где терм *Number* имеет тип *integer*, а остальные термы имеют тип *string.* Терм *Number* задает число символов, которые должны быть скопированы из строки *Str1* в строку *Str2*, остальные символы будут скопированы в строку *Str3*.

Пример 54:

frontstr (6,«Expert systems», S2, S3)- в данном случае перменная S2 получит значение «Expert», а S3 получит значение ,« systems».

Синтаксис предиката frontchar:

frontchar (Str1, Char_, Str2), где терм *Char_* имеет тип *char,* а остальные термы имеют тип *string.*

Пример 55:

frontchar («Today », C, S2)- в данном случае перменная C получит значение «T», а S2 получит значение ,«oday»;

frontchar («Today », 'T', S2) – в данном случае S2 будет присвоено значение «oday»;

frontchar («Today», C, «oday») – в данном случае C будет присвоено значение «T»;

frontchar (S1, «T», «oday») – в данном случае S1 будет присвоено значение «Today»;

frontchar («Today», «T», «oday»)- будет проверена возможность склейки строк «T» и «oday» в строку «Today».

Синтаксис предиката *fronttoken*:

fronttoken (Str1, Lex, Str2), где все термы имеют тип *string*. В терм *Lex* копируется первая лексема строки *Str1*, остальные символы будут скопированы в строку *Str2*. Лексема – это имя в соответствии с синтаксисом языка Пролог или строчное представление числа или отдельный символ (кроме пробела).

Пример 56:

fronttoken («Expert systems», Lex, S2)- в данном случае перменная Lex получит значение «Expert», а S2 получит значение ,« systems».

fronttoken («$Expert», Lex, S2)- в данном случае перменная Lex получит значение «$», а S2 получит значение ,«Expert».

fronttoken («Expert systems», Lex, « systems»)- в данном случае перменная Lex получит значение «Expert»;

fronttoken («Expert systems», «Expert», S2)- в данном случае перменная S2 получит значение « systems»;

fronttoken (S1, «Expert», « systems»)- в данном случае перменная S1 получит значение «Expert systems»;

fronttoken («Expert systems», «Expert», « systems»)- в данном случае будет проверена возможность склейки лексемы и остатка в строку «Expert systems».

2.16 Преобразование данных в Прологе

Для преобразования данных из одного типа в другой Пролог имеет следующие встроенные предикаты:

upper_lower;
str_char;
str_int;
str_real;
char_int.

Все предикаты преобразования данных имеют два терма. Все предикаты имеют два направления преобразования данных в зависимости от того, какой терм является свободной или связанной переменной.

Пример 57:

```
upper_lower («STARS», S2).
upper_lower (S1,«stars»).
str_char («T», C).
str_char (S, ’T’).
str_int («123», N).
str_int (S, 123).
str_real («12.3», R).
str_real (S, 12.3).
char_int (‘A’, N).
char_int (C, 61).
```

В Прологе нет встроенных предикатов для преобразования действительных чисел в целые и наоборот, или строк в символы. На самом деле, правила преобразования данных типов очень просты и могут быть заданы в программе самими программистом.

Пример 58:

```
predicates
conv_real_int (real, integer)
conv_int_real (integer, real)
conv_str_symb (string, symbol)
clauses
conv_real_int (R, N):- R=N.
conv_int_real (N, R):- N=R.
conv_str_symb (S, Sb):- S=Sb.
goal
conv_real_int (5432.765, N). (N= 5432).
conv_int_real (1234, R). (R=1234).
conv_str_symb («Visual Prolog», Sb). (Sb=Visual Prolog).
```

Пример 59: преобразование строки в список символов с использованием предиката frontchar.

```
domains
list=char*
```

```
predicates
convert (string, list)
clauses
convert («», []).
convert (Str, [H\T]):- frontchar(Str, H, Str1),
                       convert(Str1, T).
```

2.17 Представление бинарных деревьев

Одной из областей применения списков является представление множества объектов. Недостатком представления множества в виде списка является неэффективная процедура проверки принадлежности элемента множеству. Используемый для этой цели предикат *member* неэффективен при использовании больших списков.

Для представления множеств могут использоваться различные древовидные структуры, которые обеспечивают более эффективную реализацию проверки принадлежности элемента множеству, в частности, в данном разделе для этой цели рассматриваются бинарные деревья.

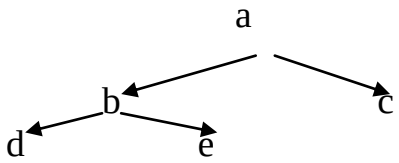
Представление бинарных деревьев основано на определении рекурсивной структуры данных, использующей функцию типа *tree (Top, Left, Right) или tree (Left, Top, Right),* где *Top* - вершина дерева, *Left и Right* - соответственно левое и правое поддерево. Пустое дерево обозначим термом *nil.* Объявить бинарное дерево можно следующим образом:

Пример 60:

```
domains
treetype1=tree(symbol, treetype1, treetype1);nil
treetype2=tree(treetype2, symbol, treetype2);nil
```

Пример 61:

Пусть дано дерево следующего вида:



Такое дерево может быть задано следующим образом:

1. левое поддерево: tree (b, tree (d, nil, nil), tree (e, nil, nil)).

2. правое поддерево: tree (c, nil, nil).

3. все дерево: tree (a, tree (b, tree (d, nil, nil), tree (e, nil, nil)), tree (c, nil, nil)).

Пример 62: написать программу проверки принадлежности вершины бинарному дереву.

```
domains
treetype = tree(symbol, treetype, treetype);nil()
predicates
```

```
in(symbol, treetype)
clauses
in(X, tree(X,_,_).
in(X, tree(_,L,_):-in(X, L).
in(X, tree(_,_,R):-in(X, R).
goal
in(d,tree(a, tree(b, tree(d, nil, nil),
                     tree(e, nil, nil)),
                     tree(c, nil, nil))).
```

Поиск вершины в неупорядоченном дереве также неэффективен, как и поиск элемента в списке. Если ввести отношение упорядочения между элементами множества, то процедура поиска элемента становится гораздо эффективнее. Можно ввести отношение *упорядочения слева направо непустого дерева tree(X, Left, Right)* следующим образом:

1. Все узлы в левом поддереве *Left* меньше *X*.
2. Все узлы в правом поддереве *Right* больше *X*.
3. Оба поддерева также являются упорядоченными.

Преимуществом упорядочивания является то, что для поиска любого узла в дереве достаточно провести поиск не более, чем в одном поддереве. В результате сравнения узла и корня дерева из рассмотрения исключается одно из поддеревьев.

Пример 63: написать программу проверки принадлежности вершины упорядоченному слева направо бинарному дереву.

```
domains
treetype = tree(byte, treetype, treetype);nil()
predicates
in(byte, treetype)
clauses
in(X, tree(X,_,_).
in(X, tree(Root,L,R):-Root>X,in(X, L).
in(X, tree(Root,L,R):-Root<X,in(X, R).
goal
in(6,tree(4, tree(2, nil, tree(3, nil, nil)),
      tree(5,tree(1,nil,nil),nil)),tree(8,tree(7,nil,nil),tree(9,nil,tree(10,nil,nil)
))).
```

Пример 64: написать программу печати вершин бинарного дерева, начиная от корневой и следуя правилу левого обхода дерева.

```
domains
treetype = tree(symbol, treetype, treetype);nil()
predicates
print_all_elements(treetype)
clauses
print_all_elements(nil).
print_all_elements(tree(X, Y, Z)) :-
```

```
        write(X), nl, print_all_elements(Y),
        print_all_elements(Z).
    goal
    print_all_elements(tree(a, tree(b, tree(d, nil, nil),
                                    tree(e, nil, nil)),
                                    tree(c, nil, nil))).
```

Пример 65: написать программу проверки изоморфности двух бинарных деревьев.

```
    domains
    treetype = tree(symbol, treetype, treetype);nil
    predicates
    isotree (treetype, treetype)
    clauses
    isotree (T, T).
    isotree (tree (X, L1, R1), tree (X, L2, R2)):- isotree (L1, L2), isotree (R1,
R2).
    isotree (tree (X, L1, R1), tree (X, L2, R2)):- isotree (L1, R2), isotree (L2,
R1).
```

Пример 66: написать предикаты создания бинарного дерева из одного узла и вставки одного дерева в качестве левого или правого поддерева в другое дерево.

```
    domains
    treetype = tree(symbol, treetype, treetype);nil
    predicates
    create_tree(symbol, tree)
    insert_left(tree, tree, tree)
    insert_rigth(tree, tree, tree)

    clauses
    create_tree(N, tree(N,nil,nil)).
    insert_left(X, tree(A,_,B), tree(A,X,B)).
    insert_rigth(X,tree(A,B,_), tree(A,B,X)).
    goal
    create_tree(a, T1)), insert_left(tree(b, nil, nil), tree(c, nil, nil), T2),
    insert_rigth(tree(d, nil, nil), tree(e, nil, nil), T3).
```

В результате: T1=tree(a, nil, nil), T2=tree(c, tree(b, nil, nil), nil), T3=tree(d, nil, tree(e, nil, nil)).

2.18 Представление графов в языке Пролог

Для представления графов в языке Пролог можно использовать три способа:

1. Использование факта языка Пролог для описания дуг или рёбер графа.
2. Использование списка рёбер.

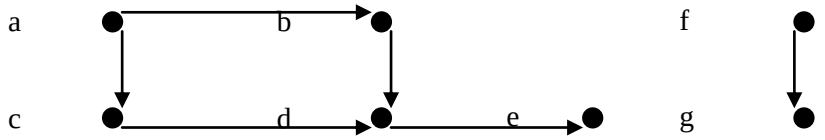
3. Использование списка списков: каждый подсписок в качестве головы содержит вершину графа, а в качестве хвоста - список смежных вершин.

Две самые распространённые операции, которые выполняются над графами:

- Поиск пути между двумя вершинами;
- Поиск в графе подграфа, который обладает некоторыми заданными свойствами.

Пример 67:

Определить наличие связи между вершинами графа, представленного на рисунке:



Две вершины графа связаны, если существует последовательность ребер, ведущая из первой вершины во вторую. Используем для описания структуры графа факты языка Пролог.

```
domains
top=symbol
predicates
edge (top, top)
/* аргументы обозначают имена вершин */
path( top,top)
/*Предикат connected(symbol, symbol) определяет отношение связанности вершин.*/
clauses
edge (a, b).
edge (c, d).
edge (a, c).
edge (d, e).
edge (b, d).
edge (f, g).
path (X, X).
path (X, Y):- edge (X, Z), path (Z, Y).
goal
path (a, d).
```

Пример 68:

Решить задачу из примера 67, используя списочные структуры для представления графа. Граф задается списком ребер. Ребро представлено при помощи структуры edge.

```
domains
edge= e(symbol, symbol)
list=edge*
predicates
path(list, symbol, symbol)
member(list,edge)
/*предикат path(graf, symbol, symbol) определяет отношение связности вершин.*/
clauses
member([H|_],H).
member([_|T],X):-member(T,X).
%path ([],_,_):-fail.
path(L,X,Y):-member(L,e(X,Y)),!.
path(L,X,Y):-member(L,e(X,Z)),path(L,Z,Y).
goal
path ([e(a, b),e(b, c),e(a, f),e(c, d),e(f,d)], a, d).
%path ([e(a, b),e(c, d),e(a, c),e(d, e),e(b, d),e(f, g)], b, g).
```

Пример 69:

Решить задачу из примера 67, используя списочные структуры для представления графа. Граф задается списком списков: в каждом подсписке голова является вершиной, а хвост – списком смежных вершин.

```
domains
list=symbol*
top=list*
graf=top*
predicates
path(graf, symbol, symbol)
member(symbol,list)
/*Предикат path(graf, symbol, symbol) определяет отношение связанности вершин в графе.*/
clauses
path ([],_,_):-fail.
path ([[X1|T1]|_],X,Y):-X1=[X], T1=[T|[]],member(Y,T).
path ([[X1|T1]|T2],X,Y):-X1=[X], T1=[T|[]],not(member(Y,T)),
                          T=[H|_],path(T2,H,Y).
path ([[Z|_]|T1],X,Y):-Z=[X1|[]],X<>X1,
path (T1,X,Y).
member(H,[H|_]):-!.
member(X,[_|T]):-member(X,T).
goal
path ([[[a],[b,c]],[[b],[d]],[[c],[d]],[[d],[e]],[[f],[g]]], a, c).
```

2.19 Поиск пути на графе.

Программы поиска пути на графе относятся к классу так называемых недетерминированных программ с неизвестным выбором альтернативы, то есть в таких программах неизвестно, какое из нескольких рекурсивных правил будет выполнено для доказательства до того момента, когда вычисление будет успешно завершено. По сути дела такие программы представляют собой спецификацию алгоритма поиска в глубину для решения определенной задачи. Программа проверки изоморфности двух бинарных деревьев, приведенная в примере 65 относится к задачам данного класса.

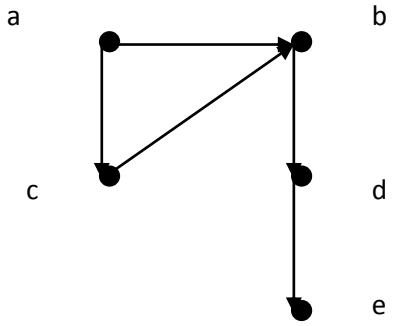
Пример 70:

Определить путь между вершинами графа, представленного на рисунке:

A- переменная обозначающая начало пути

B- вершина в которую нужно попасть

P -ациклический путь на графе (ациклический путь- это путь не имеющий повторяющих вершин).



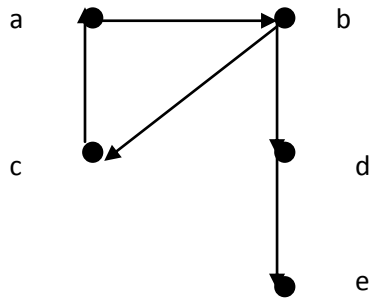

```
domains
top=symbol
listtop=top*
predicates
edge (top, top)
/* аргументы обозначают имена вершин */
path (top, top, listtop)
/*Предикат path( top, top, listtop) создает список из вершин, составляющих путь.*/
clauses
edge (a, b).
edge (b, c).
edge (c, a).
edge (b, d).
edge (d, e).
path (A, A, [A]).
```

path (A, B, [A\P]):-edge(A, N), path(N, B, P).

С помощью поиска в глубину осуществляется корректный обход любого конечного дерева или ориентированного ациклического графа. Однако, встречаются задачи, где требуется производить обход графа с циклами. В процессе вычислений может образоваться бесконечный программный цикл по одному из циклов графа.

Неприятностей с зацикливанием можно избежать двумя способами: ограничением на глубину поиска или накоплением пройденных вершин. Последний способ можно реализовать при помощи модификации отношения path. К аргументам отношения добавляется дополнительный аргумент, используемый для накопления уже пройденных при обходе вершин, для исключения повторного использования одного и того же состояния применяется проверка.

Пример 71: написать программу обхода конечного ориентированного графа, представленного на рисунке.




```
domains
top=symbol
listtop=top*
predicates
edge(top,top)
path (top,top,listtop,listtop)
path (top,top)
member(top,listtop)
reverse(listtop,listtop,listtop)
clauses
edge(a,b).
edge(b,c).
edge(c,a).
edge(b,d).
edge(d,e).
member(A,[A|_]):-!.
member(A,[_|T]):-member(A,T).
reverse([],T2,T2).
reverse([H|T],T1,T2):-reverse(T,[H|T1],T2).
path(A,B,P,[B|P]):-edge(A,B).
```

```
path(A,B,P,P2):-edge(A,N),not(member(N,P)),
                P1=[N|P], path (N,B,P1,P2).
path(A,B):-path(A,B,[A],P),reverse(P,[],Res),write(Res).
goal
path(a,e).
```

2.20 Метод “образовать и проверить”

Метод “образовать и проверить” – общий прием, используемый при проектировании алгоритмов и программ. Суть его состоит в том, что один процесс или программа генерирует множество предполагаемых решений задачи, а другой процесс или программа проверяет эти предполагаемые решения, пытаясь найти те из них, которые действительно являются решением задачи.

Используя вычислительную модель Пролога, легко создавать логические программы, реализующие метод “образовать и проверить”. Обычно такие программы содержат конъюнкцию двух целей, одна из которых действует как генератор предполагаемых решений, а вторая проверяет, являются ли эти решения приемлемыми. В Прологе метод “образовать и проверить” рассматривается как метод недетерминированного программирования. В такой недетерминированной программе генератор вносит предположение о некотором элементе из области возможных решений, а затем просто проверяется, корректно ли данное предположение генератора.

Для написания программ недетерминированного выбора конкретного элемента из некоторого списка в качестве генератора обычно используют предикат *member* из примера 27, порождающий множество решений. При задании цели *member* (*X, [1,2,3,4]*) будут даны в требуемой последовательности решения *X=1, X=2, X=3, X=4*.

Пример 72: проверить существование в двух списках одного и того же элемента.

```
domains
list=integer*
predicates
member (integer, list)
intersect(list,list)
clauses
member (Head, [Head |_ ]).
member (Head, [_ | Tail ]):- member (Head, Tail).
intersect (L1, L2):- member(X, L1), member(X, L2).
goal
intersect ([1, 4, 3, 2], [2, 5,6]).
```

Первая подцель *member* в предикате *intersect* генерирует элементы из первого списка, а с помощью второй подцели *member* проверяется, входят ли эти элементы во второй список. Описывая данную программу как

недетерминированную, можно говорить, что первая цель делает предположение о том, что *X* содержится в списке *L1*, а вторая цель проверяет, является ли *X* элементом списка *L2*.

Следующее определение предиката *member* с использованием предиката *append:*

member(X, L):- append(L1, [X|L2], L) само по существу является программой, в которой реализован принцип «образовать и проверить». Однако, в данном случае, два шага метода сливаются в один в процессе унификации. С помощью предиката *append* производится расщепление списка и тут же выполняется проверка, является ли *X* первым элементом второго списка.

Еще один пример преимущества соединения генерации и проверки дает программа для решения задачи об *N* ферзях: требуется разместить *N* ферзей на квадратной доске размером *NxN* так, чтобы на каждой горизонтальной, вертикальной или диагональной линии было не больше одной фигуры. В первоначальной формулировке речь шла о размещении *8* ферзей на шахматной доске таким образом, чтобы они не угрожали друг другу. Отсюда пошло название задачи о ферзях.

Эта задача хорошо изучена в математике. Для *N=2* и *N=3* решения не существует; два симметричных решения при *N=4* показаны на рисунке. Для *N=8* существует *88* (а с учетом симметричных – *92*) решений этой задачи.

		Q	
Q			
			Q
	Q		

	Q		
			Q
Q			
		Q	

Приведенная в *примере 73* программа представляет решение задачи об *N ферзях*. Решение задачи представляется в виде некоторой перестановки списка от *1 до N*. Порядковый номер элемента этого списка определяет номер вертикали, а сам элемент – номер горизонтали, на пересечении которых стоит ферзь. Так решение *[2, 4, 1, 3]* задачи о четырех ферзях соответствует первому решению, представленному на рисунке, а решение *[3, 1, 4, 2]*- второму решению. Подобное описание решений и программа их генерации неявно предполагают, что в любом решении задачи о ферзях на каждой горизонтали и на каждой вертикали будет находиться по одному ферзю.

Пример 73: программа решения задачи об N ферзях.

```
domains
list=integer*
predicates
range (integer, integer, list)
/* предикат порождает список, содержащий числа в заданном интервале*/
queens (list, list, list)
/* предикат формирует решение задачи о N ферзях в виде списка решений, при этом первый список – текущий вариант списка размещения ферзей, второй список промежуточное решение, третий список - результат*/
select (integer, list, list)
/*предикат удаляет из списка одно вхождение элемента*/
attack (integer, list)
/*предикат преобразует attack, чтобы ввести начальное присваивание разности в номерах горизонталей */
attack (integer, integer, list)
/*предикат проверяет условие атаки ферзя другими ферзями из списка, два ферзя находятся на одной и той же диагонали, на расстоянии M вертикалей друг от друга, если номер горизонтали одного ферзя на M больше или на M меньше номера горизонтали другого ферзя*/
fqueens (integer, list)
clauses
range (M, N, [M|T]):- M<N, M1=M+1, range (M1, N, T).
range (N, N, [N]):-!.
select(X,[X|T1],T1).
select (X, [Y|T1], [Y|T2]):-select (X, T1, T2).
attack1 (X, L):- attack(X, 1, L).
attack( X, N, [Y|T2]):-N=X-Y; N=Y-X.
attack( X, N, [Y|T2]):-N1=N+1, attack (X, N1, T2).
queens (L1, L2, L3):-select (X, L1, L11),
                     not (attack1 (X,L2)),
                     queens (L11, [X|L2], L3).
queens ([], L, L).
```

fqueens(N,L):-range (1, N, L1),
queens(L1,[],L).
goal
fqueens (4,L),write(L).

При таком задании цели, будет выдано второе решение, представленное на рисунке, если задать внешнюю цель, то будут выданы оба решения.

В данной программе реализован принцип «образоввать и проверить», так как сначала с помощью предиката range генерируется список, содержащий числа от 1 до N. Предикат select перебирает все элементы из полученного списка для размещения очередного ферзя, при этом корректность размещения проверяется при помощи предиката attack. Таким образом, генератором является предикат select, а проверка реализуется при помощи отрицания предиката attack. Чтобы проверить, в безопасном положении находится новый ферзь, необходимо знать позиции ранее размещенных ферзей. В данном случае для хранения промежуточных результатов используется второй параметр предиката queens, так как решение задачи находится на прямом ходе рекурсии, для закрепления результата при выходе из рекурсии используется третий параметр.

3 Основные стратегии решения задач. Поиск решения в пространстве состояний

3.1 Понятие пространства состояния

Пространство состояний – это граф, вершины которого соответствуют ситуациям, встречающимся в задаче («проблемные ситуации»), а решение задачи сводится к поиску путей на этом графе. На самом деле, задача поиска пути на графе и задача о *N* ферзях - это задачи, использующие одну из стратегий перебора альтернатив в пространстве состояний, а именно – стратегию поиска в глубину.

Рассмотрим другие задачи, для решения которых можно использовать в качестве общей схемы решения пространство состояний.

К таким задачам относятся следующие задачи:

задача о восьми ферзях;

переупорядочение кубиков, поставленных друг на друга в виде столбиков;

головоломка «игра в восемь»;

головоломка о «ханойской башне»;

задача о перевозке через реку волка, козы и капусты;

задача о двух кувшинах;

задача о коммивояжере;

другие оптимизационные задачи.

Со всеми задачами такого рода связано два типа понятий:

проблемные ситуации;

разрешенные ходы или действия, преобразующие одни проблемные ситуации в другие.

Проблемные ситуации вместе с возможными ходами образуют направленный граф, называемый пространством состояний.

Пространство состояний некоторой задачи определяет «правила игры»: вершины пространства состояний соответствуют ситуациям, а дуги – разрешенным ходам или действиям, или шагам решения задачи. Конкретная задача определяется:

пространством состояний;

стартовой вершиной;

целевым условием или целевой вершиной.

Каждому разрешенному ходу или действию можно приписать его стоимость. Например, в задаче о коммивояжере ходы соответствуют переездам из города в город, ясно, что стоимость хода в данном случае – это расстояние между соответствующими городами.

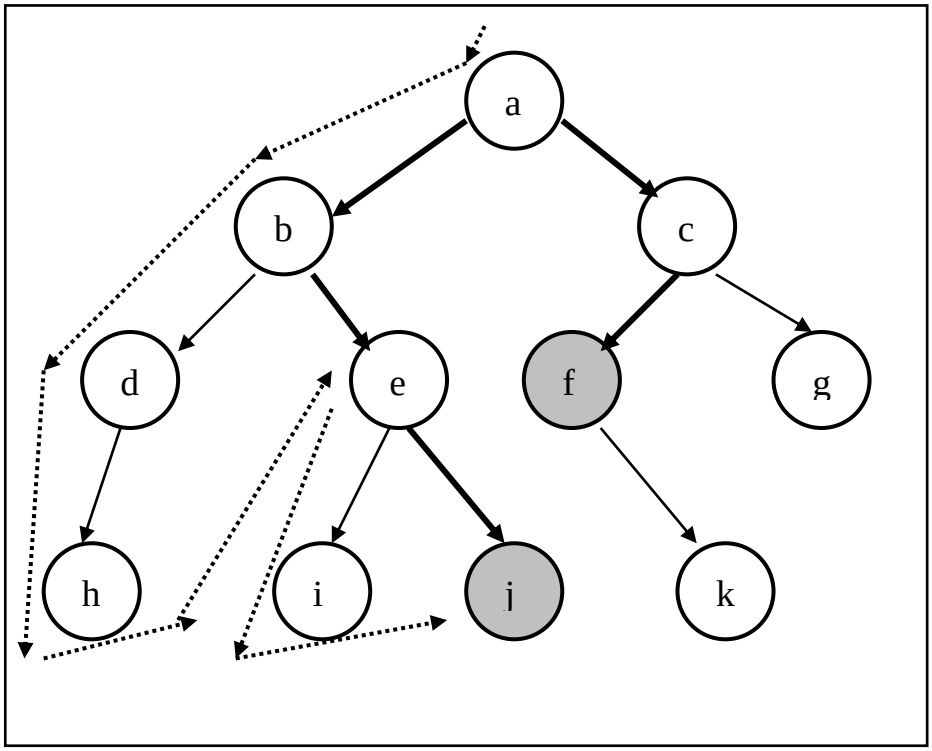
В тех случаях, когда ход имеет стоимость, программист заинтересован в отыскании решения минимальной стоимости. Стоимость решения – это сумма стоимостей дуг, из которых состоит «решающий путь» – путь из стартовой вершины в целевую. Даже если стоимости не заданы, все равно может возникнуть оптимизационная задача: требуется найти кратчайшее решение.

В представленной в примере 73 программе о N ферзях проблемная ситуация (вершина в пространстве состояний) описывается в виде списка из N X-координат ферзей, а переход из одной вершины в другую генерирует предикат *queens*, причем начальная ситуация генерируется предикатом *range*, а целевая ситуация определяется при помощи предиката *attack*.

3.2 Основные стратегии поиска решений в пространстве состояний

3.2.1 Поиск в глубину

Программа решения задачи о N ферзях реализует стратегию поиска в глубину. Под термином «в глубину» имеется в виду тот порядок, в котором рассматриваются альтернативы в пространстве состояний. Всегда, когда алгоритму поиска в глубину надлежит выбрать из нескольких вершин ту, в которую следует перейти для продолжения поиска, он предпочитает самую «глубокую» из них. Самая глубокая вершина – это вершина, расположенная дальше других от стартовой вершины. На следующем рисунке показан пример, который иллюстрирует работу алгоритма поиска в глубину. Этот порядок в точности соответствует результату трассировки процесса вычислений при поиске решения.



Порядок обхода вершин указан пунктирными стрелками, a – начальная вершина, j и f – целевые вершины.

Поиск в глубину наиболее адекватен рекурсивному стилю программирования, принятому в Прологе. Причина этого состоит в том, что обрабатывая цели, Пролог сам просматривает альтернативы именно в глубину.

На Прологе переход от одной проблемной ситуации к другой может быть представлен при помощи предиката *after (X, Y, C),* который истинен тогда, когда в пространстве состояний существует разрешенный ход из вершины *X* в вершину *Y*, стоимость которого равна *C*. Предикат *after* может быть задан в программе явным образом в виде фактов, однако такой принцип оказывается непрактичным, если пространство состояний является достаточно сложным. Поэтому отношение следования *after* обычно определяется неявно, при помощи правил вычисления вершин, следующих за некоторой заданной вершиной.

Другой проблемой при описании пространства состояний является способ представления самих вершин, то есть самих состояний.

В качестве первого примера решения таких задач рассмотрим задачу о ханойских башнях. Есть три стержня и набор дисков разного диаметра. В начале игры все диски надеты на левый стержень. Цель игры заключается в переносе всех дисков на правый стержень по одному стержню за раз, при этом нельзя ставить диск большего диаметра на диск меньшего диаметра. Для этой игры есть простая стратегия:

1. Один диск перемещается непосредственно;
2. *N* дисков переносятся в 3 этапа:

- Перенести *N-1* диск на средний стержень;
- Перенести последний диск на правый стержень;

- Перенести *N-1* диск со среднего на правый стержень.

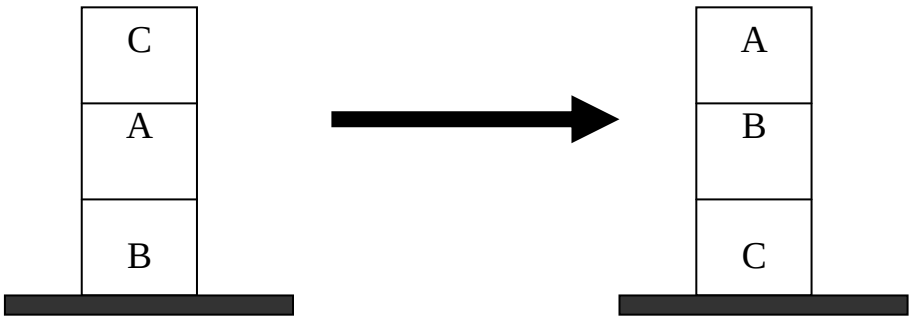
В программе на языке Пролог есть 3 предиката:

- *hanoi* – запускающий предикат, указывает сколько дисков надо переместить;
- *move* – описывает правила перемещения дисков с одного стержня на другой;
- *inform* – указывает на действие с конкретным диском.

Пример 74: решение задачи о ханойских башнях.

```
domains
loc=right;middle;left
% описывает состояние стержней
predicates
hanoi(integert)
% определяет размерность задачи
move(integer,loc,loc,loc)
% определяет переход из одной вершины пространства состояния в другую, то есть описывает правила перекладывания дисков
inform(loc,loc)
% распечатывает действия с дисками
clauses
hanoi(N):-move(N,left,middle,right).
move(1,A,_,C):-inform(A,C),!.
move(N,A,B,C):-N1=N-1, move(N1,A,C,B),
inform(A,C), move(N1,B,A,C).
inform(Loc1,Loc2):-nl, write("Move a disk from ",Loc1," to ", Loc2).
goal
hanoi(3).
```

В качестве второго примера решения таких задач рассмотрим задачу нахождения плана переупорядочивания кубиков, представленную на следующем рисунке.



На каждом шаге разрешается переставлять только один кубик. Кубик можно взять только тогда, когда его верхняя поверхность свободна. Кубик можно поставить либо на стол, либо на другой кубик. Для того, чтобы получить требумое состояние, необходимо получить последовательность

ходов, реализующую данную трансформацию. В качестве примера будет рассмотрен общий случай данной задачи, когда имеется произвольное число кубиков в столбиках. Число столбиков ограничено некоторым максимальным значением.

Проблемную ситуацию можно представить как список столбиков. Каждый столбик, в свою очередь, представляется списком кубиков, из которых он составлен. Кубики упорядочены в списке таким образом, что самый верхний кубик находится в голове списка. «Пустые» столбики изображаются как пустые списки. Таким образом, исходную ситуацию на рисунке можно записать как терм *[[c, a, b], [], []]*.

Целевая ситуация- это любая конфигурация кубиков, содержащая столбик, составленный из имеющихся кубиков в указанном порядке. Таких ситуаций три:

[[a, b, c], [], []];
[[], [a, b, c], []];
[[], [], [a, b, c]].

Пример 75: решение задачи о перемещении кубиков.ОБЪЯСНИТЬ!

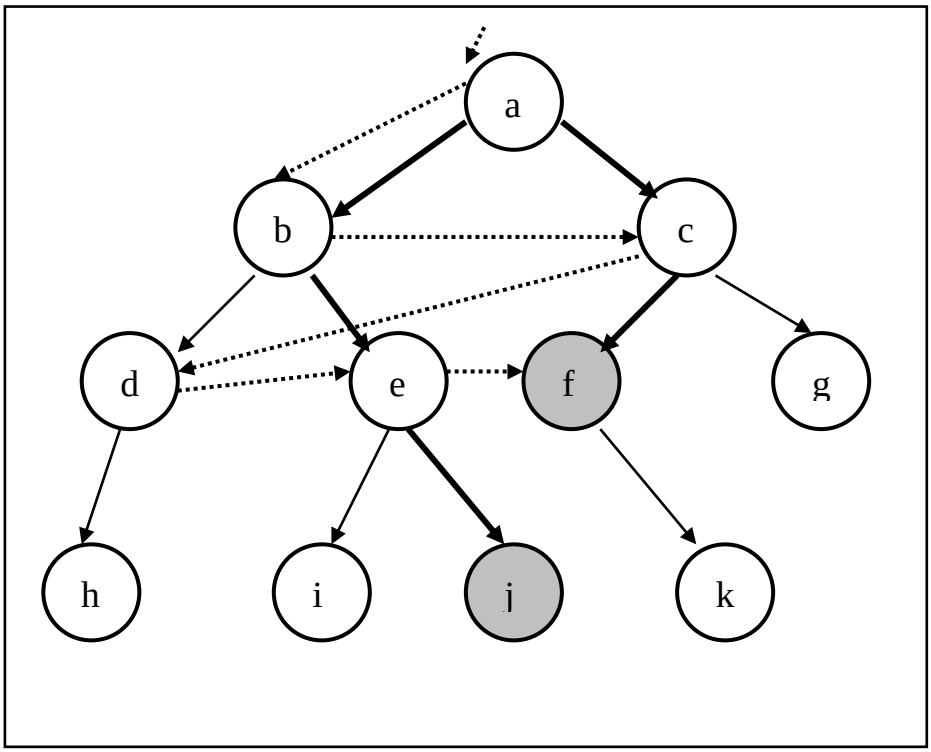
```
domains
list = char*
predicates
inverse(list,list,list);
perest(list,list,list,list,integer)
start(list,list)
clauses
inverse([],L,L).
inverse([H1|T1],P,L) :- inverse(T1,[H1|P],L).
perest([],[],_,_,0):-!.
perest(T1,[H2],T3,[H4|T4],0):-H2=H4,write("\n2-3"),perest(T1,[],[H2|T3],T4,0).
perest([H1|T1],T2,T3,[H4|T4],0):- H1=H4,write("\n1->3"),
                                  perest(T1,T2,[H1|T3],T4,0).
perest(T1,[H2],T3,[H4|T4],0):-H2<>H4,write("\n2->3"),
                                  perest(T1,[],[H2|T3],[H4|T4],1).
perest([H1,H2|T1],[],T3,[H4|T4],0) :- H2=H4,write("\n1->2"),write("\n1->3"),
                                  perest(T1,[H1],[H2|T3],T4,0).
perest([H1|T1],[],T3,[H4|T4],X):-H1<>H4,write("\n1->3"),
                                  X1=X+1,perest(T1,[],[H1|T3],[H4|T4],X1).
perest([H1|T1],[],T3,[H4|T4],X):-%H1=H4,
                                  write("\n1->2"),perest(T1,[H1],T3,[H4|T4],X).
perest(T1,[H1],[H3|T3],T4,X):-H1=H4,write("\n3->1"),X1=X-1,
                                  perest([H3|T1],[H1],T3,T4,X1).
start(L1,L2) :- inverse(L2,[],L3),perest(L1,[],[],L3,0).

goal
start(['a','b','c','d'], ['a','b','c','d']).
```

% start(['a','b','c'], ['b','c','a']).
% start(['a','b','c','d'], ['a','d','c','b']).

3.2.2 Поиск в ширину

В противоположность поиску в глубину стратегия поиска в ширину предусматривает переход в первую очередь к вершинам, ближайшим к начальной вершине. На следующем рисунке показан пример, который иллюстрирует работу алгоритма поиска в ширину.



Поиск в ширину программируется не так легко, как поиск в глубину. Причина состоит в том, что приходится сохранять все множество альтернативных вершин-кандидатов, а не только одну вершину как при поиске в глубину. Более того, если при помощи процесса поиска необходимо получит решающий путь, то следует хранить не множество вершин-кандидатов, а множество путей-кандидатов. Для представления множества путей-кандидатов обычно используют списки, однако при таком способе одинаковые участки путей хранятся в нескольких экземплярах. Избежать подобной ситуации можно, если представить множество путей-кандидатов в виде дерева, в котором общие участки путей хранятся в его верхней части без дублирования. При реализации стратегии поиска в ширину решающие пути порождаются один за другим в порядке увеличения их длин, следовательно, стратегия поиска в ширину гарантирует получение кратчайшего решения первым.

Представленные выше стратегии поиска в глубину и поиска в ширину не учитывают стоимости, приписанной дугам в пространстве состояний. Если критерием оптимальности является минимальная стоимость пути, а не

его длина, то в данном случае поиск в ширину не решает поставленную задачу.

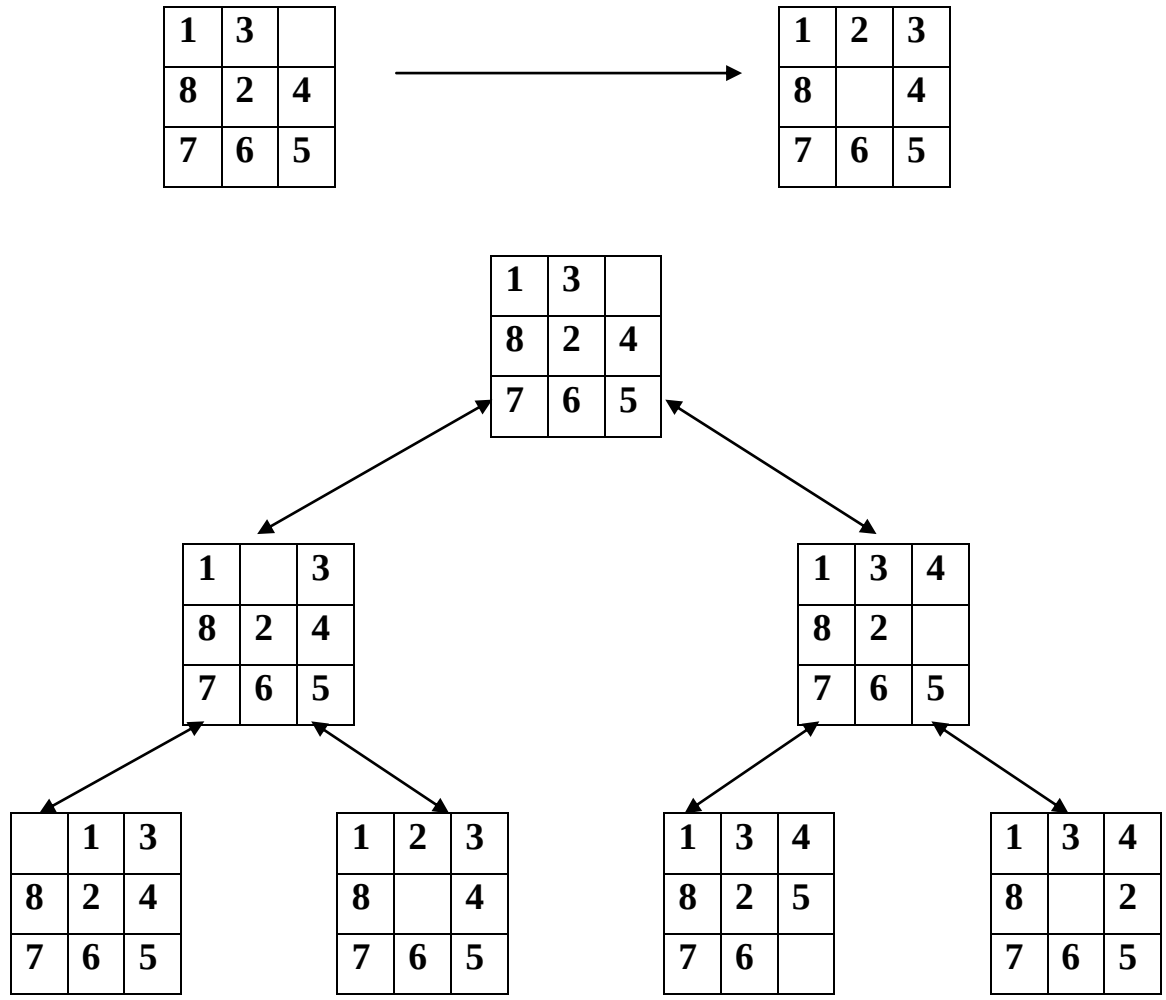
Еще одна проблема, возникающая при решении задачи поиска – это проблема *комбинаторной сложности*. Для сложных предметных областей число альтернатив столь велико, что проблема сложности часто принимает критический характер, так как длина решающего пути (тем более, если их множество, как при реализации поиска в ширину) может привести к *экспоненциальному росту длины в зависимости от размерности задачи*, что приводит к ситуации, называемой *комбинаторным взрывом*. Стратегии поиска в глубину и в ширину недостаточно «умны» для борьбы с такой ситуацией, так как все пути рассматриваются как одинаково перспективные.

По-видимому, процедуры поиска должны использовать какую-либо информацию, отражающую специфику данной задачи, с тем, чтобы на каждой стадии поиска принимать решения о наиболее перспективных путях поиска. В результате процесс будет продвигаться к целевой вершине, обходя бесполезные пути. Информация, относящаяся к конкретной решаемой задаче и используемая для управления поиском, называется *эвристикой*. Алгоритмы поиска, *использующие эвристики*, называют *эвристическими алгоритмами*.

Одним из эвристических алгоритмов решения задач является алгоритм A*, который является усовершенствованным алгоритмом поиска в ширину. Это, так называемый, алгоритм поиска по заданному критерию. Для каждого возможного перехода за один шаг определяется эвристическая оценка и для продолжения поиска решающего пути выбирается наилучшая в соответствии с данной оценкой вершина.

Предполагается, что для всех дуг в пространстве состояний определена функция стоимости перемещения от вершины к вершинам-преемникам. Допустим, для эвристической оценки применяется функция *f(n)* такая, что для каждой вершины *n* она служит для оценки «сложности достижения n». Соответственно наиболее перспективной является та вершина, для которой значение функции *f(n)* является минимальным. Рассмотрим алгоритм A* на примере решения задачи «игра в восемь».

На следующем рисунке представлена задача «игра в восемь» в виде задачи поиска пути в пространстве состояний. В головоломке используется восемь перемещаемых фишек, пронумерованных цифрами от 1 до 8. Фишки располагаются в девяти ячейках, образующих матрицу 3x3. Одна из ячеек всегда пуста, любая смежная с ней фишка может быть передвинута в эту ячейку. Конечная ситуация – это некоторая заранее заданная конфигурация фишек.



При этом можно выделить четыре основных оператора:

1. Перемещение пустой фишки вниз;
2. Перемещение пустой фишки вверх;
3. Перемещение пустой фишки влево;
4. Перемещение пустой фишки вправо.

Оценочная функция *f(n)* формируется как стоимость оптимального пути к цели из начального состояния через *n* вершин дерева поиска. Значение оценочной функции для вершины *n* равно: *f(n)=g(n)+h(n),* где *g(n)* – стоимость оптимального пути от начальной вершины до *n*-ой, а *h(n)* – стоимость оптимального пути от *n*-ой вершины до целевой. Понятно, что *h(n)* это эвристическая гипотеза, основанная на имеющейся информации о данной конкретной задаче.

При этом *g(n)* принимается равной глубине пройденного пути на дереве поиска от начальной вершины до *n*-ой, а *h(n)* – расстояние Хемминга от *n*-ой вершины до целевой (в данном случае оно равно числу фишек, стоящих не на своих местах). Существует модификация алгоритма A*, в которой *h(n)* представляет сумму манхэттеновских расстояний (считается как сумма расстояний в горизонтальном и вертикальном направлениях) от

каждой фишки до её «целевой клетки» плюс утроенное значение «оценки упорядоченности». Оценка упорядоченности определяет степень упорядоченности фишек текущей позиции по отношению к целевой позиции и высчитывается по следующим правилам:

- Фишка в центре имеет оценку 1;
- Фишка в другой позиции имеет оценку 0, если за ней в направлении по часовой стрелке следует соответствующий ей преемник;
- Фишка в другой позиции имеет оценку 2, если за ней в направлении по часовой стрелке не следует соответствующий ей преемник.

Стратегия выбора следующей вершины в пространстве состояний – минимальное значение оценочной функции.

Формулировка алгоритма:

1. Рассматриваем все варианты перемещения пустой фишки за один шаг и выбираем вариант с минимальной оценкой *h(n).*
2. Переходим в новое состояние.
3. Создаём вершины следующего уровня иерархии.
4. Выбираем состояние с минимальной оценкой *h(n).*
5. Повторяем до тех пор, пока не достигнем цели.

Цель не будет достигнута до тех пор, пока число перемещений меньше числа фишек, находящихся не на своих местах.

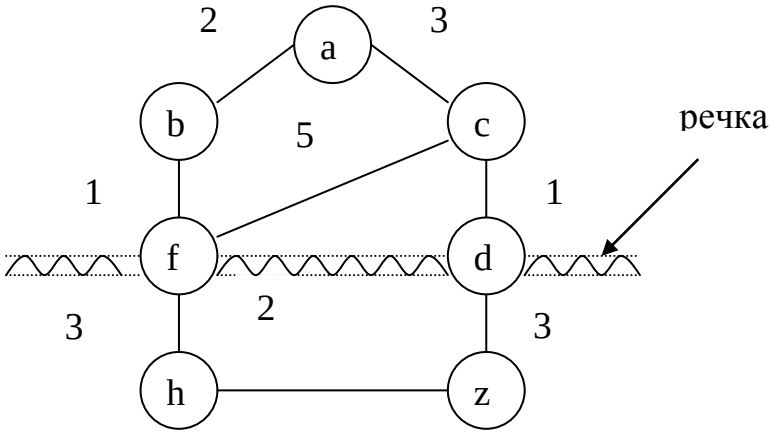
Для задачи, изображённой на рисунке, исходный алгоритм и модифицированный алгоритмы находят решение за два шага.

Для исходного алгоритма оценочные стоимости вершин на первом шаге равны f(1)=n+h(1)=1+1=2 и f(2)=1+3=4, а на втором соотвественно: f(3)=2+2=4 и f(4)=2+0=2. Последняя вершина является целевой.

В соответствии с модифицированным алгоритмом, от начальной вершины возможен переход в два состояния с оценочной стоимостью f(1)=n+h(1)=1+4+3*(1+2)=14 и f(2)=1+3+3*(1+2+2)=19. Выбираем минимальную стоимость и переходим в состояние 1. Далее генерируем вершины 3 и 4 с оценочными стоимостями соответственно f(3)=2+2+3*(1+2+2)=19 и f(4)=2+0=2. Последняя вершина является целевой.

3.3 Сведение задачи к подзадачам и И/ИЛИ графы.

Для некоторых категорий задач более естественным решением является разбиение задачи на подзадачи. Разбиение на подзадачи дает преимущество в том случае, когда подзадачи взаимно независимы, и, следовательно, решать их можно независимо друг от друга. Проиллюстрируем это на примере решения задачи поиска на карте дорог маршрута между заданными городами как показано на рисунке.



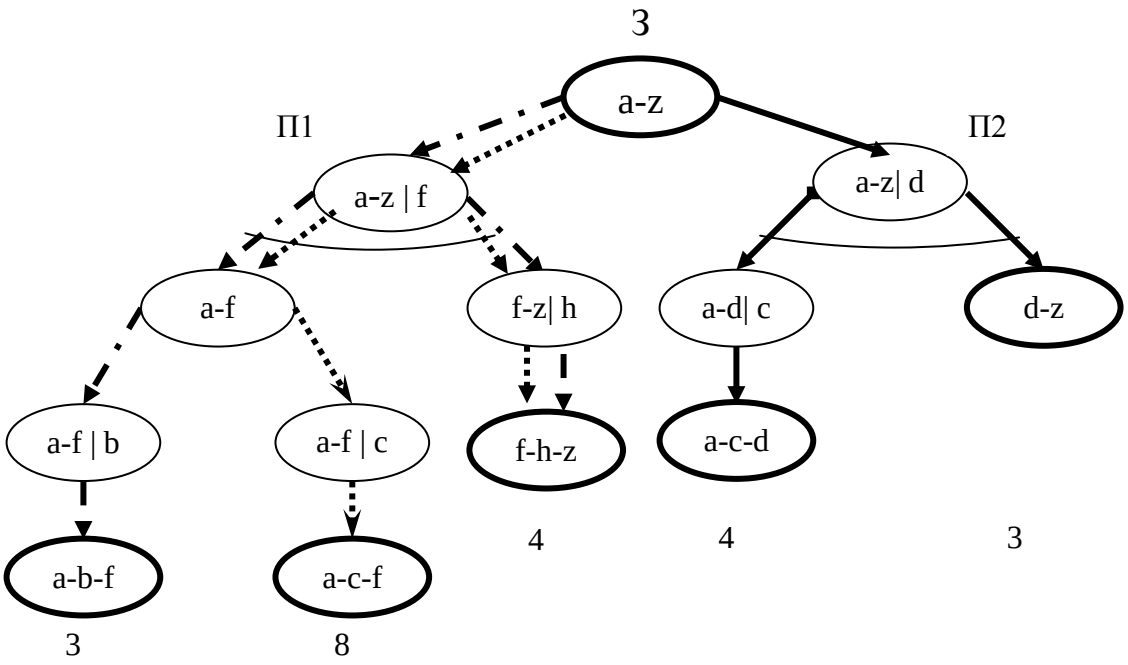
Вершинами a,b,c,d,f,h,z – представлены города. Расстояние между городами обозначено весом дуги из одной вершины графа в другую. На карте есть река. Допустим, что переправиться через реку можно только по двум мостам: один находится в городе f, а второй – в городе d. Очевидно, что искомый маршрут обязательно должен проходить через один из мостов, а значит должен проходить либо через f, либо через d. Таким образом, мы имеем две главные альтернативы:

- путь из a в z, проходящий через f;
- путь из a в z, проходящий через d.

Затем, каждую из этих двух альтернативных задач можно, в свою очередь, разбить следующим образом:

1. для того, чтобы найти путь из a в z через f, необходимо:
- найти путь из a в f и
- найти путь из f в z.
2. для того, чтобы найти путь из a в z через d, необходимо:
- найти путь из a в d и
- найти путь из d в z.

Таким образом, в двух альтернативах мы получили четыре подзадачи, которые можно решать независимо друг от друга. Полученное разбиение исходной задачи можно изобразить в форме И/ИЛИ – графа, представленного на рисунке.



Круглые дуги на графе указывают на отношение **И** между соответствующими подзадачами. Задачи более низкого уровня называются задачами-преемниками.

И/ИЛИ- граф- это направленный граф, вершины которого соответствуют задачам, а дуги – отношениям между задачами.

Между дугами также существуют свои отношения – это отношения И и ИЛИ, в зависимости от того, должны ли мы решить только одну из задач-преемников или же несколько из них. В принципе из вершины могут выходить дуги, находящиеся в отношении И вместе с дугами, находящимися в отношении ИЛИ. Тем не менее, будем предполагать, что каждая вершина имеет либо только И-преемников, либо только ИЛИ-преемников, так как в такую форму можно преобразовать любой И/ИЛИ- граф, вводя в него при необходимости вспомогательные ИЛИ-вершины. Вершину, из которой выходят только И- дуги называются И- вершиной; вершину, из которой выходят только ИЛИ- дуги,- ИЛИ- вершиной.

Решением задачи, представленной в виде И/ИЛИ- графа является решающее дерево, так как решение должно включать в себя все подзадачи И-вершин.

Решающее дерево Т определяется следующим образом:

исходная задача Р – это корень дерева Т;

если Р является ИЛИ-вершиной, то в Т содержится только один из ее преемников (из И/ИЛИ-графа) вместе со своим собственным решающим деревом;

если Р – это И-вершина, то все ее преемники (из И/ИЛИ-графа) вместе со своими решающими деревьями содержатся в Т.

На представленном выше И/ИЛИ- графе представлены три решающих дерева, обозначенных штих-пунктирной, пунктирной и сплошной линиями. Соответственно, стоимости данных деревьев составляют 7, 12, 7. В данном случае стоимости определены как суммы стоимостей всех дуг дерева. Иногда стоимость решающего дерева определяется сумой стоимостей всех его вершин. В соответствии с заданным критерием, из всех решающих деревьев выбирается оптимальное.

3.4 Решение игровых задач в терминах И/ИЛИ- графа

Такие игры, как шахматы или шашки, естественно рассматривать как задачи, представленные И/ИЛИ- графами. Игры такого рода называются играми двух лиц с полной информацией. Будем считать, что существует только два возможных исхода игры:

- выигрыш;
- проигрыш.

Игры с тремя возможными исходами можно свести к играм с двумя исходами, считая, что есть: выигрыш и невыигрыш.

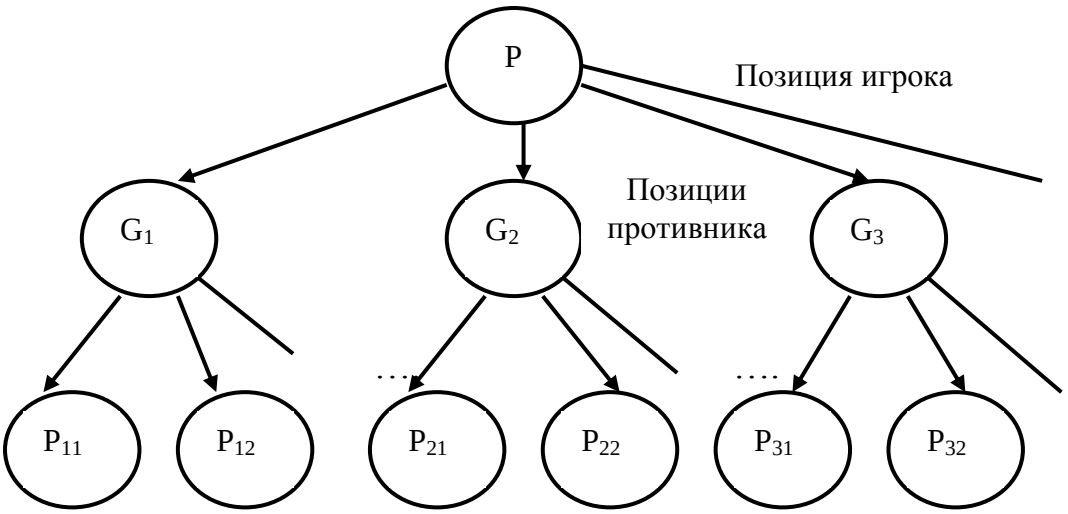
Так как участники игры ходят по очереди, то выделим два вида позиций, в зависимости от того, чей ход:

позиция игрока;

позиция противника.

Допустим, что начальная позиция P – это позиция игрока. Каждый вариант хода игрока в этой позиции приводит к одной из позиций противника G_1, или в G_2, или в G_3 и так далее. Каждый вариант хода противника в позиции G_i приводит к одной из позиций игрока P_{ij}.

В И/ИЛИ- дереве, показанном на рисунке, вершины соответствуют позициям, а дуги – возможным ходам. Уровни позиций игрока чередуются в дереве с уровнями позиций противника. Игрок выигрывает в позиции P, если он выигрывает в G_1, G_2, G_3 и так далее. Следовательно, P – это ИЛИ-вершина. Позиции G_i – это позиции противника, поэтому если в этой позиции выигрывает игрок, то он выигрывает после каждого варианта хода противника, то есть игрок выигрывает в G_i, если он выигрывает во всех позициях P_{ij}. Таким образом, все позиции противника – это И-вершины. Целевые позиции – это позиции, выигранные согласно правилам игры. Для того, чтобы решить игровую задачу, мы должны построить решающее дерево, гарантирующее победу игрока независимо от ответов противника. Такое дерево задает полную стратегию достижения выигрыша: для каждого возможного продолжения, выбранного противником, в дереве стратегии есть ответный ход, приводящий к победе.



Рассмотрим решение подобных задач на примере игры в «2 лунки».

Игрок или его противник может взять из одной любой лунки любое количество камешков. Проигрывает тот, кто берет последний камешек.

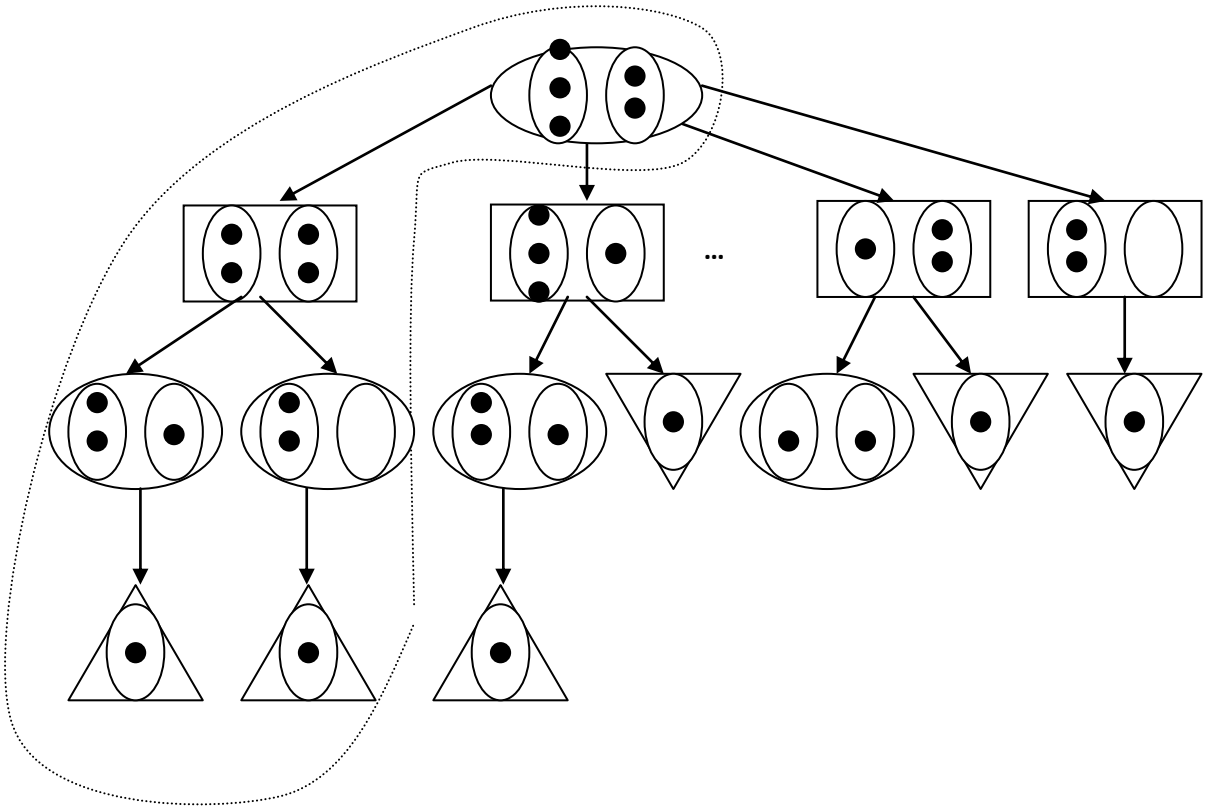
- позиция противника

- позиция игрока

- выигрыш игрока

- проигрыш игрока

Дерево решений этой игры представлено на рисунке.



Пунктирной линией обозначена оптимальная стратегия игрока, которая приведет к выигрышу.

3.5 Минимаксный принцип поиска решений

Алгоритмы поиска пути на И/ИЛИ- графах могут использовать стратегии поиска в глубину и ширину, однако, для большинства игр, дерево игры имеет большое количество позиций, что приводит к комбинаторному взрыву при реализации просмотра всех вершин дерева решений.

Основной подход к организации поиска на игровых деревьях использует оценочные функции. Оценочная функция используется для вычисления оценки текущего состояния игры.

Для выбора следующего хода используется простой алгоритм:

найти всевозможные состояния игры, которые могут быть достигнуты за один ход;

используя оценочную функцию, вычислить оценки состояний;

выбрать ход, ведущий к позиции с наивысшей оценкой.

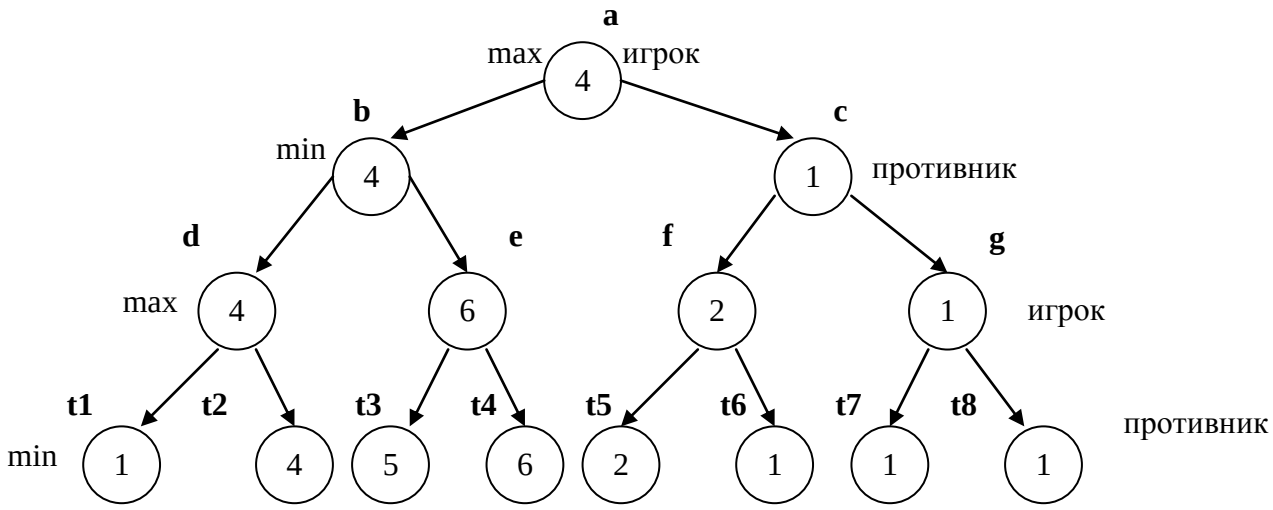
Если оценочная функция была бы совершенной, то есть ее значение отражало бы какие позиции ведут к победе, а какие – к поражению, то достаточно было бы просмотра вперед на один шаг. Обычно совершенная оценочная функция неизвестна, поэтому стратегия выбора хода на основе просмотра на один шаг вперед не дает хорошего результата, поэтому используется стратегия просмотра на несколько шагов вперед.

Стандартный метод определения оценки позиции, основанный на просмотре вперед нескольких слоев игрового дерева, назывыается минимаксным алгоритмом.

Минимаксный алгоритм предполагает, что противник из нескольких возможных ходов сделает выбор, лучший для себя , то есть худший для игрока. Поэтому целью игрока является выбор такого хода, который даст максимальную оценку своей позиции, возможной после лучшего хода противника, то есть минимизирующего оценку позиции противника. Отсюда название – минимаксный алгоритм. Число слоев игрового дерева, просматриваемых при поиске, зависит от доступных ресурсов. На последнем слое используется оценочная функция.

В предположении, что оценочная функция выбрана разумно, алгоритм будет давать тем лучшие результаты, чем больше слоев просматривается при поиске.

Пусть мы имеем следующее дерево игры:



Задана некая оценочная функция $\varphi(P_k)$, где P_k- некоторая игровая ситуация.

Предположим, что игрок максимизирует свой выигрыш, а противник минимизирует свой проигрыш. Вариант решения, образованный минимаксной стратегией движения по дереву игры, будем называть основным вариантом решения.

Если существует оценочная функция, то можно ввести внутреннюю функцию $\varphi(P_k)$ такую, что:

$$\varphi(p_k)=\begin{cases} \max \varphi(p_k) \to p_k - \max \textit{ вершина} \\ \min \varphi(p_k) \to p_k - \min \textit{ вершина} \\ \varphi(p_k) \to p_k - \textit{терминальная вершина} \end{cases}$$

Пример 76:

```
trace
domains
pozic = symbol
spoz = pozic*
facts
xod (pozic, spoz)
xod_min (pozic)
xod_max (pozic)
predicates
minmax (pozic, pozic, integer)
best (spoz, pozic, integer)
oc_term(pozic, integer)
vibor(pozic, integer, pozic, integer, pozic, integer)
clauses
xod (a, [b,c]).
```

```
xod (b, [d,e]).
xod (c, [f,g]).
xod (d, [t1,t2]).
xod (e, [t3,t4]).
xod (f, [t5,t6]).
xod (g, [t7,t8]).
xod_max (a).
xod_max (d).
xod_max (e).
xod_max (f).
xod_max (g).
xod_min (b).
xod_min (c).
xod_min (t1).
xod_min (t2).
xod_min (t3).
xod_min (t4).
xod_min (t5).
xod_min (t6).
xod_min (t7).
xod_min (t8).
oc_term (a,4).
oc_term (b,4).
oc_term (c,1).
oc_term (d,4).
oc_term (e,6).
oc_term (f,2).
oc_term (g,1).
oc_term (t1,1).
oc_term (t2,4).
oc_term (t3,5).
oc_term (t4,6).
oc_term (t5,2).
oc_term (t6,1).
oc_term (t7,1).
oc_term (t8,1).
minmax (Poz, BestPoz, Oc):-
        xod (Poz, SpPoz),!,
        best(SpPoz, BestPoz, Oc);
        oc_term(Poz, Oc).
best ([Poz], Poz, Oc):- minmax (Poz, _ , Oc), !.
best ([Poz1| T], BestPoz, BestOc):-
        minmax (Poz1, _ , Oc1),
        best (T, Poz2, Oc2),
```

```
        vibor(Poz1,Oc1,Poz2,Oc2,BestPoz,BestOc).
vibor(Poz0, Oc0, Poz1, Oc1, Poz0, Oc0):-
        xod_min (Poz0), Oc0>Oc1,!;
        xod_max (Poz0), Oc0<Oc1,!.
vibor(Poz0, Oc0, Poz1, Oc1, Poz1, Oc1).
goal
minmax(a,BestPoz,Oc),write(BestPoz),write(Oc).
```

Литература

1. Адаменко А.Н., Кучуков А. Логическое программирование и Visual Prolog – Спб.: БХВ – Петербург, 2003.
2. Братко И. Алгоритмы искусственного интеллекта на языке Prolog. М.: Вильямс, 2004. – 637 с.
3. Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог: Пер. с англ. М.: Мир.1990.
4. Солдатова О. П., Лёзина И.В. Программирование на языке ПРОЛОГ: метод. указания / О. П. Солдатова, И.В.Лёзина.-Самара: Изд-во Самар. гос. аэрокосм. ун-та, 2008- 52 с.

«Системы искусственного интеллекта»

Методические указания к лабораторным работам по курсу

Содержание

1.Основные положения языка программирования Пролог. 3
1.1 Основы логического программирования 3
1.2 Использование дизъюнкции и отрицания. 6
1.3 Управление поиском решения. 7
1.4 Процедурность Пролога. 7
2.Структура программы на языке Пролог. **8**
3.Использование списков в Прологе **10**
4. Применение списков в программах 12
4.1 Поиск элемента в списке 12
4.2 Объединение двух списков 13
4.3 Сортировка списков 13
4.4 Компоновка данных в список 15
5.Использование составных термов 16
6.Основы работы в Visual Prolog. 23
6.1. Создание TestGoal –проекта для выполнения программ 23
6.2. Запуск и тестирование программы. 24
7.Задания для лабораторных работ. 25
7.1 Задания для лабораторной работы на тему: работа со списками 25
7.2 Задания для лабораторной работы на тему: получение структурированной информации из базы данных. 27
7.3 Задания для лабораторной работы на тему: решение логических головоломок 38
Список рекомендуемой литературы 43

1.Основные положения языка программирования Пролог

1.1 Основы логического программирования

Язык программирования Пролог (PROgramming LOGic) предполагает получение решения задачи при помощи логического вывода из ранее известных фактов. Программа на языке Пролог не является последовательностью действий – она представляет собой набор фактов и правил, обеспечивающих получение логических заключений из данных фактов. Поэтому Пролог считается *декларативным* языком программирования.

Пролог базируется на *фразах (предложениях) Хорна*, являющихся подмножеством формальной системы, называемой *логикой предикатов.*

Пролог использует упрощенную версию синтаксиса логики предикатов, он прост для понимания и очень близок к естественному языку.

Пролог имеет механизм вывода, который основан на сопоставлении образцов. С помощью подбора ответов на запросы Пролог извлекает хранящуюся информацию. Пролог пытается ответить на запрос, запрашивая информацию, о которой уже известно, что она истинна.

Одной из важнейших особенностей Пролога является то, что он ищет не только ответ на поставленный вопрос, но и все возможные альтернативные решения. Вместо обычной работы программы на процедурном языке от начала и до конца, Пролог может возвращаться назад и просматривать все остальные пути при решении всех частей задачи.

Программист на Прологе описывает *объекты* и *отношения*, а также *правила,* при которых эти отношения являются истинными.

Объекты рассуждения в Прологе называются *термами* – синтаксическими объектами одной из следующих категорий:

- константы,
- переменные,
- функции (составные термы или структуры), состоящие из имени функции и списка аргументов-термов, имена функций начинаются со строчной буквы.

Константа в Прологе служит для обозначения имен собственных и начинается *со строчной буквы*.

Переменная в Прологе служит для обозначения объекта на который нельзя сослаться *по имени*.

Пролог не имеет оператора присваивания.

Переменные в Прологе инициализируются при сопоствлении с константами в фактах и правилах.

До инициализации переменная свободна, после присвоения ей значения она становится связанной. Переменная остается связанной только то время, которое необходимо для получения решения по запросу, затем Пролог освобождает ее и ищет другое решение.

Переменные в Прологе предназначены для установления соответствия между термами предикатов, действующих в пределах одной фразы (предложения), а не местом памяти для хранения данных. Переменная начинается с прописной буквы или знаков подчеркивания.

В Прологе программист свободен в выборе имен констант, переменных, функций и предикатов. Исключения составляют резервированные имена и числовые константы. Переменные от констант отличаются первой буквой имени: у констант она строчная, у переменных – заглавная буква или символ подчеркивания.

Область действия имени представляет собой часть программы, где это имя имеет один и тот же смысл:

- для переменной областью действия является предложение (факт, правило или цель), содержащее данную переменную;
- для остальных имен (констант, функций или предикатов) – вся программа.

Специальным знаком «_» обозначается анонимная переменная, которая используется тогда, когда конкретное значение переменной не существенно для данного предложения. Анонимные переменные не отличаются от обычных при поиске соответствий, но не принимают значений и не появляются в ответах. *Различные вхождения знака подчеркивания означают различные анонимные переменные.*

Отношения между объектами в Прологе называются фактами. Факт соответствует фразе Хорна, состящей из одного положительного литерала.

Факт – это простейшая разновидность предложения Пролога.

Любой факт имеет соответствующее значение истинности и определяет отношение между термами.

Факт является простым предикатом, который записывается в виде функционального терма, состоящего из имени отношения и объектов, заключенных в круглые скобки, например:

мать(мария, анна).

отец(иван, анна).

Точка, стоящая после предиката, указывает на то, что рассматриваемое выражение является фактом.

Вторым типом предложений Пролога является вопрос или *цель. Цель* – это средство формулировки задачи, которую должна решать программа. Простой вопрос (цель) синтаксически является разновидностью факта, например:

Цель: мать (мария, юлия).

В данном случае программе задан вопрос, является ли мария матерью юлии. Если необходимо задать вопрос, кто является матерью юлии, то цель будет иметь следующий вид:

Цель: мать(X, юлия).

Сложные цели представляют собой конъюнкцию простых целей и имеют следующий вид:

Цель: Q_1, Q_2,…,Q_n, где запятая обозначает операцию конъюнкции, а Q_1, Q_2,…,Q_n – подцели главной цели.

Конъюнкция в Прологе истинна только при истинности всех компонент, однако, в отличие от логики, в Прологе учитывается *порядок оценки истинности компонент* (слева направо).

Пример 1.

Пусть задана семейная база данных (БД) при помощи перечисления родительских отношений в виде списка фактов:

мать(мария, анна).

мать(мария, юлия).

мать(анна, петр).

отец(иван, анна).

отец(иван, юлия).

Тогда вопрос, является ли иван дедом петра, можно задать в виде следующей цели:

Цель: отец(иван, X), мать(X, петр).

На самом деле БД Пролога включает не только факты, но и правила. Факты и правила представляют собой не множество, а список. Для получения ответа БД просматривается по порядку, то есть в порядке следования фактов и предикатов в тексте программы.

Цель достигнута, если в БД удалось найти факт или правило, который (которое) удовлетворяет предикату цели, то есть превращает его в истинное высказывание. В нашем примере первую подцель удовлетворяют факты *отец(иван, анна). и отец(иван, юлия).* Вторую подцель удовлетворяет факт *мать(анна, петр).* Следовательно, главная цель удовлетворена, переменная *X* связывается с константой *анна*.

Третьим типом предложения является *правило*. Правило позволяет вывести один факт из других фактов. Иными словами, правило – это заключение, для которого известно, что оно истинно, если одно или несколько других найденных заключений или фактов являются истинными.

Правила – это предложения вида

H: - P_1, P_2,…, P_n.

Символ «: -» читается как «если», предикат H называется заключением, а последовательность предикатов P_1, P_2,…, P_n называется посылками. Приведенное правило является аналогом хорновского дизъюнкта $\neg P_1 \vee \neg P_2,\ldots, \vee \neg P_n \vee H$. Заключение истинно, если истинны все посылки. В посылках переменные связаны квантором существования, а в заключении - квантором всеобщности.

Пример 2.

Добавим в БД примера 18 правила, задающие отношение «дед»:

мать(мария, анна).

мать(мария, юлия).

мать(анна, петр).

отец(иван, анна).

отец(иван, юлия).

дед (X, Y): - отец(X, Z), мать(Z, Y).

дед (X, Y): - отец(X, Z), отец(Z, Y).

Тогда вопрос, является ли иван дедом петра, можно задать в виде следующей цели:

Цель: дед(иван, петр).

Правила - самые общие предложения Пролога, факт является частным случаем правила без правой части, а цель – правило без левой части.

Все предложения для одного предиката связаны между собой отношением «или».

Очень часто правила в Прологе являются рекурсивными. Например, для нашей семейной БД предикат «предок» определяется рекурсивно:

предок(x, y): - мать(x, y).

предок(x, y): - отец(x, y).

предок(x, y): - мать(x, z), предок(z, y).

предок(x, y): - отец (x, z), предок(z, y).

Рекурсивное определение предиката обязательно должно содержать нерекурсивную часть, иначе оно будет логически некорректным и программа зациклится. Чтобы избежать зацикливания, следует также позаботиться о порядке выполнения предложений, поэтому практически полезно, а порой и необходимо придерживаться принципа: *«сначала нерекурсивные выражения»*.

Программа на Прологе - это конечное множество предложений.

К комментарию на Прологе относится всё, что находится между знаками /* и */.

/* Это комментарий. */

Либо можно поставить в начале строки знак % и тогда вся строка будет считаться комментарием.

% И это тоже комментарий.

1.2 Использование дизъюнкции и отрицания.

Чистый Пролог разрешает применять в правилах и целях только конъюнкцию, однако, язык, используемый на практике, допускает применение дизъюнкции и отрицания в телах правил и целях. Для достижения цели, содержащей дизъюнкцию, Пролог–система сначала пытается удовлетворить левую часть дизъюнкции, а если это не удается, то переходит к поиску решения для правой части дизъюнкции. Аналогичные действия производятся при выполнении тела правил, содержащих дизъюнкцию. Для обозначения дизъюнкции используется символ « ; ».

В Прологе отрицание имеет имя *«not»* и для представления отрицания какого-либо предиката *P* используется запись *not(P). Цель not(P) достижима тогда и только тогда, когда не удовлетворяется предикат (цель) P.* При этом переменным значения не присваиваются. В самом деле, если достигается *P*, то не достигается *not(P)*, значит надо стереть все

присваивания, приводящие к данному результату. Наоборот, если *P* не достигается, то переменные не принимают никаких значений.

1.3 Управление поиском решения.

Встроенный в Пролог механизм поиска с возвратом может привести к поиску ненужных решений, в результате чего снижается эффективность программы в случае, если надо найти только одно решение. В других случаях бывает необходимо продолжить поиск, даже если решение найдено.

Пролог обеспечивает два встроенных предиката, которые дают возможность управлять механизмом поиска с возвратом: предикат *fail* – используется для инициализации поиска с возвратом и предикат *отсечения !* – используется для запрета возврата.

Предикат fail всегда имеет ложное значение!

Пример 3: Использование предиката fail. Для примера 1 можно добавить правило для печати всех матерей, которые есть в БД:

печать_матерей:-мать(X,Y), write(X,” есть мать”,Y),nl,fail.

goal

печать_матерей.

В результате будет выдано 3 решения:

X=мария, Y= анна.

X=мария, Y= юлия.

X=анна, Y= петр.

Отсечение так же, как и *fail* помещается в тело правила. *Однако, в отличие от fail предикат отсечения имеет всегда истинное значение.*

При этом выполняется обращение к другим предикатам в теле правила, следующим за отсечением. Следует иметь в виду, что невозможно произвести возврат к предикатам, расположенным в теле правила перед отсечением, а также невозможен возврат к другим правилам данного предиката.

Существует только два случая применения предиката отсечения:

1. Если заранее известно, что определенные посылки никогда не приведут к осмысленным решениям – это так называемое «зеленое отсечение».
2. Если отсечения требует сама логика программы для исключения альтернативных подцелей – это так называемое «красное отсечение».

1.4 Процедурность Пролога.

Пролог – декларативный язык. Описывая задачу в терминах фактов и правил, программист предоставляет Прологу самому искать способ решения. В процедурных языках программист должен сам писать процедуры и функции, которые подробно «объясняют» компьютеру, какие шаги надо сделать для решения задачи.

Тем не менее, рассмотрим Пролог с точки зрения процедурного программирования:

1. Факты и правила можно рассматривать как определения процедур.
2. Использование правил для условного ветвления программы. Правило, в отличие от процедуры, позволяет задавать множество альтернативных определений одной и той же процедуры. Поэтому, правило можно считать аналогом оператора *case* в Паскале.
3. В правиле может быть выполнено сравнение, как в условных операторах.
4. Отсечение можно считать аналогом *go to.*
5. Возврат вычисленного значения производится аналогично процедурам. В Прологе это делается путем связывания свободных переменных при сопоставлении цели с фактами и правилами.

2.Структура программы на языке Пролог.

Программа, написанная на Прологе, состоит из пяти основных разделов: раздел описания доменов, раздел базы данных, раздел описания предикатов, раздел описания предложений и раздел описания цели. Ключевые слова *domains, constants, database (facts), predicates, clauses и goal* отмечают начала соответствующих разделов. Назначение этих разделов таково:

- раздел *domains* содержит определения доменов, которые описывают различные типы данных, используемых в программе;
- раздел *constants* используется для объявления символических констант, используемых в программе;
- раздел *database (facts)* содержит описания предикатов внутренней базы данных Пролога, если программа такой базы данных не требует, то этот раздел может быть опущен;
- раздел *predicates* служит для описания предикатов, не принадлежащих внутренней базе данных;
- в раздел *clauses* заносятся факты и правила самой программы;
- в разделе *goal* на языке Пролог формулируется назначение создаваемой программы. Составными частями при этом могут являться некие подцели, из которых формируется единая цель программы.

В Visual Prolog разрешает объявление разделов domains, facts, predicates, clauses как глобальных разделов, то есть с ключевым словом global.

Пролог имеет следующие встроенные типы доменов:

Тип данных	Ключевое слово	Диапазон значений	Примеры использования
Символы	char	Все возможные символы	‘a’, ’b’, ‘#’, ‘B’, ‘%’
Целые числа	integer byte word dword	От –32768 до 32767 От 0 до 255 От 0 до 65535 От 0 до 2^{32}	-63, 84, 2349
Действительные числа	real short ushort long ulong unsigned	От +1E-307 до +1E308 16 битов со знаком 16 битов без знака 32 бита со знаком 32 бита без знака 16 или 32 бита без знака	360, - 8324, 1.25E23, 5.15E-9
Строки	string	Последовательность символов (не более 250)	«today», «123», «school_day»
Символические имена	symbol	1.Последовательность букв, цифр, символов подчеркивания; первый символ – строчная буква. 2. Последовательность любых символов, заключенная в кавычки.	flower, school_day «string and symbol»
Ссылочный тип	ref		
Файлы	file	Допустимое в DOS имя файла	mail.txt, LAB.PRO

Если в программе необходимо использовать новые домены данных, то они должны быть описаны в разделе *domains*.

Пример 4:

```
domains
number=integer
name, person=symbol.
```

Различие между *symbol* и *string* - в машинном представлении и выполнении, синтаксически они не различимы.

Visual Prolog выполняет автоматическое преобразование типов между доменами *string* и *symbol*. Однако, по принятому соглашению, символическую строку в двойных кавычках нужно рассматривать как *string,* а без кавычек – как *symbol:*

Symbol - имена, начинающиеся с символа нижнего регистра и содержащие только символы, цифры, и символы подчеркивания.

String – в двойных кавычках могут содержать любую комбинацию символов, кроме #0, который отмечает конец строки.

Visual Prolog поддерживает и другие типы стандартных доменов данных, например, для работы с внешними БД или объектами.

Предикаты описываются в разделе *predicates*. Предикат представляет собой строку символов, первым из которых является строчная буква. Предикаты могут не иметь аргументов, например «go» или «repeat». Если предикаты имеют аргументы, то они определяются при описании предикатов в разделе predicates:

Пример 5:
predicates
mother (symbol, symbol)
father (symbol, symbol).

Факты и правила определяются в разделе *clauses,* а вопрос к программе задается в разделе *goal* – в этом случае цель называется *внутренней целью*. Программа на Турбо-Прологе может не содержать раздел *goal,* в этом случае цель задается в процессе работы программы и является *внешней целью*. Для задания внешней цели в окне *Dialog* будет выведено приглашение *Goal.* После удовлетворения внешней цели программа на Турбо-Прологе не заканчивает свою работу, а просит ввести следующую цель, таким образом можно задать программе несколько различных целей. Если цель в программе является внутренней целью, то процесс вычисления остановится после первого ее успешного вычисления. Если цель в программе является внешней целью, то процесс вычисления цели повторяется до тех пор, пока не будут найдены все успешные способы вычисления цели.

В Visual Prolog раздел goal в тексте программы является обязательным. Разница в режимах исполнения программы состоит в разном использовании утилиты Test Goal. Если утилита создается для запуска любой программы, то при этом ищутся все решения, если утилита создается для автономного запуска программы – то ищется одно решение.

3.Использование списков в Прологе

Список – это упорядоченный набор объектов одного и того же типа. Элементами списка могут быть целые числа, действительные числа, символы, строки, символические имена и структуры. Порядок расположения элементов в списке играет важную роль: те же самые элементы списка, упорядоченные иным способом, представляют уже совсем другой список.

Совокупность элементов списка заключается в квадратные скобки ([]), элементы друг от друга отделяются запятыми. Список может содержать произвольное число элементов, единственным ограничением является объем оперативной памяти. Количество элементов в списке называется его длиной. Список может содержать один элемент и даже не содержать ни одного элемента. Список, не содержащий элементов, называется пустым или нулевым списком.

Непустой список можно рассматривать как список, состоящий из двух частей: головы – первого элемента списка; и хвоста – остальной части списка. Голова является элементом списка, хвост является списком. Голова списка – это неделимое значение, хвост представляет собой список, составленный из того, что осталось от исходного списка в результате «отделения головы». Этот новый список обычно можно делить и дальше. Если список состоит из одного элемента, то его можно разделить на голову, которой будет этот самый элемент, и хвост, являющийся пустым списком. Пустой список нельзя разделить на голову и хвост.

Операция деления списка на голову и хвост обозначается при помощи вертикальной черты (|):

[Head | Tail].

Head здесь является переменной для обозначения головы списка, переменная Tail обозначает хвост списка (для имен головы и хвоста списка пригодны любые допустимые Прологом имена). Данная операция также присоединяет элемент в начало списка, например, для того , чтобы присоединить X к списку S следует написать [X | S].

Отличительной особенностью описания списков является наличие звездочки (*) после имени домена элементов.

Пример 6: объявление списков, состоящих из элементов стандартных типов доменов или типа структуры.

```
domains
list1=integer*
list2=char*
list3=string*
list4=real*
list5=symbol*
personal_library = book (title, author, publisher, year)
list6= personal_library*
list7=list1*
list8=list5*
```

В первых пяти объявлениях списков в качестве элементов используются стандартные домены данных, в шестом типе списка в качестве элемента используется домен структуры personal_library, в седьмом и восьмом типе списка в качестве элемента используется ранее объявленный список.

Пример 7: демонстрация разделения списков на голову и хвост.

Список	*Голова*	*Хвост*
[1, 2, 3, 4, 5]	*1*	*[2, 3, 4, 5]*
[6.9, 4.3]	*6.9*	*[4.3]*
[cat, dog, horse]	*Cat*	*[dog, horse]*
[‘S’, ‘K’, ‘Y’]	*‘S’*	*[‘K’, ‘Y’]*
[«PIG»]	*«PIG»*	*[]*
[]	*Не определена*	*Не определен*

4. Применение списков в программах

4.1 Поиск элемента в списке

Для применения списков в программах на Прологе необходимо описать домен списка в разделе *domains*, предикаты, работающие со списками необходимо описать в разделе *predicates*, задать сам список можно либо в разделе *clauses* либо в разделе *goal*.

Над списками можно реализовать различные операции: поиск элемента в списке, разделение списка на два списка, присоединение одного списка к другому, удаление элементов из списка, сортировку списка, создание списка из содержимого БД и так далее.

Поиск элемента в списке является очень распространенной операцией. Поиск представляет собой просмотр списка на предмет выявления соответствия между объектом поиска и элементом списка. Если такое соответствие найдено, то поиск заканчивается успехом, в противном случае поиск заканчивается неуспехом. Стратегия поиска при этом будет состоять в рекурсивном выделении головы списка и сравнении ее с объектом поиска.

Пример 8: поиск элемента в списке.

```
domains
list=integer*
predicates
member (integer, list)
clauses
member (Head, [Head | _ ]).
member (Head, [_ | Tail ]):- member (Head, Tail).
goal
member (3, [1, 4, 3, 2]).
```

Правило поиска может сравнить объект поиска и голову текущего списка, эта операция записана в виде факта предиката member. Этот вариант предполагает наличие соответствия между объектом поиска и головой списка. Отметим, что хвост списка в данном случае не важен, поэтому хвост списка присваивается анонимной переменной. Если объект поиска и голова списка различны, то в результате исполнения первого предложения будет неуспех, происходит возврат и поиск другого правила или факта, с которыми можно попытаться найти соответствие. Для этой цели служит второе предложение, которое выделяет из списка следующий по порядку элемент, то есть выделяет голову текущего хвоста, поэтому текущий хвост представляется как новый список, голову которого можно сравнить с объектом поиска. В случае исполнения второго предложения, голова текущего списка ставится в соответствие анонимной переменной, так как значение головы с писка в данном случае не играет никакой роли.

Процесс повторяется до тех пор, пока первое предложение даст успех, в случае установления соответствия, либо неуспех, в случае исчерпания списка. В представленном примере предикат *find* находит все совпадения

объекта поиска с элементами списка. Для того, чтобы найти только первое совпадение следует модифицировать первое предложение следующим образом:

member (Head, [Head |_]):- !.

Отсечение отменяет действие механизма возврата, поэтому поиск альтернативных успешных решений реализован не будет.

4.2 Объединение двух списков

Слияние двух списков и получение, таким образом, третьего списка принадлежит к числу наиболее полезных при работе со списками операций. Обозначим первый список L1, а второй список - L2. Пусть L1 = [1, 2, 3], а L2 = [4, 5]. Предикат append присоединяет L2 к L1 и создает выходной список L3, в который он должен переслать все элементы L1 и L2. Весь процесс можно представить следующим образом:

1. Список L3 вначале пуст.
2. Элементы списка L1 пересылаются в L3, теперь значением L3 будет [1, 2, 3].
3. Элементы списка L2 пересылаются в L3, в результате чего тот принимает значение [1, 2, 3, 4, 5].

Тогда программа на языке Пролог имеет следующий вид:

Пример 9: объединение двух списков.

```
domains
list=integer*
predicates
append (list, list, list)
clauses
append ( [], L2, L2).
append ([H|T1], L2, [H|T3 ]):- append (T1, L2, T3).
goal
append ( [1, 2, 3], [4, 5], L3).
```

Основное использование предиката *append* состоит в объединении двух списков, что делается при помощи задания цели вида *append ([1, 2, 3], [4, 5], L3)*. Поиск ответа на вопрос типа: *append (L1, [3, 4, 5], [1, 2, 3, 4, 5])* – сводится к поиску такого списка *L1=[1, 2]*, который при слиянии со списком *L2 = [3, 4, 5]* даст список *L3 = [1, 2, 3, 4, 5]*. При обработки *цели append (L1, L2, [1, 2, 3, 4, 5])* ищутся такие списки *L1* и *L2*, что их объединение даст список *L3 = [1, 2, 3, 4, 5]*.

4.3 Сортировка списков

Сортировка представляет собой переупорядочение элементов списка определенным образом. Назначением сортировки является упрощение доступа к нужным элементам. Для сортировки списков обычно применяются три метода:

- метод перестановки,

- метод вставки,
- метод выборки.

Также можно использовать комбинации указанных методов.

Первый метод сортировки заключается в перестановке элементов списка до тех пор, пока он не будет упорядочен. Второй метод осуществляется при помощи неоднократной вставки элементов в список до тех пор, пока он не будет упорядочен. Третий метод включает в себя многократную выборку и перемещение элементов списка.

Второй из методов, метод вставки, особенно удобен для реализации на Прологе.

Пример 10: сортировка списков методом вставки.

```
domains
list=integer*
predicates
insert_sort (list, list)
insert (integer, list, list)
asc_order (integer, integer)
clauses
insert_sort ( [], []).
insert_sort ([H1|T1], L2):- insert_sort (T1, T2),
                                    insert(H1, T2, L2).
insert (X, [H1| T1], [H1| T2]) :- asc_order (X, H1), !,
                                  insert (X, T1, T2).
insert (X, L1, [X| L1]).
asc_order (X, Y):- X>Y.
goal
insert_sort ([4, 7, 3, 9], L).
```

Для удовлетворения первого правила оба списка должны быть пустыми. Для того, чтобы достичь этого состояния, по второму правилу происходит рекурсивный вызов предиката *insert_sort,* при этом значениями *H1* последовательно становятся все элементы исходного списка, которые затем помещаются в стек. В результате исходный список становится пустым и по первому правилу выходной список также становится пустым.

После того, как произошло успешное завершение первого правила, Пролог пытается выполнить второй предикат *insert,* вызов которого содержится в теле второго правила. Переменной *H1* сначала присваивается первое взятое из стека значение *9,* а предикат принимает вид *insert (9, [], [9]).*

Так как теперь удовлетворено все второе правило, то происходит возврат на один шаг рекурсии в предикате *insert_sort.* Из стека извлекается 3 и по третьему правилу вызывается предикат *asc_order,* то есть происходит попытка удовлетворения пятого правила *asc_order (3, 9):- 3>9.* Так как данное правило заканчивается неуспешно, то неуспешно заканчивается и третье правило, следовательно, удовлетворяетсячетвертое правило и *3* вставляется в выходной список слева от *9*: *insert (3, [9], [3, 9]).*

Далее происходит возврат к предикату *insert_sort*, который принимает следующий вид: *insert_sort ([3, 9], [3, 9]).*

На следующем шаге рекурсии из стека извлекается 7 и по третьему правилу вызывается предикат *asc_order* в виде *asc_order (7, 3):- 7>3.* Так как данное правило заканчивается успешно, то элемент 3 убирается в стек и *insert* вызвается рекурсивно еще раз, но уже с хвостом списка – *[9]: insert (7, [9], _).* Так как правило *asc_order (7, 9):- 7>9* заканчивается неуспешно, то выполняется четвертое правило, происходит возврат на предыдущие шаги рекурсии сначала *insert,* затем *insert_sort.*

В результате *7* помещается в выходной список между элементами *3* и *9*: *insert (7, [3, 9], [3, 7, 9]).*

При возврате еще на один шаг рекурсии из стека извлекается 4 и по третьему правилу вызывается предикат *asc_order* в виде *asc_order (4, 3):- 4>3.* Так как данное правило заканчивается успешно, то элемент 3 убирается в стек и *insert* вызвается рекурсивно еще раз, но уже с хвостом списка – *[7, 9]: insert (4, [7, 9], _).* Так как правило *asc_order (4, 7):- 4>7* заканчивается неуспешно, то выполняется четвертое правило, происходит возврат на предыдущие шаги рекурсии сначала *insert,* затем *insert_sort.*

В результате *4* помещается в выходной список между элементами *3* и *7*:

insert (4, [3, 7, 9], [3, 4, 7, 9]).
insert_sort [4, 7, 3, 9], [3, 4, 7, 9]).

4.4 Компоновка данных в список

Иногда при программировании определенных задач возникает необходимость собрать данные из фактов БД в список для последующей их обработки. Пролог содержит встроенный предикат *findall*, который позволяет выполнить данную операцию. Описание предиката *findall* выглядит следующим образом:

Findall (Var_, Predicate_, List_), где Var_ обозначает имя для терма предиката Predicate_, в соответствии с типом которого формируются элементы списка List_.

Пример 11: использование предиката findall.

```
domains
d=integer
predicates
decimal (d)
write_decimal
clauses
decimal (0)
decimal (1)
decimal (2)
decimal (3)
decimal (4)
```

decimal (5)
decimal (6)
decimal (7)
decimal (8)
decimal (9)
write_decimal:- findall(C, decimal (C), L), write (L).
goal
write_decimal.

5.Использование составных термов

В Прологе функциональный терм или предикат можно рассматривать как структуру данных, подобную записи в языке Паскаль. Терм, представляющий совокупность термов, называется составным термом. Предикаты, записанные в виде составного терма, называются составной структурой данных. Составные структуры данных в Турбо-Прологе объявляются в разделе *domains.* Если термы структуры относятся к одному и тому же типу доменов, то этот объект называется *однодоменной структурой данных*. Если термы структуры относятся к разным типам доменов, то такая структура данных называется *многодоменной структурой данных*. Использование доменной структуры упрощает структуру предиката.

Пример 12: Необходимо создать БД, содержащую сведения о книгах из личной библиотеки. Зададим составной терм с именем personal_library, имеющим следующую структуру: personal_library= book (title, author, publisher, year), и предикат collection (collector, personal_library). Терм book называется функтором структуры данных. Пример программы, использующей составные термы для описания личной библиотеки и поиска информации о книгах, напечатанных в 1990 году, выглядит следующим образом:

domains
collector, title, author, publisher = symbol
year = integer
personal_library = book (title, author, publisher, year)
predicates
collection (collector, personal_library)
clauses
collection (irina, book («Using Turbo Prolog», «Yin with Solomon», »Moscow, World», 1993)).
collection (petr, book («The art of Prolog», «Sterling with Shapiro», »Moscow, World», 1990)).
collection (anna, book («Prolog: a relation language and its applications», «John Malpas», »Moscow, Science», 1990)).
goal
collection (X, book(Y,_, _, 1990)

Представление данных часто требует наличия большого числа структур. В Прологе эти структуры должны быть описаны. Для более

компактного описания структур данных в Прологе предлагается использование альтернативных описаний доменов.

Пример 13: Необходимо создать БД, содержащую сведения о книгах и аудиозаписях из личной библиотеки.

```
domains
person, title, author, artist, album, type = symbol
thing = book (title, author); record (artist, album, type)
predicates
owns (person, thing)
clauses
owns (irina, book («Using Turbo Prolog», «Yin with Solomon»)).
owns (petr, book («The art of Prolog», «Sterling with Shapiro»)).
owns (anna, book («Prolog: a relation language and its applications», «John Malpas»)).
owns (irina, record («Elton John», «Ice Fair», «popular»)).
owns (petr, record («Benny Goodman», «The King of Swing», »jazz»)).
owns (anna record («Madonna», «Madonna», «popular»»)).
goal
owns (X, record(_, _, «jazz»)
```

Пример 14:

Создать базу данных о заданной предметной области в виде множества фактов языка Пролог (не менее 5 фактов). Информацию о каждом компоненте БД представить в виде структуры. Разработать набор предикатов, осуществляющих взаимодействие с БД, при помощи которых можно реализовать все типы запросов, приведенные в варианте задания.

Предметная область – база данных товаров мебельного магазина. Каждый магазин может быть описан структурой: название магазина, адрес, фамилия директора, список имеющихся товаров. Каждый товар может быть описан структурой: название товара, цена, страна-производитель, список имеющихся расцветок.

Реализовать следующие запросы:

1. *Найти адрес магазина, у которого директором является человек с заданной фамилией.*
2. *Найти названия всех магазинов, продающих заданный товар.*
3. *Найти название товара, имеющего максимальную цену.*

В приведенном ниже примере используются следующие домены:

list_colours - список расцветок, *list_products* - список имеющихся товаров, *name_shop* - название магазина, *address* –адрес магазина, *chief*, - фамилия директора, *name_product* – название товара, *country* - страна-производитель, *price* – цена.

shop – предикат, описывающий заданную предметную область, *q1*,*q2*, *q3* – предикаты, реализующие первый, второй и третий запрос

соответственно, *q21, q31*–вспомогательные предикаты для второго и третьего запросов, *max* – динамический предикат, необходимый для третьего запроса.

```
domains
        list_colours = symbol*
        list_products = product*
        name_shop, address, chief, name_product, country = symbol
        price =real
        product = prod (name_product, price, country, list_colours)
facts
        max(name_product, price)
predicates
        shop (name_shop, address, chief, list_products)
        q1(chief)
        q2(name_product)
        q21(name_product, list_products)
        q3
        q31(list_products)
clauses
shop("Company1","Debenko_5","Ivanov",[prod(table,3000,"Italy",[white,
red, black]),prod(chair,4000,"Japan",[brown, grey])]).
shop("Company2","S.Lazo_4","Petrov",[prod(sofa,9000,"China",[brown,
black]),prod(chair,5000,"England",[red, grey])]).
shop("Company3","Sadovaya_3","Sidorov",[prod(sofa,19000,"Mexico",
[black, white])]).
shop("Company4","Dachnaya_2","Orlov",[prod(bookcase,7000,"Russia",
[brown, black]),prod(table,6000,"Spain",[red,grey])]).
shop("Company5","Pobeda_1","Galkin",[prod(bed,5000,"China",[brown,
black]),prod(arm_chair,21000,"Japan",[red, grey])]).

% первый запрос: обращаемся к предикату shop, проверяем, совпадает
ли фамилия директора данного магазина с заданной фамилией, для этого
используется одна и та же переменная F, в случае совпадения печатаем
адрес магазина A.
q1(F):-shop (_,A,F,_),write(A),nl.

% второй запрос: обращаемся к предикату shop, извлекаем название
магазина N и список имеющихся товаров L, вызываем вспомогательный
предикат q21, которому передаем название заданного товара P и список
имеющихся товаров L, q21 будет истинен, если заданный товар есть в
списке имеющихся товаров, в этом случае выводим на экран название
магазина.
q2(P):-shop(N,_,_,L),q21(P,L),write(N),nl,fail.
q21(H,[H1|_]):-H1=prod(H,_,_,_).
q21(H,[_|T]):-q21(H,T).
```

% третий запрос: записываем в динамический предикат max любое название товара, например, a и его цену 0, так как мы ищем название товара, имеющего максимальную цену, соответственно первоначальная цена должна быть маленькая, чтобы обязательно был товар с более высокой ценой, обращаемся к предикату shop, извлекаем список имеющихся товаров L, вызываем вспомогательный предикат q31, которому передаем список имеющихся товаров L, q31 проверяет, имеет ли очередной товар из списка имеющихся товаров цену, более высокую, чем цена, записанная в предикате max, в этом случае предикатом retract удаляем старые название товара и цену, предикатом assert записываем новое название товара и цену, иначе просто переходим к следующему товару.

```
max(a,0).
q3:-shop(_,_,_,L),q31(L),fail.
q31([]).
q31([H1|T]):-H1=prod(H,P,_,_), max(A,B), P>B, retract(max(A,B)), assert(max(H,P)), q31(T).
q31([H1|T]):-H1=prod(_,P,_,_),max(_,B),P<=B,q31(T).
goal
    % вызываем каждый запрос по очереди.
    q1("Petrov").
    %q2(table).
    %q3;max(A,B).
```

Пример 15: решить предыдущую задачу с использованием внутренней базы данных. Найденные решения записать в виде фактов внутренней базы данных Пролога. Предусмотреть проверку факта, являющегося ответом на запрос в БД. Если такой факт существует, то выдать его в качестве ответа на запрос. Если такого факта не существует во внутренней базе данных, то запустить запрос на выполнение и записать результат в БД.

```
domains
    list_colours = symbol*
    list_products = product*
    name_shop,address,chief,name_product,country = symbol
    price =real
    product = prod (name_product,price,country,list_colours)
facts
    max(name_product,price)
    % динамические предикаты для второго и третьего запроса
    dq1(address)
    dq2(name_shop)
predicates
    shop (name_shop,address,chief,list_products)
    q1(chief)
    q2(name_product)
```

```
        q21(name_product,list_products)
        q3
        q31(list_products)
    clauses
    shop("Company1","Debenko_5","Ivanov",[prod(table,3000,"Italy",[white,
    red,black]),prod(chair,4000,"Japan",[brown,grey])]).
    shop("Company2","S.Lazo_4","Petrov",[prod(sofa,9000,"China",[brown,
    black]),prod(chair,5000,"England",[red,grey])]).
    shop("Company3","Sadovaya_3","Sidorov",[prod(sofa,19000,"Mexico",
    [black,white])]).
    shop("Company4","Dachnaya_2","Orlov",[prod(bookcase,7000,"Russia",
    [brown,black]),prod(table,6000,"Spain",[red,grey])]).
    shop("Company5","Pobeda_1","Galkin",[prod(bed,5000,"China",[brown,
    black]),prod(arm_chair,21000,"Japan",[red,grey])]).

    % проверяем наличие факта, являющегося ответом на запрос в БД,
если такой факт существует, то выдаём его в качестве ответа на запрос,
печатаем перед ним "*".
    q1(_):-dq1(A), write("*",A),nl.
    % Если такого факта не существует во внутренней базе данных, то
запускаем запрос на выполнение и записываем  результат в БД.
    q1(F):-not(dq1(_)), shop (_,A,F,_),write(A),nl,assert(dq1(A)).

    q2(_):-dq2(A), write("*",A),nl,fail.
    q2(P):-not(dq2(_)), shop(N,_,_,L),q21(P,L),write(N),nl,assert(dq2(N)),fail.
    q21(H,[H1|_]):-H1=prod(H,_,_,_).
    q21(H,[_|T]):-q21(H,T).

    max(a,0).
    q3:-shop(_,_,_,L),q31(L),fail.
    q31([]).
    q31([H1|T]):-
H1=prod(H,P,_,_),max(A,B),P>B,retract(max(A,B)),assert(max(H,P)),q31(T).
    q31([H1|T]):-H1=prod(_,P,_,_),max(_,B),P<=B,q31(T).
    goal
        % Запускаем каждый запрос на исполнение два раза - первый раз
        программа находит ответ на запрос, а второй раз выдает ответ из
        БД.
        q1("Petrov"),q1("Petrov").
        %q2(table);q2(table).
        %q3;max(A,B).
```

Пример 16: Решить логическую головоломку. В пяти домах, окрашенных в разные цвета, обитают мужчины разных национальностей. Они держат разных животных, предпочитают разные напитки и курят сигареты разных марок. Известно, что англичанин живет в красном доме; у

испанца есть собака; кофе пьют в зеленом доме; украинец пьет чай; зеленый дом – первый по правую руку от дома цвета слоновой кости; курильщик «Уинстона» держит улиток; сигареты «Кул» курят в жёлтом доме; молоко пьют в среднем доме; норвежец живет в крайнем слева доме; мужчина, курящий «Честерфилд», живет в доме, соседнем с домом мужчины, у которого есть лиса; сигареты «Кул» курят в доме, соседнем с домом, где имеется лошадь; мужчина, предпочитающий «Лаки страйк», пьёт апельсиновый сок; японец курит сигареты «Парламент»; норвежец живет в доме рядом с голубым домом, у одного из мужчин есть зебра, один из мужчин пьет воду. Кто в каком доме живет, какое животное держит, что пьёт и курит?

```
domains
        %описание дома - цвет дома, национальность живущего там мужчины, его животное, напиток и марка сигарет.
        house=h(colour,nationality,pet,drink,cigarette)
        houses=house*
        colour,nationality,pet,drink,cigarette=symbol
predicates
        colour(house, colour)
        nationality(house, nationality)
        pet(house, pet)
        drink(house, drink)
        cigarette(house, cigarette)
        first(house,houses)
        middle(house,houses)
        solve
        neighbourhood(house, house,houses)
        neighbourhoodright(house, house,houses)
clauses
        % в описании дома цвет дома находится на первом месте
        colour(h(C,_,_,_,_),C).
        % в описании дома национальность живущего там мужчины находится на втором месте
        nationality(h(_,N,_,_,_),N).
        % в описании дома животное находится на третьем месте
        pet(h(_,_,P,_,_),P).
        % в описании дома напиток находится на четвертом месте
        drink (h(_,_,_,D,_),D).
        % в описании дома сигареты находятся на пятом месте
        cigarette(h(_,_,_,_,S),S).
        %один из домов - крайний слева
        first(X,[X,_,_,_,_]).
        % один из домов находится в середине
        middle(Y,[_,_,Y,_,_]).
```

```
% перебор всех возможных соседних домов
neighbourhood(A,B,[A,B,_,_,_]).
neighbourhood(A,B,[B,A,_,_,_]).
neighbourhood(A,B,[_,A,B,_,_]).
neighbourhood(A,B,[_,B,A,_,_]).
neighbourhood(A,B,[_,_,A,B,_]).
neighbourhood(A,B,[_,_,B,A,_]).
neighbourhood(A,B,[_,_,_, A,B]).
neighbourhood(A,B,[_,_,_, B,A]).
% перебор всех возможных соседних домов, из которых дом B по правую руку от дома A
neighbourhoodright (A,B,[A,B,_,_,_]).
neighbourhoodright (A,B,[_,A,B,_,_]).
neighbourhoodright (A,B,[_,_,A,B,_]).
neighbourhoodright (A,B,[_,_,_, A,B]).
% англичанин живет в красном доме
solve:- neighbourhood(H1,_,Houses), nationality(H1,englishman),
colour(H1,red),
%; у  испанца есть собака
neighbourhood(H2,_,Houses), nationality(H2, spaniard), pet(H2,dog),
% кофе пьют в зеленом доме
neighbourhood(H3,_,Houses), colour(H3,green), drink(H3, coffee),
% украинец пьет чай
neighbourhood(H4,_,Houses), nationality(H4, ukrainian),
drink(H4, tea),
% зеленый дом – первый по правую руку от дома цвета слоновой кости
neighbourhoodright(H5,H6,Houses), colour(H6,green),
colour(H5, ivory),
%курильщик «Уинстона» держит улиток
neighbourhood(H7,_,Houses), cigarette(H7,winston), pet(H7, snail),
% сигареты «Кул» курят в жёлтом доме;
neighbourhood(H8,_,Houses), cigarette(H8,cool), colour(H8, yellow),
% молоко пьют в среднем доме
middle(H9,Houses), drink(H9, milk),
% норвежец живет в крайнем слева доме;
first(H10, Houses), nationality(H10, norwegian),
% мужчина, курящий «Честерфилд», живет в доме, соседнем с домом мужчины, у которого есть лиса
neighbourhood(H11,H12,Houses),
cigarette(H11,chesterfild),pet(H12,fox),
%; сигареты «Кул» курят в доме, соседнем с домом, где имеется лошадь
neighbourhood(H13,H14,Houses),
cigarette(H13,cool),pet(H14,horse),
```

```
% мужчина, предпочитающий «Лаки страйк», пьёт апельсиновый сок
neighbourhood(H15,_,Houses),cigarette(H15,lakestrike),
drink(H15,orange),
% японец курит сигареты «Парламент»
neighbourhood(H16,_,Houses),nationality(H16,japanese),
cigarette(H16,parlament),
% норвежец живет в доме рядом с голубым домом
neighbourhood(H17,H18,Houses),nationality(H17,norwegian),
colour(H18,blue),
% у одного из мужчин есть зебра
neighbourhood(H19,_,Houses), pet(H19,zebra),
% один из мужчин пьет воду
neighbourhood(H20,_,Houses), drink(H20,water),
write(Houses).

goal
    solve.
```

6.Основы работы в Visual Prolog.

6.1. Создание TestGoal –проекта для выполнения программ

Для использования утилиты TestGoal для выполнения программ требуется определить некоторые опции компилятора Visual Prolog. Для этого необходимо выполнить следующие действия:

1. Запустите среду визуальной разработки Visual Prolog.
2. Создайте новый проект: выберите команду Project | New Project, активизируется диалоговое окно Application Expert.
3. Определите базовый каталог и имя проекта, рекомендуется имя в поле Base Directory задать по следующему образцу: C:\student\641\ivanov\TestGoal, а имя в поле Project Name следует определить как TestGoal. Далее следует установить флажок Multiprogrammer Mode и щелкнуть мышью внутри полей Name of.VPR File и Name of.PRJ File. Определите цель проекта: на вкладке Target рекомендуется выбрать следующие параметры: Windows32, Easywin, exe, Prolog.
4. Установите требуемые опции компилятора для созданного TestGoal-проекта, для этого следует активизировать диалоговое окно Compiler Options при помощи команды Options | Project | Compiler Options. Далее откройте вкладку Warnings и выполните следующие действия:
    - Установите переключатель Nondeterm. Это нужно для того, чтобы компилятор принимал по умолчанию, что все определенные пользователем предикаты –

недетерминированные (могут породить более одного решения);

- Снимите флажки Non Quoted Symbols, Strong Type Conversion Check, Check Type of Predicates.
- Нажмите кнопку OK, чтобы сохранить установки опций компилятора.

6.2. Запуск и тестирование программы.

Для проверки правильности настройки системы, создадим простую тестовую программу. Для создания нового окна редактирования можно использовать команду меню File | New. Редактор среды визуальной разработки – стандартный текстовый редактор.

Введите в окне редактирования следующий текст:

Goal write (“Hello world!”),nl.

Это раздел цели программы и для того, чтобы она могла быть выполнена, следует активизировать команду Project | Test Goal или нажать комбинацию клавиш <Ctrl> + <G>.

Результат выполнения программы будет расположен вверху в отдельном окне, которое необходимо закрыть перед тем, как тестировать другую Goal.

Утилита Test Goal интерпретирует Goal как специальную программу, которая компилируется, генерируется в исполняемый файл и Test Goal запускает его на выполнение. Эта утилита внутренне расширяет заданный код Goal, чтобы сгенерированная программа находила все возможные решения и показывала значения всех используемых переменных.

Замечание: утилита Test Goal компилирует только тот код, который определен в активном окне редактора.

7.Задания для лабораторных работ.

7.1 Задания для лабораторной работы на тему: работа со списками

Вариант №1

Написать программу вычисления номера элемента в списке.

Вариант №2

Написать программу замены всех вхождений элемента в список на заданную константу.

Вариант №3

Написать программу замены N-го элемента в списке на заданную константу.

Вариант №4

Написать программу удаления N-го элемента в списке.

Вариант №5

Написать программу удаления M элементов из списка, начиная с N-ой позиции.

Вариант №6

Написать программу объединения двух списков в третий так, чтобы нечетные (по номеру) элементы были из первого списка, а четные - из второго.

Вариант №7

Написать программу разделения списка на два так, чтобы нечетные (по номеру) элементы были в первом списке, а четные - во втором.

Вариант №8

Написать программу разделения списка на два так, чтобы в первом списке были элементы с первого до N-го, а во втором - с N+1 до последнего.

Вариант №9

Написать программу разделения списка на два так, чтобы в первом списке были элементы с первого до заданного значения элемента, а во втором - остальные.

Вариант №10

Написать программу вставки заданного элемента в список на N-ую позицию.

Вариант №11

Написать программу замены элементов списка, номера которых заданы другим списком, на произвольную константу.

Вариант №12

Написать программу удаления из списка всех вхождений заданного элемента.

Вариант №13

Написать программу удаления элементов из списка, номера которых заданы другим списком.

Вариант №14

Написать программу объединения двух списков в третий так, чтобы одинаковые элементы из разных списков не повторялись.

Вариант №15

Написать программу инвертирования списка.

Вариант №16

Написать программу вставки в список другого списка, начиная с N-ой позиции.

Вариант №17

Написать программу определения номера позиции в списке, с которого начинается заданный подсписок.

Вариант №18

Написать программу замены m элементов списка, начиная с N-ой позиции, на произвольную константу.

Вариант №19

Написать программу подсчета количества вхождений элемента в список.

Вариант №20

Написать программу вычисления числа элементов списка, после заданного.

Вариант №21

Написать программу вычисления числа элементов списка, после N-ой позиции.

Вариант №22

Написать программу замены всех четных (по номеру) элементов списка, на произвольную константу.

Вариант №23

Написать программу замены всех нечетных (по номеру) элементов списка, начиная с N-ой позиции, на произвольную константу.

Вариант №24

Написать программу вставки m раз в список произвольной константы, начиная с N-ой позиции.

Вариант №25

Написать программу создания нового списка, путем дублирования всех элементов исходного списка.

Вариант №26

Написать программу разделения исходного списка по N-му элементу и соединения полученных частей в новый список таким образом, чтобы первая часть стала второй, а вторая - первой.

Вариант №27

Написать программу разделения исходного списка на три части: с первого по N-ый элемент, с N+1 по M-ый и с M+1 элемента до конца списка.

Вариант №28

Написать программу удаления из исходного списка N элементов с конца списка.

Вариант №29

Написать программу замены на заданную константу N элементов исходного списка, считая с конца списка.

Вариант №30

Написать программу дописывания заданной константы N раз в конец списка.

7.2 Задания для лабораторной работы на тему: получение структурированной информации из базы данных.

Текст задания.

1. Создать базу данных о заданной предметной области в виде множества фактов языка Пролог (не менее 5 фактов). Информацию о каждом компоненте БД представить в виде структуры.

2. Разработать набор предикатов, осуществляющих взаимодействие с БД, при помощи которых можно реализовать все типы запросов, приведенные в варианте задания. Найденные решения записать в виде фактов внутренней базы данных Пролога.

3. Предусмотреть проверку факта, являющегося ответом на запрос в БД. Если такой факт существует, то выдать его в качестве ответа на запрос. Если такого факта не существует в базе данных, то запустить запрос на выполнение и записать результат в БД.

Вариант №1.

Предметная область – семья. Каждая семья может быть описана структурой из трех компонент: мужа, жены и детей. Каждый член семьи может быть описан структурой: имя, отчество, фамилия, год рождения, пол, ежемесячный доход. Для детей добавить поле «близнец».

Реализовать следующие типы запросов:

1. Проверить, существует ли в БД заданный человек (по ФИО);
2. Найти всех работающих детей;
3. Найти всех работающих мужей, чей доход больше чем у жены;
4. Найти всех людей, которые не работают и родились до указанного года;
5. Найти число семей, у которых есть близнецы.

Вариант №2.

Предметная область – семья. Каждая семья может быть описана структурой из трех компонент: мужа, жены и детей. Каждый член семьи может быть описан структурой: имя, отчество, фамилия, год рождения, пол, ежемесячный доход. Для детей добавить поле «близнец».

Реализовать следующие типы запросов:

1. Найти всех близнецов;
2. Найти всех детей, родившихся в заданном году;
3. Найти всех работающих жен, чей доход больше заданной суммы;
4. Найти фамилии людей, у которых есть заданное число детей.
5. Найти самого старшего ребенка в БД.

Вариант №3.

Предметная область – семья. Каждая семья может быть описана структурой из трех компонент: мужа, жены и детей. Каждый член семьи может быть описан структурой: имя, отчество, фамилия, год рождения, пол, ежемесячный доход. Для детей добавить поле «близнец».

Реализовать следующие типы запросов:

1. Найти всех людей, чей доход меньше заданного;
2. Найти всех детей, младше заданного возраста;
3. Найти всех неработающих жен, которые родились позже заданного года;
4. Найти всех детей, у которых разница в возрасте родителей превышает заданную величину;
5. Подсчитать количество семей, у которых нет близнецов.

Вариант №4.

Предметная область – библиотека. Каждая книга может быть описана структурой: название, автор, список изданий, число экземпляров. Автор может быть описан структурой: имя, фамилия, год рождения. Издание может быть описано структурой: издательство, номер издания, год издания, количество страниц, цена.

Реализовать следующие типы запросов:

1. Найти автора, у которого книга имеет самый ранний год издания;
2. Найти все книги, изданные более одного раза (проверка по номеру издания);
3. Найти все книги, изданные в заданном издательстве за последние десять лет;
4. Найти все книги заданного автора;
5. Найти все книги, цена которых превышает заданную сумму.

Вариант №5.

Предметная область – библиотека. Каждая книга может быть описана структурой: название, автор, список изданий, число экземпляров. Автор может быть описан структурой: имя, фамилия, год рождения. Издание может быть описано структурой: издательство, номер издания, год издания, количество страниц, цена.

Реализовать следующие типы запросов:

1. Найти книгу, у которой минимальная цена;
2. Найти все книги, изданные только один раз (проверка по номеру издания);
3. Найти всех авторов, родившихся позже указанного года;
4. Найти все издательства, в которых была издана указанная книга;
5. Найти все книги, количество страниц в которых не превышает заданного значения.

Вариант №6.

Предметная область – библиотека. Каждая книга может быть описана структурой: название, автор, список изданий, число экземпляров. Автор может быть описан структурой: имя, фамилия, год рождения. Издание может

быть описано структурой: издательство, номер издания, год издания, количество страниц, цена.

Реализовать следующие типы запросов:

1. Найти книгу, у которой максимальное число страниц;
2. Найти все книги, изданные в заданном издательстве;
3. Найти всех авторов, книги которых имеют цену, находящуюся в заданном диапазоне;
4. Найти все книги, которые имеются в нескольких экземплярах;
5. Найти все издательства, выпускавшие книги после указанного года.

Вариант №7.

Предметная область – страны мира. Каждая страна может быть описана структурой: название, площадь, географическое положение, население. Географическое положение может быть описана структурой: часть света, материк, океаны, моря, горные хребты. Население может быть описано структурой: численность, государственный язык, национальный состав. Национальный состав может быть описан структурой: национальность, численность, процент от всего населения.

Реализовать следующие типы запросов:

1. Найти страну, у которой максимальная численность населения;
2. Найти все страны, находящиеся на указанном материке с населением больше заданной величины;
3. Найти все страны, у которых однородный национальный состав (численность основной национальности более 90%);
4. Найти все страны, имеющие выход к указанному морю;
5. Найти все страны с указанным государственным языком.

Вариант №8.

Предметная область – страны мира. Каждая страна может быть описана структурой: название, площадь, географическое положение, население. Географическое положение может быть описана структурой: часть света, материк, океаны, моря, горные хребты. Население может быть описано структурой: численность, государственный язык, национальный состав. Национальный состав может быть описан структурой: национальность, численность, процент от всего населения.

Реализовать следующие типы запросов:

1. Найти страну, которую омывает больше всего морей;
2. Найти все страны, на территории которых находится указанный горный хребет;
3. Найти все страны, у которых число национальностей превышает заданную величину;
4. Найти все горные хребты, находящиеся на территории указанной страны;
5. Найти все страны, у которых численность населения меньше заданной величины.

Вариант №9.

Предметная область – страны мира. Каждая страна может быть описана структурой: название, площадь, географическое положение, население. Географическое положение может быть описана структурой: часть света, материк, океаны, моря, горные хребты. Население может быть описано структурой: численность, государственный язык, национальный состав. Национальный состав может быть описан структурой: национальность, численность, процент от всего населения.

Реализовать следующие типы запросов:

1. Найти страну, у которой максимальная плотность населения;
2. Найти все моря, которые омывают территорию указанной страны;
3. Найти страну, у которой численность ни одной из национальностей не превышает 50%;
4. Найти все страны, имеющие выход к указанному океану;
5. Найти все страны, у которых название части света совпадает с названием материка.

Вариант №10.

Предметная область – биржа труда. Каждая вакансия может быть описана структурой: название предприятия, должность, ежемесячный доход, требования к соискателю, список соискателей. Каждый соискатель может быть описан структурой: фамилия, имя отчество, соответствие требованиям. Требования к соискателю и соответствие требованиям могут быть описаны одной структурой: образование, возраст, пол, владение иностранными языками, умение работать на ПК, стаж работы по специальности.

Реализовать следующие типы запросов:

1. Найти должность, для которой существует максимальное число соискателей;
2. Найти все должности для мужчин, с высшим образованием и свободно владеющих иностранным языком;
3. Найти все предприятия, предлагающие доход выше указанного уровня;
4. Найти всех соискателей, умеющих работать на ПК, и имеющих стаж работы более 5 лет;
5. Найти предприятия, у которых есть заданная вакансия.

Вариант №11.

Предметная область – биржа труда. Каждая вакансия может быть описана структурой: название предприятия, должность, ежемесячный доход, требования к соискателю, список соискателей. Каждый соискатель может быть описан структурой: фамилия, имя отчество, соответствие требованиям. Требования к соискателю и соответствие требованиям могут быть описаны одной структурой: образование, возраст, пол, владение иностранными языками, умение работать на ПК, стаж работы по специальности.

Реализовать следующие типы запросов:

1. Найти всех соискателей, которые соответствуют требованиям по заданной должности;
2. Подсчитать количество соискателей, имеющих высшее образование;
3. Найти все должности для соискателей, указанной специальности;
4. Найти все должности для мужчин с ежемесячным доходом выше указанного значения;
5. Найти все должности, для которых не требуется высшего образования.

Вариант №12.

Предметная область – биржа труда. Каждая вакансия может быть описана структурой: название предприятия, должность, ежемесячный доход, требования к соискателю, список соискателей. Каждый соискатель может быть описан структурой: фамилия, имя отчество, соответствие требованиям. Требования к соискателю и соответствие требованиям могут быть описаны одной структурой: образование, возраст, пол, владение иностранными языками, умение работать на ПК, стаж работы по специальности.

Реализовать следующие типы запросов:

1. Найти должность, у которой минимальный ежемесячный доход;
2. Найти все должности для мужчин, с указанным уровнем образования, владеющих хотя бы одним иностранным языком;
3. Найти предприятия, для вакансий которых нет соискателей;
4. Найти все должности для женщин, не старше указанного возраста;
5. Найти все должности, для которых требуется знание иностранного языка.

Вариант №13.

Предметная область – служба знакомств. Каждый клиент может быть описан структурой: фамилия, имя, отчество, характеристика клиента, требования к партнеру, список возможных партнеров. Характеристика клиента и требования к партнеру могут быть описаны одной структурой: возраст, образование, национальность, ежемесячный доход, владение жилой площадью, наличие детей, отсутствие вредных привычек. Возможный партнер может быть описан следующей структурой: фамилия, имя, отчество, характеристика партнера. Характеристика партнера может быть описана структурой, одинаковой со структурой характеристики клиента.

Реализовать следующие типы запросов:

1. Подсчитать количество клиентов в БД;
2. Найти всех клиентов, с указанным уровнем образования, имеющих жилую площадь, без вредных привычек;
3. Найти всех возможных партнеров с указанной национальностью;
4. Найти всех клиентов, которым необходим партнер, не старше указанного возраста и не имеющий детей;
5. Найти клиента, которому требуется самый молодой партнер.

Вариант №14.

Предметная область – служба знакомств. Каждый клиент может быть описан структурой: фамилия, имя, отчество, характеристика клиента, требования к партнеру, список возможных партнеров. Характеристика клиента и требования к партнеру могут быть описаны одной структурой: возраст, образование, национальность, ежемесячный доход, владение жилой площадью, наличие детей, отсутствие вредных привычек. Возможный партнер может быть описан следующей структурой: фамилия, имя, отчество, характеристика партнера. Характеристика партнера может быть описана структурой, одинаковой со структурой характеристики клиента.

Реализовать следующие типы запросов:

1. Найти всех клиентов, для которых требования к партнеру совпадают с характеристикой партнера;
2. Найти всех партнеров указанного клиента, у которых есть дети;
3. Найти всех клиентов с заданным уровнем дохода и младше указанного возраста;
4. Найти самого старого клиента службы знакомств;
5. Найти всех клиентов, у которых нет жилой площади и есть вредные привычки.

Вариант №15.

Предметная область – служба знакомств. Каждый клиент может быть описан структурой: фамилия, имя, отчество, характеристика клиента, требования к партнеру, список возможных партнеров. Характеристика клиента и требования к партнеру могут быть описаны одной структурой: возраст, образование, национальность, ежемесячный доход, владение жилой площадью, наличие детей, отсутствие вредных привычек. Возможный партнер может быть описан следующей структурой: фамилия, имя, отчество, характеристика партнера. Характеристика партнера может быть описана структурой, одинаковой со структурой характеристики клиента.

Реализовать следующие типы запросов:

1. Найти самого молодого возможного партнера в БД;
2. Найти клиента, у которого нет возможных партнеров;
3. Найти всех клиентов указанной национальности, не старше указанного возраста;
4. Найти всех партнеров указанного клиента без вредных привычек;
5. Найти всех клиентов, у которых нет детей, и которым подходит партнер, имеющий детей.

Вариант №16.

Предметная область – спортивные соревнования. Каждое соревнование может быть описано структурой: ранг соревнований, вид спорта, год проведения, страна проведения, список команд - участников. Команды - участники могут быть описаны следующей структурой: название команды, страна, результаты соревнований. Результаты соревнований могут быть описаны списком структур: название команды – соперника, страна, тип результата (выигрыш, проигрыш, ничья).

Реализовать следующие типы запросов:

1. Найти ранг соревнования, в котором участвовало минимальное число команд, в заданном году и в заданном виде спорта;
2. Найти все команды указанного ранга соревнований и года проведения, у которых не было ни одного проигрыша;
3. Найти всех соперников указанной команды в соревнованиях заданного ранга в заданном году;
4. Найти вид спорта, в котором проводились соревнования в заданном году;
5. Найти все команды указанной страны, участвовавшие в соревнованиях заданного ранга.

Вариант №17.

Предметная область – спортивные соревнования. Каждое соревнование может быть описано структурой: ранг соревнований, вид спорта, год проведения, страна проведения, список команд - участников. Команды - участники могут быть описаны следующей структурой: название команды, страна, результаты соревнований. Результаты соревнований могут быть описаны списком структур: название команды – соперника, страна, тип результата (выигрыш, проигрыш, ничья).

Реализовать следующие типы запросов:

1. Найти вид спорта, в котором участвовало максимальное число команд, в заданном году и в заданном ранге соревнований;
2. Найти все страны, где проводились чемпионаты мира по указанному виду спорта.
3. Найти всех соперников указанной команды в соревнованиях определенного ранга в заданном году;
4. Найти все соревнования, проводившиеся в заданном году;
5. Найти все команды, у которых были ничьи.

Вариант №18.

Предметная область – спортивные соревнования. Каждое соревнование может быть описано структурой: ранг соревнований, вид спорта, год проведения, страна проведения, список команд - участников. Команды - участники могут быть описаны следующей структурой: название команды, страна, результаты соревнований. Результаты соревнований могут быть описаны списком структур: название команды – соперника, страна, тип результата (выигрыш, проигрыш, ничья).

Реализовать следующие типы запросов:

1. Найти год, в который участвовало максимальное число команд, в заданном виде спорта и в заданном ранге соревнований;
2. Найти вид спорта, в котором выступает заданная команда;
3. Найти все команды, которые участвовали в Олимпийских играх по определенному виду спорта;
4. Найти все команды, участвовавшие в соревнованиях в заданном году;

5. Найти все команды определенной страны, у которых были выигрыши.

Вариант №19.

Предметная область – видеотека. Каждая видеокассета может быть описана структурой: название фильма, год создания, киностудия, атрибуты фильма. Атрибуты фильма могут быть описаны структурой: автор сценария, режиссер, список фамилий исполнителей главных ролей, премии. Премии могут быть описаны списком из следующих структур: название фестиваля, год проведения.

Реализовать следующие типы запросов:

1. Найти фильм, получивший больше всего премий;
2. Подсчитать число фильмов указанного режиссера;
3. Найти всех режиссеров, фильмы которых создавались на заданной киностудии;
4. Найти все фильмы, определенного актера за указанный период времени;
5. Найти всех сценаристов, в фильмах которых снимался определенный актер.

Вариант №20.

Предметная область – видеотека. Каждая видеокассета может быть описана структурой: название фильма, год создания, киностудия, атрибуты фильма. Атрибуты фильма могут быть описаны структурой: автор сценария, режиссер, список фамилий исполнителей главных ролей, премии. Премии могут быть описаны списком из следующих структур: название фестиваля, год проведения.

Реализовать следующие типы запросов:

1. Найти сценариста, в фильме которого снялось минимальное число актеров;
2. Найти режиссеров и сценаристов, у которых есть фильмы, получившие премии;
3. Найти все фильмы указанного режиссера, снятого на определенной киностудии;
4. Найти всех исполнителей главных ролей указанного фильма;
5. Найти все киностудии, которые работали с указанным режиссером.

Вариант №21.

Предметная область – видеотека. Каждая видеокассета может быть описана структурой: название фильма, год создания, киностудия, атрибуты фильма. Атрибуты фильма могут быть описаны структурой: автор сценария, режиссер, список фамилий исполнителей главных ролей, премии. Премии могут быть описаны списком из следующих структур: название фестиваля, год проведения.

Реализовать следующие типы запросов:

1. Найти режиссера, у которого фильм получил максимальное число премий;

2. Найти все фильмы, получившие премии, в которых снимался указанный актер;
3. Найти все фильмы определенного сценариста, снятые в указанном году.
4. Найти количество актеров, снимавшихся на определенной киностудии;
5. Найти всех актеров, снимавшихся в фильмах определенного сценариста.

Вариант №22.

Предметная область – учебная группа факультета. Каждая учебная группа может быть описана структурой: название факультета, код специальности, номер группы, состав группы. Состав группы может быть описан списком структур, описывающих отдельного студента: фамилия, имя, отчество, обучение на военной кафедре, сводная ведомость. Сводная ведомость может быть описана списком из следующих структур: предмет, оценка.

Реализовать следующие типы запросов:

1. Подсчитать число групп определенной специальности;
2. Найти оценку определенного студента по заданному предмету;
3. Найти группу, которая сдала больше всего предметов в сессию;
4. Найти всех студентов, имеющих задолженности;
5. Подсчитать число студентов, обучающихся на военной кафедре.

Вариант №23.

Предметная область – учебная группа факультета. Каждая учебная группа может быть описана структурой: название факультета, код специальности, номер группы, состав группы. Состав группы может быть описан списком структур, описывающих отдельного студента: фамилия, имя, отчество, обучение на военной кафедре, сводная ведомость. Сводная ведомость может быть описана списком из следующих структур: предмет, оценка.

Реализовать следующие типы запросов:

1. Подсчитать общее число студентов на указанном факультете;
2. Найти группу, у которой больше всего отличников;
3. Найти все предметы, которые изучала определенная группа;
4. Найти всех студентов, не обучающихся на военной кафедре;
5. Найти все группы, изучавшие определенный предмет.

Вариант №24.

Предметная область – учебная группа факультета. Каждая учебная группа может быть описана структурой: название факультета, код специальности, номер группы, состав группы. Состав группы может быть описан списком структур, описывающих отдельного студента: фамилия, имя, отчество, обучение на военной кафедре, сводная ведомость. Сводная ведомость может быть описана списком из следующих структур: предмет, оценка.

Реализовать следующие типы запросов:

1. Подсчитать средний балл сессии по указанной группе;
2. Найти группу, в которой минимальное число студентов;
3. Найти все предметы в указанной группе, по которым сдавался экзамен;
4. Найти код специальности указанной группы;
5. Найти всех студентов, обучающихся на военной кафедре.

Вариант №25.

Предметная область – база данных продажи автомобилей. Каждый автомобиль может быть описана структурой: марка автомобиля, страна фирмы-изготовителя, список фирм-продавцов. Фирма-продавец может быть описана структурой: название фирмы, страна, список имеющихся моделей. Модель может быть описана структурой: наименование модели, цена, список имеющихся расцветок.

Реализовать следующие типы запросов:

1. Найти марку автомобиля, которую продает больше всего фирм;
2. Подсчитать число стран, в которых продаются автомобили заданной марки;
3. Найти все фирмы, продающие автомобили заданной марки;
4. Найти все модели автомобилей, цена которых ниже заданной;
5. Найти все фирмы, которые продают автомобили заданной модели.

Вариант №26.

Предметная область – база данных продажи автомобилей. Каждый автомобиль может быть описана структурой: марка автомобиля, страна фирмы-изготовителя, список фирм-продавцов. Фирма-продавец может быть описана структурой: название фирмы, страна, список имеющихся моделей. Модель может быть описана структурой: наименование модели, цена, список имеющихся расцветок.

Реализовать следующие типы запросов:

1. Найти марку и модель автомобиля, у которой минимальная цена;
2. Подсчитать число расцветок автомобиля заданной модели у определенного продавца;
3. Найти все страны-изготовителя, выпускающие автомобили заданной марки;
4. Найти все марки автомобилей, продающиеся в заданной стране;
5. Найти все фирмы, которые продают автомобили заданной расцветки.

Вариант №27.

Предметная область – база данных продажи автомобилей. Каждый автомобиль может быть описана структурой: марка автомобиля, страна фирмы-изготовителя, список фирм-продавцов. Фирма-продавец может быть описана структурой: название фирмы, страна, список имеющихся моделей. Модель может быть описана структурой: наименование модели, цена, список имеющихся расцветок.

Реализовать следующие типы запросов:

1. Найти марку и модель автомобиля, у которой максимальное число расцветок;
2. Подсчитать число фирм, в которых продаются автомобили заданной марки;
3. Найти все фирмы, продающие автомобили в заданной стране;
4. Найти все марки автомобилей, выпускающиеся в заданной стране;
5. Найти все фирмы, которые продают автомобили заданной цены.

Вариант №28.

Предметная область – расписание движения самолетов. Каждый рейс может быть описан структурой: название авиакомпании, номер рейса, пункт отлета, пункт прилета, время отлета, время прилета, список пунктов промежуточных посадок, список тарифов. Тариф может быть описан структурой: тип класса салона, цена.

Реализовать следующие типы запросов:

1. Найти авиакомпанию, у которой максимальная стоимость билета по заданному маршруту;
2. Подсчитать число авиакомпаний, самолеты которых летают в заданный город (включая промежуточные посадки);
3. Найти все номера рейсов, улетающих до указанного времени;
4. Найти все авиакомпании, самолеты которых летают из указанного города;
5. Найти все рейсы, на которые есть билеты эконом класса.

Вариант №29.

Предметная область – расписание движения самолетов. Каждый рейс может быть описан структурой: название авиакомпании, номер рейса, пункт отлета, пункт прилета, время отлета, время прилета, список пунктов промежуточных посадок, список тарифов. Тариф может быть описан структурой: тип класса салона, цена.

Реализовать следующие типы запросов:

1. Найти авиакомпанию, у которой минимальная продолжительность полета по заданному маршруту;
2. Подсчитать число рейсов, совершающих промежуточную посадку в заданном пункте;
3. Найти все авиакомпании, у которых есть билеты бизнес класса;
4. Найти все рейсы, прилетающие в указанный пункт до заданного времени;
5. Найти все рейсы, летающие по заданному маршруту.

Вариант №30.

Предметная область – расписание движения самолетов. Каждый рейс может быть описан структурой: название авиакомпании, номер рейса, пункт отлета, пункт прилета, время отлета, время прилета, список пунктов промежуточных посадок, список тарифов. Тариф может быть описан структурой: тип класса салона, цена.

Реализовать следующие типы запросов:

1. Найти авиакомпанию, у которой максимальное число тарифов на заданный маршрут;
2. Подсчитать число рейсов, улетающих из заданного города до указанного времени;
3. Найти все авиакомпании, у которых есть билеты класса люкс;
4. Найти все рейсы, у которых цена билета ниже заданной;
5. Найти все авиакомпании, у которых время полета меньше заданного.

7.3 Задания для лабораторной работы на тему: решение логических головоломок

Вариант №1

Три друга заняли первое, второе и третье места на школьной олимпиаде. Друзья учатся в разных классах, зовут их по-разному, и они участвовали в олимпиадах по разным предметам. Алексей участвовал в олимпиаде по химии, и он выступил лучше, чем девятиклассник. Семиклассник Сергей выступил лучше участника олимпиады по физике. Участник олимпиады по математике занял первое место. Кто из них в каком классе учится, и кто, в какой олимпиаде участвовал?

Вариант №2

Трое хищников – волк, медведь и лиса пошли на охоту. У них разный возраст и они охотились на разную дичь. Медведь любит рыбу, и поймал дичи больше, чем тот, кому 2 года. Трехлетний волк поймал больше, чем любитель зайцев. Тот, кто любит кур, поймал больше всех. Одному из них 4 года. Определите, у кого какой возраст и кто какую дичь предпочитает.

Вариант №3

Три друга – Иван, Дмитрий и Степан преподают биологию, физику и химию в школах Москвы, Санкт-Петербурга и Киева. Иван – не в Москве, Дмитрий – не в Санкт-Петербурге. Москвич не преподаёт физику. Тот, кто живёт в Санкт-Петербурге, преподаёт химию. Дмитрий не преподаёт биологию. Кто, в каком городе живёт и что преподаёт?

Вариант №4

Три ученика – Коля, Миша и Андрей сидят в классе за партами первого ряда. У них разный цвет волос и они любят разные предметы. Миша любит физику и сидит ближе к классной доске, чем рыжий. Блондин Коля сидит ближе к классной доске, чем любитель литературы. Тот, кто любит математику, сидит за первой партой. Один из них брюнет. Определите, у кого какой цвет волос, и кто какой предмет предпочитает.

Вариант №5

Три друга заняли первое, второе и третье места в соревнованиях универсиады. Друзья разной национальности, зовут их по-разному, и любят они разные виды спорта. Майкл предпочитает баскетбол и играет лучше, чем американец. Израильтянин Саймон играет лучше теннисиста. Игрок в крикет занял первое место. Кто из них австралиец? Каким видом спорта увлекается

Ричард?

Вариант №6

Кондратьев, Давыдов и Фёдоров живут на одной улице. Один из них - столяр, другой – маляр, третий – водопроводчик. У столяра самый большой дом из троих и у него нет автомобиля. Федорову нравится машина Кондратьева и его дом меньше, чем у маляра. Определите, кто чем занимается, у кого есть машина.

Вариант №7

Браун, Гриффит, Клеменс и Грин - четверо студентов университетов разных стран приехали на международный фестиваль молодёжи и студентов. Один из них – канадец, второй – американец, третий – англичанин, четвёртый – австралиец. Браун и Клеменс были на концерте, в котором принимал участие их знакомый англичанин. Гриффит и австралиец знакомы, так как пели дуэтом под аккомпанемент их знакомого американца. Австралиец пригласил к себе в гости своего знакомого Грина и собирается пригласить своего знакомого Брауна. Определите национальности студентов.

Вариант №8

В одном театре работают четыре актёра: Смирнов, Снегов, Морев и Никитин. Один из них играет роль Отелло, другой – короля Лира, третий – Ромео, четвёртый – Гамлета. Смирнов – не Отелло и не Гамлет. Морев – не Ромео и не Отелло. Никитин – не Гамлет, не Отелло. Снегов не играет ни Гамлета, ни Ромео. Если Морев играет Гамлета, то Смирнов не играет короля Лира. Кто из актёров кого играет?

Вариант №9

Пол, Джон и Джордж – три музыканта. Один из них – гитарист, другой – ударник, третий – пианист. Каждый из них выступает в составе одной из рок-групп - «Ромашка», «Василёк» или «Колокольчик». Тот, кто выступает в составе «Ромашки», не гитарист. Джон выступает в составе не «Василька». Пол – выступает в составе не «Ромашки». Ударник выступает в составе рок-группы «Василёк». Джон не пианист. На каком инструменте играет каждый из трёх музыкантов, и в составе какой рок-группы выступает?

Вариант №10

Лена, Коля, Марина и Саша работают на одном предприятии конструктором, технологом, экономистом и дизайнером. Лена и Марина учились вместе с экономистом. Коля и дизайнер в обеденный перерыв часто играют в теннис с технологом. Дизайнер постоянно сталкивается по работе с Леной и иногда с Сашей. Кто кем работает?

Вариант №11

Трое мальчиков Костя, Фома и Марат дружили с тремя девочками – Женей, Светой и Мариной. Но вскоре компания разделилась на пары, потому что оказалось, что Света ненавидит ходить на лыжах. Костя, Женин брат, часто катается со своей подружкой на лыжах. А Фома бежит на свидание к Костиной сестре. С кем же проводит время Марат?

Вариант №12

В комнате женского общежития живут четыре девушки: Мира, Мона,

Мод и Мэри. Как-то раз они собрались все вместе, причём каждая занималась своим делом. Одна делала маникюр, другая – расчёсывала волосы, третья – прихорашивалась перед зеркалом, а четвёртая – читала. Точно о них известно следующее: Мира не делала маникюр и не читала. Мод не прихорашивалась перед зеркалом и не делала маникюр. Мэри не читала и не занималась маникюром. Мона не читала и не прихорашивалась перед зеркалом. Если Мод читала, то Мэри прихорашивалась перед зеркалом. Кто из девушек делала маникюр, кто расчёсывала волосы, кто прихорашивалась перед зеркалом, а кто читала?

Вариант №13

«Динамо», «Торпедо» и «Спартак» - команды-участники розыгрыша кубка по футболу из Челябинска, Краснодара и Самары. Команда из Челябинска забила меньше всех мячей. «Динамо» забила мячей больше, чем команда из Краснодара, а «Торпедо» меньше «Спартака». За какие города выступают названные команды?

Вариант №14

В одном купе поезда ехали три пассажира – мистер Джонс, мистер Смит и мистер Робинсон. Один из них житель Чикаго, другой живёт в Детройте, третий – в Окленде. За разговорами выяснилось, что мистер Джонс из Чикаго старше, чем тот, у кого нет детей. Житель Детройта самый старый. Мистер Смит, у которого двое детей, старше, чем житель Окленда. У одного из них только один ребенок. Определите, кто из какого города и у кого сколько детей.

Вариант №15

Билл, Джон и Ричард играют в одном оркестре. Они владеют разными музыкальными инструментами и выступают в костюмах разных цветов. Джон играет на саксофоне и находится ближе к дирижёру, чем тот, кто выступает в белом костюме. Билл на концерт одевает чёрный костюм и сидит ближе к дирижёру, чем флейтист. Альтист сидит к дирижёру ближе всех. Один из друзей приходит на концерт в костюме синего цвета. Определите, кто какими инструментами владеет, и в каком костюме выступает.

Вариант №16

Петров, Семёнов и Арбузов – пилот, штурман и бортинженер. Один из них летает на самолёте ТУ-134, другой на АН-24, третий на ИЛ-76. Тот, кто летает на ТУ-134, не штурман. Семёнов работает не на АН-24. Петров – не на ТУ-134. Бортинженер летает на АН-24. Семёнов не пилот. У кого какая профессия, и какой самолёт?

Вариант №17

Андреев, Борисов и Николаев – поэт, композитор и художник. У поэта нет бороды, и он зарабатывает меньше всех. Андреев считает, что Николаеву очень идет борода. Андреев зарабатывает больше композитора. Определите, кто является поэтом, кто композитором, кто художником и у кого есть борода.

Вариант №18

За столиком в кафе познакомились три девушки: Алёна, Светлана и

Марианна. Одна из них работает парикмахером в Салоне красоты, другая – модельером в Доме Мод, третья – медсестрой в санатории. Также выяснилось, что парикмахер – самая молодая из троих, Марианна старше модельера, а Светлана младше Алёны. Определите профессии девушек.

Вариант №19

В одном институте учатся три друга – Липатов, Кондратьев и Зайцев. Один из них будущий физик, второй – математик, третий – программист. У физика нет ни братьев, ни сестёр. Он самый младший из друзей. Зайцеву нравится сестра Липатова, и он старше математика. Определите будущие профессии друзей.

Вариант №20

У Васи, Вани, Славы и Вовы живут дома собака, кошка, морская свинка и попугай. Вася и Слава знакомы с хозяином кошки. Ваня и хозяин собаки часто ходят в кино с хозяином попугая. Хозяин собаки постоянно встречается в школе с Васей и иногда с Вовой. У кого какое животное?

Вариант №21

Трое коллег – Костя, Митя и Артем пошли в ресторан. У них разное семейное положение и они любят разные блюда. Митя любит рыбу и получает больше, чем тот, кто холостой. Разведенный Костя получает больше, чем любитель пельменей. Тот, кто любит плов, получает больше всех. Один из них женат. Определите, у кого какое семейное положение, и кто какое блюдо предпочитает.

Вариант №22

В книжный магазин пришли четыре подруги: Соколова, Ястребова, Орлова и Голубева. Одна из них искала книгу «Поющие в терновнике», другая – «Финансиста», третья – «Охоту на овец», четвёртая – «Заводной апельсин». Соколова не искала книги «Поющие в терновнике» и «Заводной апельсин». Орлова – не искала книги «Охота на овец» и «Поющие в терновнике». Голубева не искала книги «Заводной апельсин» и «Поющие в терновнике». Если Ястребова искала книгу «Поющие в терновнике», то Голубева искала книгу «Финансист». Ястребова не искала книги «Заводной апельсин» и «Охота на овец». Какая девушка, какую книгу искала?

Вариант №23

Боб, Джек и Рональд часто ходят в кино. Они любят разные жанры и предпочитают разные кинотеатры. Джек любит комедии и садится ближе, чем тот, кто ходит в кинотеатр «Звезда». Боб ходит в кинотеатр «Самара» и сидит ближе, чем любитель триллеров. Тот, кто любит боевики, сидит к экрану ближе всех. Один из друзей ходит в кинотеатр «Огонек». Определите, кто какой жанр любит, и в какой кинотеатр предпочитает ходить.

Вариант №24

Кукушкина, Петухова и Галкина – модельер, менеджер и страховой агент. Одна из них водит форд, другая рено, третья ауди. Тот, кто ездит на форде, не менеджер. Петухова ездит не на рено. Кукушкина – не на форде. Страховой агент ездит на рено. Петухова не модельер. У кого какая профессия, и какой автомобиль?

Вариант №25

Три семейные пары– Ивановы, Петровы и Сидоровы купили путевки за 2000$, 3000$ и 5000$ в Турцию, Италию и Испанию. Ивановы поехали не в Турцию, Петровы – не в Италию. Те, кто поехали в Турцию, не платили за путевку 5000$. Те, кто поехали в Италию, заплатили 3000$. Петровы не платили за путевку 2000$. Кто в какой стране отдыхал и кто сколько заплатил за путевку?

Вариант №26

«Тройка» думает, что «Туз» не в своём уме. «Четвёрка» думает, что «Тройка» и «Двойка» обе не могут быть не в своём уме. «Пятёрка» думает, что «Туз» и «Четвёрка» либо оба не в своём уме, либо оба в своём уме. «Шестёрка» думает, что «Туз» и «Двойка» оба в своём уме. «Семёрка» думает, что «Пятёрка» не в своём уме. «Валет» думает, что «Шестёрка» и «Семёрка» обе не могут быть не в своём уме. В своём ли уме «Валет»?

Вариант №27

Король думает, что королева думает, что она не в своём уме. В своём ли уме король?

Вариант №28

Антон, Максим и Дима учатся в художественной школе. Каждый ученик художественной школы рисует или портреты, или пейзажи, или натюрморты и у каждого ученика один любимый художник - И.Репин, И.Шишкин или А.Рублёв. Антон рисует портреты и учится лучше, чем тот, кто любит И.Репина. Максим любит И.Шишкина и учится лучше, чем тот, кто рисует пейзажи. Тот, кто рисует натюрморты, учится лучше всех. Один из учеников любит А.Рублёва. Определите, кто из учеников художественной школы, что рисует и у кого какой любимый художник.

Вариант №29

На экскурсию в Эрмитаж приехали трое туристов – Кузнецов, Смирнов и Попов. Один из них из Казани, другой – из Самары, третий – из Москвы. Выяснилось, что житель Казани купил билет самым первым из троих. Кузнецов из Самары купил билет раньше, чем тот, кто живет в гостинице №1. Смирнов, живущий в гостинице №2, купил билет раньше, чем москвич. Один из них живет в гостинице №3.

Определите, кто из какого города, и кто в какой гостинице живет.

Вариант №30

Тони, Майкл и Джон – спортсмены. Один из них боксер, второй футболист, третий – пловец. Один из них любит снег, второй любит дождь, третий любит солнечную погоду. Тот, кто любит солнечную погоду, не боксер. Тони не любит снег. Майкл – не любит солнечную погоду. Футболист любит снег. Тони не пловец. У кого какая любимая погода и кто каким спортом занимается?

Список рекомендуемой литературы

1. Адаменко А.Н., Кучуков А. Логическое программирование и Visual Prolog – Спб.: БХВ – Петербург, 2003.
2. Братко И. Алгоритмы искусственного интеллекта на языке Prolog. М.: Вильямс, 2004. – 637 с.
3. Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог: Пер. с англ. М.: Мир.1990.

Вопросы к экзамену по курсу «Системы искусственного интеллекта».

1. Классификация систем искусственного интеллекта. Системы с интеллектуальным интерфейсом, экспертные системы, самообучающиеся системы, адаптивные системы.
2. Характеристики знаний. Логические, сетевые, продукционные и фреймовые модели представления знаний.
3. Исчисление высказываний. Алфавит и правила построения формул. Правила вывода в исчислении высказываний. Понятие аксиоматической системы. Свойства аксиоматических систем.
4. Интерпретация формул в исчислении высказываний. Общезначимые и противоречивые формулы. Определение логического следствия. Основные теоремы доказательства логического следствия.
5. Исчисление предикатов первого порядка. Алфавит. Понятие терма. Определение предиката. Свободные и связанные переменные. Правила построения формул в исчислении предикатов.
6. Интерпретация формул в логике предикатов первого порядка. Общезначимые и противоречивые формулы в исчислении предикатов. Системы аксиом логики предикатов. Правила вывода в исчислении предикатов.
7. Пренексные нормальные формы. Алгоритм преобразования формул в ПНФ. Скулемовские стандартные формы. Алгоритм преобразования ПНФ в ССФ.
8. Определение резольвенты. Метод резолюций в исчислении высказываний. Фразовая форма представления дизъюнктов. Фразы Хорна.
9. Унификация в логике предикатов. Определение унификатора. Алгоритм нахождения наиболее общего унификатора. Понятие резольвенты в исчислении предикатов. Алгоритм метода резолюций.
10. Теоретические основы языка программирования Пролог. Понятие терма в Прологе. Инициализация переменных. Область действия имен. Понятие анонимной переменной. Примеры.
11. Типы предложений в языке Пролог: факты, цели, правила. Использование дизъюнкции и отрицания в Прологе. Примеры.
12. Унификация переменных в Прологе. Правила унификации. Вычисление цели. Механизм возврата. Примеры.
13. Управление поиском решений. Примеры. Процедурность Пролога.
14. Структура программ в языке Пролог. Типы данных. Использование составных термов в Прологе. Определение списка

в Прологе. Операции над списками. Примеры предикатов для работы со списками.

15. Повторение и рекурсия в Прологе. Использование механизма возврата. Метод возврата после неудачи. Правило повтора, использующее бесконечный цикл. Примеры.
16. Методы организации рекурсии. Оптимизация хвостовой рекурсии. Примеры.
17. Создание динамических баз данных. Встроенные предикаты для работы с динамическими базами. Примеры.
18. Рекурсивные структуры данных. Представление бинарных деревьев в языке Пролог. Определение бинарного дерева. Программа обхода бинарного дерева.
19. Представление графов в языке Пролог. Поиск пути на графе. Пример программы поиска пути на графе.
20. Основные стратегии решения задач в Прологе. Метод «образовать и проверить». Примеры.
21. Поиск решений в пространстве состояний. Алгоритм А*. Пример.
22. И/ИЛИ – графы. Решение игровых задач в терминах И/ИЛИ-графа. Пример решения задачи. Минимаксный принцип поиска решений.
23. Проектирование экспертных систем. Типы решаемых задач.
24. Инструментальные средства разработки. Нечёткие знания в экспертных системах.
25. Продукционные правила для представления знаний. Формирование ответа на вопрос «почему». Формирование ответа на вопрос «как».

Тесты для контроля знаний

Дисциплина «Системы искусственного интеллекта»

Студент_____ группа_____ дата экзамена_____

Вариант №1

№	Вопрос	Варианты ответа	Ответ	Балл
1	Гипертекстовые системы относятся к классу:	1. Экспертных систем. 2. Самообучающихся систем. 3. Систем с интеллектуальным интерфейсом. 4. Адаптивных систем.		
2	Упорядочите по возрастанию приоритетов следующую последовательность логических операций:	1. Отрицание. 2. Импликация. 3. Конъюнкция. 4. Дизъюнкция.		
3	Заданная таблица истинности соответствует операции: X Y ? И И И И Л Л Л И Л Л Л Л	1. Дизъюнкции. 2. Импликации. 3. Эквивалентности. 4. Исключающему ИЛИ. 5. Конъюнкции.		
4	Общезначимая формула - это:	1. Формула, истинная во всех интерпретациях. 2. Формула, ложная во всех интерпретациях. 3. Формула, истинная хотя бы в одной интерпретации. 4. Формула, ложная хотя бы в одной интерпретации.		
5***	Истинностными значениями формулы $\neg(P \rightarrow Q) \leftrightarrow P$ при заданном множестве значений P и Q: И И И Л Л И Л Л - являются:	1.Л 2.И 3.И 4.Л 5.Л И Л Л И И Л Л Л И Л И Л И И Л		
6	Свойство непротиворечивости аксиоматической системы означает:	1. Никакая аксиома не выводима из других аксиом. 2. Невозможно вывести отрицание тавтологии. 3. Любая тавтология выводима из системы аксиом. 4. Любая выводимая формула есть тавтология.		
7*	*ЭС* решают задачи распознавания ситуаций.	Вставить название типа ЭС.		
8*	означает, что каждая информационная единица должна иметь уникальное имя, по которому система находит ее, а также отвечает на запросы, в которых это имя упомянуто.	Вставить название характеристики знаний.		
9***	Формула $\forall x \exists y\ (P(x,a) \rightarrow Q(f(y)))$ в заданной области интерпретации*:* D={0,1}.Оценка для a: a=1. Оценки для f: f(0)=1; f(1)=0. Оценки для P и Q: P(0,1)=И;P(1,1)=И;Q(0)=И;Q(1)=Л; - ...	1. Истинна. 2. Ложна.		
10	Следующая фраза языка: «Все целые числа являются рациональными числами», где *C(x)* - предикат «*x* – есть целое число»,	*1.* $\forall x\ C(x) \rightarrow \exists x\ R(x)$. 2. $\exists x\ C(x) \rightarrow \forall x\ R(x)$.		

	R(x) - предикат «*x* – есть рациональное число» соответствует формуле:	3. $\exists x\,(C(x)\rightarrow R(x))$. 4. $\forall x\,(C(x)\rightarrow R(x))$.		
11***	Пренексная нормальная форма формулы $\neg(\forall x\,P(x)\rightarrow\forall x\,Q(x))$ – это:	1. $\forall x\exists y\,(P(x)\vee\neg Q(x))$. 2. $\forall x\,\forall y\,(P(x)\wedge Q(x))$. 3. $\forall x\exists y\,(P(x)\wedge\neg Q(y))$. 4. $\forall x\,\forall y\,(P(x)\vee\neg Q(y))$. 5. $\forall x\exists y\,(P(x)\vee\neg Q(y))$.		
12***	Скулемовская стандартная форма формулы $\exists x\,\forall y\,\forall z\exists w\neg(P(x,y,a)\vee Q(z,w,f(x,y)))$.- это:	1. $\forall y\,\forall z\,(\neg P(a,y,a)\wedge\neg Q(z,g(y,z),f(a,y)))$. 2. $\forall y\,\forall z\,(\neg P(b,y,a)\wedge\neg Q(z,g(y,z),f(b,y)))$. 3. $\forall y\,\forall z\,(\neg P(b,y,a)\wedge\neg Q(z,f(y,z),f(b,y)))$. 4. $\forall y\,\forall z\,(\neg P(b,y,a)\wedge\neg Q(z,c,f(b,y)))$. 5. $\forall y\,\forall z\,(\neg P(b,y,a)\wedge\neg Q(z,c,f(x,y)))$.		
13	Константы унифицируются в соответствии со следующим правилом:	1. Унифицируются только тогда, когда они совпадают. 2. Унифицируются всегда, и вместо одной константы подставляется другая. 3. Не унифицируются никогда.		
14*	Для любых двух дизъюнктов C_1 и C_2, если существует литерал L_1 в C_1, который…… литералу L_2 в C_2, то вычеркнув L_1 и L_2 из C_1 и C_2 соответственно и построив дизъюнкцию оставшихся дизъюнктов, получим резолюцию C_1 и C_2.	Вставить пропущенное слово.		
15*	Метод резолюций в исчислении предикатов может быть применим к формуле представленной в…форме	Вставить название формы.		
16***	Следующее множество дизъюнктов: 1. $P(a)$. 2. $\neg D(y)\vee L(a,y)$. 3. $\neg P(x)\vee\neg Q(y)\vee\neg L(x,y)$. 4. $D(b)$. 5. $Q(b)$.	1. Противоречиво. 2. Непротиворечиво.		
17*	Областью действия переменной в языке Пролог является…:	Вставить текст.		
18*	Раздел программы … на языке Visual Prolog содержит объявления различных типов данных, используемых в программе.	Вставить название раздела.		
19	Следующий фрагмент программы иллюстрирует метод…: *clauses* *decimal (0).* *decimal (1).* *write_decimal:-decimal(C), write(C), nl, fail.* *goal* *write_decimal.*	1. Использования механизма возврата. 2. Возврата после неудачи. 3. Повтора, использующего бесконечную рекурсию. 4. Обобщенное рекурсивное правило.		
20	Следующий предикат на языке Пролог производит: *predicates* *p1 (integer, list)* *clauses* *p1 (Head, [Head|_]):-!,.* *p1 (Head, [_ |Tail]):- p1 (Head, Tail).*	1. Объединение двух списков 2. Определение первого вхождения элемента в список. 3. Определение всех вхождений элемента в список. 4. Удаление всех вхождений элемента из списка. 5. Удаление первого вхождения элемента из списка.		

21	Внутреннюю (динамическую) базу данных языка Пролог образуют:	1. Предикаты в виде фактов, объявленные в разделе predicates. 2. Предикаты в виде правил, объявленные в разделе predicates. 3. Предикаты в виде фактов, объявленные в разделе facts. 4. Предикаты в виде правил, объявленные в разделе facts.		
22**	Установить соответствие четырёх приведённых примеров с двумя понятиями: а) оптимизированная хвостовая рекурсия; б) неоптимизированная хвостовая рекурсия.	1. *count1(50).* *count1(N):-* *write(N),nl,N1=N+1,count1(N1).* *goal* *nl, count1(0).* 2. *count2(50).* *count2(N):-* *N>0,!,write(N),N1=N+1,count2(N1).* *count2(N):- write("N – отрицательное число").* *goal* *nl, count2(0).* 3. *count3(50).* *count3(N):-* *write(N),N1=N+1,count3(N1),nl.* *goal* *count3(0).* 4. *count4(50).* *count4(N):-* *write(N),nl,N1=N+1,count4(N1).* *count4(N):-N<0, write("N – отрицательное число").* *goal* *nl, count4(0).*		
23***	Найти наиболее общий унификатор (если он существует) следующего множества дизъюнктов: *{Q(a,x,f(x),u), Q(a,y,f(g(z)),b)}.*			
24***	Привести к скулемовской стандартной форме формулу: $\forall x \exists y P(x,y) \rightarrow (\exists z Q(z) \rightarrow \forall z G(z))$.			
25***	Написать программу замены всех вхождений заданного элемента в списке на другое значение.			
ИТОГО				

Примечание:

1. **Вопросы, отмеченные * требуют вставки ответа в текст вопроса,**
2. **Вопросы, отмеченные ** предполагают разделение ответов на два указанных класса,**
3. **Вопросы, отмеченные *** требуют приложения к тесту решения примера.**

Общее число баллов за тест: 40. Общее число баллов за лабораторные работы:40.

Итоговая оценка_____

Тесты рассмотрены и утверждены на заседании кафедры 23.10.2012 г.

Заведующий кафедрой ИСТ, С.А. ПРОХОРОВ
д.т.н., профессор

Составитель, к.т.н.,
доцент кафедры ИСТ И.В. ЛЁЗИНА

Дисциплина «Системы искусственного интеллекта»

Студент_____ группа_____ дата экзамена_____

Вариант №2

№	Вопрос	Варианты ответа	Ответ	Балл
1	Мультиагентные системы относятся к классу:	1. Систем с интеллектуальным интерфейсом. 2. Самообучающихся систем. 3. Адаптивных систем. 4. Экспертных систем.		
2	Упорядочите по возрастанию приоритетов следующую последовательность логических операций:	1. Отрицание. 2. Дизъюнкция. 3. Конъюнкция. 4. Эквивалентность.		
3	Заданная таблица истинности соответствует операции: X Y ? И И И И Л Л Л И И Л Л И	1. Импликации. 2. Эквивалентности. 3. Конъюнкции. 4. Дизъюнкции. 5. Исключающему ИЛИ.		
4	Необщезначимая формула - это:	1. Формула, ложная во всех интерпретациях. 2. Формула, истинная во всех интерпретациях. 3. Формула, истинная хотя бы в одной интерпретации 4. Формула, ложная хотя бы в одной интерпретации.		
5***	Истинностными значениями формулы $\neg(P\leftrightarrow Q)\rightarrow P$ при заданном множестве значений P и Q: И И И Л Л И Л Л - являются:	1.И 2.И 3.И 4.Л 5.Л И И И И Л И Л Л Л И Л Л И Л И		
6	Свойство независимости аксиоматической системы означает:	1. Любая выводимая формула есть тавтология. 2. Любая тавтология выводима из системы аксиом. 3. Никакая аксиома не выводима из других аксиом. 4. Невозможно вывести отрицание тавтологии.		
7*	……*ЭС* используются для решения задач с не полностью определенными данными и знаниями.	Вставить название типа ЭС.		
8*	……означает, что информационные единицы должны соответствовать «принципу матрешки», то есть рекурсивной вложенности одних информационных единиц в другие	Вставить название характеристики знаний.		
9***	Формула $\forall x\exists y(P(a,y)\rightarrow Q(f(x)))$ в заданной области интерпретации: D ={0,1}.Оценка для a: a=1. Оценки для f: f(0)=0; f(1)=1. Оценки для P и Q: P(1,0)=И;P(1,1)=Л;Q(0)=Л;Q(1)=Л;:	1. Истинна. 2. Ложна.		
10	Следующая фраза языка: «Существует такое рациональное	*1.* $\forall x\,(R(x)\rightarrow \exists x\,C(x))$. *2.* $\forall x\,R(x)\rightarrow\exists x\,C(x)$.		

	число, которое является целым числом», где $C(x)$ - предикат «x – есть целое число», $R(x)$ - предикат «x – есть рациональное число», соответствует формуле:	3. $\exists x\,(R(x)\rightarrow C(x))$. 4. $\exists x\,R(x)\rightarrow\forall x\,C(x)$.		
11***	Пренексная нормальная форма формулы $\neg\,(\exists x\,P(x)\rightarrow\forall x\,Q(x))$ – это:	1. $\exists x\exists y\,(P(x)\vee\neg Q(y))$. 2. $\exists x\,(P(x)\wedge\neg Q(x))$. 3. $\forall x\exists y\,(P(x)\vee\neg Q(y))$. 4. $\forall x\,\forall y\,(P(x)\wedge\neg Q(y))$. 5. $\exists x\exists y\,(P(x)\wedge\neg Q(y))$.		
12***	Скулемовская стандартная форма формулы $\exists x\exists w\,\forall y\,\forall z\neg(P(x,y,a)\vee Q(z,w,f(x,y))$- это:	1. $\forall y\,\forall z\,(\neg P(a,y,a)\wedge\neg Q(z,b,f(a,y)))$. 2. $\forall y\,\forall z\,(\neg P(b,y,a)\wedge\neg Q(z,b,f(a,y)))$. 3. $\forall y\,\forall z\neg(P(b,y,a)\vee Q(z,c,f(b,y)))$. 4. $\forall y\,\forall z\,(\neg P(b,y,a)\wedge\neg Q(z,c,f(b,y)))$. 5. $\forall y\,\forall z\neg(P(b,y,a)\vee Q(z,h(y,z),f(b,y)))$.		
13	Переменные унифицируются в соответствии со следующим правилом:	1. Унифицируются всегда. 2. Унифицируются только тогда, когда они совпадают. 3. Не унифицируются никогда.		
14*	Для любых двух дизъюнктов C_1 и C_2, если существует литерал L_1 в C_1, который контрарен литералу L_2 в C_2, то вычеркнув L_1 и L_2 из C_1 и C_2 соответственно и построив дизъюнкцию оставшихся дизъюнктов, получим……C_1 и C_2.	Вставить пропущенное слово.		
15*	Формула F, находится в … форме, если она имеет вид: $(K_1x_1)\ldots(K_nx_n)\,(M)$, где каждое (K_ix_i), $i = 1,\ldots,n$, есть или $(\forall x_i)$ или $(\exists x_i)$, и M есть формула, не содержащая кванторов.	Вставить название формы.		
16***	Следующее множество дизъюнктов: 1. $P(a)\vee\neg L(x,y)$. 2. $\neg D(y)\vee L(a,y)$. 3. $\neg P(x)\vee\neg Q(y)$. 4. $D(b)$. 5. $Q(b)$.	1. Противоречиво. 2. Непротиворечиво.		
17*	Анонимная переменная в языке Пролог используется, если её значение…:	Вставить условие использования анонимной переменной.		
18*	Раздел программы … на языке Visual Prolog содержит объявления предикатов, используемых в виде неизменяемых фактов и правил.	Вставить название раздела.		
19	Следующий фрагмент программы иллюстрирует метод…: *clauses* *fact (0, 1).* *fact (1, 1):-!.* *fact (N, R):- N1=N-1, fact(N1, R1), R=N*R1.* *goal* *fact (5, F).*	1. Использования механизма возврата. 2. Возврата после неудачи. 3. Повтора, использующего бесконечную рекурсию. 4. Обобщенное рекурсивное правило.		
20	Следующий предикат на языке Пролог производит: *predicates*	1. Определение первого вхождения элемента в список. 2. Определение всех вхождений элемента в		

	p2(list, list, list) *clauses* *p2([], L2, L2).* *p2 ([H|T1], L2, [H|T3]):- p2(T1, L2, T3).*	список. 3. Удаление всех вхождений элемента из списка. 4. Инвертирование списка. 5. Объединение двух списков.		
21	Механизм возврата Пролога осуществляет возврат:	1. На начало программы. 2. На начало предложения. 3. В ближайшую слева точку, к которой было найдено успешное решение. 4. К предыдущей подцели.		
22**	Установить соответствие четырёх приведённых примеров с двумя понятиями: а) оптимизированная хвостовая рекурсия; б) неоптимизированная хвостовая рекурсия.	*1. count1(50).* *count1(N):-* *write(N),nl,N1=N+1,check(N1),count1(N1).* *check(Z):-Z>=0.* *check(Z):-Z<0.* *goal* *nl, count1(0).* *2. count2(50).* *count2(N):-write(N),nl,N1=N+1,count2(N1).* *goal* *nl, count2(0).* *3. count3(50).* *count3(N):-write(N),N1=N+1,check(N1),!,* *count3(N1).* *check(Z):-Z>=0.* *check(Z):-Z<0.* *goal* *nl, count3(0).* 4. *count4(50).* *count4(N):-write(N),nl,N1=N+1,count4(N1).* *count4(N):-N<0, write("N – отрицательное число").* *goal* *nl, count4(0).*		
23***	Найти наиболее общий унификатор (если он существует) следующего множества дизъюнктов: *{Q(a,y,f(y)), Q(u,h(u,b),f(c)}.*			
24***	Привести к скулемовской стандартной форме формулу: $\forall x(\neg \forall y P(x,y)) \rightarrow (\exists y R(y) \wedge \exists z Q(z))$.			
25***	Написать программу формирования из списка двух других таким образом, чтобы в первом списке были элементы с M-го до N-го, а во втором - с N+1 до последнего.			
ИТОГО				

Примечание:

1. **Вопросы, отмеченные * требуют вставки ответа в текст вопроса,**
2. **Вопросы, отмеченные ** предполагают разделение ответов на два указанных класса,**
3. **Вопросы, отмеченные *** требуют приложения к тесту решения примера.**

Общее число баллов за тест: 40. Общее число баллов за лабораторные работы:40.

Итоговая оценка_____

Тесты рассмотрены и утверждены на заседании кафедры 23.10.2012 г.

Заведующий кафедрой ИСТ, С.А. ПРОХОРОВ
д.т.н., профессор

Составитель, к.т.н.,
доцент кафедры ИСТ И.В. ЛЁЗИНА

Дисциплина «Системы искусственного интеллекта»

Студент_____ группа_____ дата экзамена_____

Вариант №3

№	Вопрос	Варианты ответа	Ответ	Балл
1	Индуктивные системы относятся к классу:	1. Систем с интеллектуальным интерфейсом. 2. Самообучающихся систем. 3. Адаптивных систем. 4. Экспертных систем.		
2	Упорядочите по убыванию приоритетов следующую последовательность логических операций.	1. Отрицание. 2. Конъюнкция. 3. Дизъюнкция. 4. Импликация.		
3	Заданная таблица истинности соответствует операции: X Y ? И И И И Л И Л И И Л Л Л	1. Конъюнкции. 2. Дизъюнкции. 3. Импликации. 4. Эквивалентности. 5. Исключающему ИЛИ.		
4	Непротиворечивая формула - это:	1. Формула, истинная во всех интерпретациях. 2. Формула, истинная хотя бы в одной интерпретации. 3. Формула, ложная во всех интерпретациях. 4. Формула, ложная хотя бы в одной интерпретации.		
5***	Истинностными значениями формулы $\neg(P\wedge Q)\leftrightarrow P$ при заданном множестве значений P и Q: И И И Л Л И Л Л – являются:	1.И 2.И 3.Л 4.Л 5.И И Л И Л И И И Л И Л Л И Л И Л		
6	Свойство минимальности аксиоматической системы означает:	1. Никакая аксиома не выводима из других аксиом. 2. Невозможно вывести отрицание тавтологии. 3. Любая выводимая формула есть тавтология. 4. Любая тавтология выводима из системы аксиом.		
7*	……*ЭС* реализуют преобразование знаний в процессе решения задачи.	Вставить название типа ЭС.		
8*	……означает, что между информационными единицами должна быть предусмотрена возможность установления связей различного типа, характеризующих отношения между информационными единицами.	Вставить название характеристики знаний.		
9***	Формула $\forall x\exists y\,(P(x,a)\leftrightarrow Q(f(y)))$ в заданной области интерпретации: D={0,1}.Оценка для a: a=0. Оценки для f: f(0)=0; f(1)=1. Оценки для P и Q: P(0,0)=Л;	1. Истинна. 2. Ложна.		

	P(1,0)=И;Q(0)=Л;Q(1)=Л;			
10	Следующая фраза языка: «Не все рациональные числа являются целыми числами», где *C(x)* - предикат «*x* – есть целое число», *R(x)* - предикат «*x* – есть рациональное число» соответствует формуле:	1. $\neg \forall x\ (\neg R(x) \rightarrow C(x))$. 2. $\neg \forall x\ (R(x) \rightarrow C(x))$. 3. $\forall x\ (R(x) \rightarrow \neg C(x))$. 4. $\forall x\ (\neg R(x) \rightarrow C(x))$.		
11***	Пренексная нормальная форма формулы $\neg(\exists x P(x) \rightarrow \exists x Q(x))$- это:	1. $\forall x \exists y\ (P(x) \wedge \neg Q(y))$. 2. $\exists x \forall y\ (P(x) \vee \neg Q(x))$. 3. $\forall x \exists y\ (P(x) \vee \neg Q(x))$. 4. $\exists x \exists y\ (P(x) \wedge \neg Q(y))$. 5. $\exists x \forall y\ (P(x) \wedge \neg Q(y))$.		
12***	Скулемовская стандартная форма формулы $\forall z \exists x \exists w \forall y \neg(P(x,y,a) \vee Q(z,w,f(x,y)))$ – это:	1. $\forall y \forall z\ (\neg P(g(z),y,a) \wedge \neg Q(z,h(z)),f(g(z),y)))$. 2. $\forall y \forall z\ (\neg P(f(z),y,a) \wedge \neg Q(z,g(z),f(f(z),y)))$. 3. $\forall y \forall z\ (\neg P(g(z),y,a) \wedge \neg Q(z,g(z),f(g(z),y)))$. 4. $\forall y \forall z \neg(P(g(z),y,a) \vee Q(z,g(z)),f(g(z),y)))$. 5. $\forall y \forall z \neg(P(g(z),y,a) \vee Q(z,h(z)),f(g(z),y)))$.		
13	Переменная и константа унифицируются по следующему правилу:	1. Унифицируются всегда, вместо переменной подставляется константа. 2. Унифицируются всегда, вместо константы подставляется переменная. 3. Не унифицируются никогда.		
14*	Любая резольвента двух дизъюнктов C_1 и C_2 есть …….C_1 и C_2.	Вставить пропущенные слова.		
15*	Формула *F*, находится в … форме, если она имеет вид: $F_1 \wedge F_2 \wedge ... \wedge F_n$, $n >= 1$, где каждая из $F_1, F_2, ..., F_n$ есть дизъюнкция литералов.	Вставить название формы.		
16***	Следующее множество дизъюнктов: 1. $P(x,f(x),b)$. 2. $\neg S(x) \vee \neg S(y) \vee \neg P(x,f(y),z) \vee S(z)$. 3. $S(a)$. 4. $S(b)$.	1. Противоречиво. 2. Непротиворечиво.		
17*	Правило – это предложение языка Пролог, которое имеет…	Вставить текст, определяющий структуру правила.		
18*	Раздел программы … на языке Visual Prolog содержит объявления предикатов внутренней (динамической) базы данных:	Вставить название раздела.		
19	Следующий фрагмент программы иллюстрирует метод…: *clauses* *dec(0). dec(1). dec(2). dec (3).* *w_dec(C):- dec(C), C=2, write (C), nl.* *goal* *w_dec (C).*	1. Повтора, использующего бесконечную рекурсию. 2. Обобщенное рекурсивное правило. 3. Возврата после неудачи. 4. Использования механизма возврата.		
20	Следующий предикат на языке Пролог производит: *predicates* *p3(integer, list, list)* *clauses* *p3(X,[X|T1],T1):-!.* *p3(X, [Y|T1], [Y|T2]):-p3(X, T1, T2).*	1. Определение первого вхождения элемента в список. 2. Определение всех вхождений элемента в список. 3. Удаление всех вхождений элемента из списка. 4. Удаление первого вхождения элемента из списка.		

		5. Объединение двух списков.		
21	Стратегия поиска решения задачи «в глубину» на языке Пролог обеспечивает:	1. Простоту программирования и минимальные затраты памяти. 2. Получение первым решающего пути, минимальной длины. 3. Получение первым решающего пути, минимальной длины и простоту программирования.		
22**	Установить соответствие четырёх приведённых примеров с двумя понятиями: а) оптимизированная хвостовая рекурсия; б) неоптимизированная хвостовая рекурсия.	*1. count1(50).* *count1(N):-write(N),N1=N+1,check(N1),!,* *count1(N1).* *check(Z):-Z>=0.* *check(Z):-Z<0.* *goal* *nl, count1(0).* *2. count2(50).* *count2(N):-write(N),N1=N+1,count2(N1),nl.* *goal* *count2(0).* *3. count3(50).* *coun3(N):-* *write(N),nl,N1=N+1,check(N1),count3(N1).* *check(Z):-Z>=0.* *check(Z):-Z<0.* *goal* *nl, count3(0).* *4. count4(50).* *count4(N):-* *N>0,!,write(N),N1=N+1,count4(N1).* *count4(N):- write("N – отрицательное число").* *goal* *nl, count4(0).*		
23***	Найти наиболее общий унификатор (если он существует) следующего множества дизъюнктов: *{P(x,g(x),k(b),y), P(a,g(u),k(u),v)}.*			
24***	Привести к скулемовской стандартной форме формулу: *¬∃zP(z)∧(∀yQ(y)↔∀xG(x)).*			
25***	Написать программу вычисления номера максимального элемента в списке.			
ИТОГО				

Примечание:

1. **Вопросы, отмеченные * требуют вставки ответа в текст вопроса,**
2. **Вопросы, отмеченные ** предполагают разделение ответов на два указанных класса,**
3. **Вопросы, отмеченные *** требуют приложения к тесту решения примера.**

Общее число баллов за тест: 40. Общее число баллов за лабораторные работы:40.

Итоговая оценка_____

Тесты рассмотрены и утверждены на заседании кафедры 23.10.2012 г.

Заведующий кафедрой ИСТ, С.А. ПРОХОРОВ
д.т.н., профессор

Составитель, к.т.н.,
доцент кафедры ИСТ И.В. ЛЁЗИНА

Дисциплина «Системы искусственного интеллекта»

Студент_____ группа_____ дата экзамена_____

Вариант №4

№	Вопрос	Варианты ответа	Ответ	Балл
1	Нейронные сети относятся к классу:	1. Систем с интеллектуальным интерфейсом. 2. Самообучающихся систем. 3. Адаптивных систем. 4. Экспертных систем.		
2	Упорядочите по убыванию приоритетов следующую последовательность логических операций.	1. Отрицание. 2. Дизъюнкция. 3. Конъюнкция. 4. Эквивалентность.		
3	Заданная таблица истинности соответствует операции: X Y ? И И И И Л Л Л И Л Л Л И	1. Дизъюнкции. 2. Импликации. 3. Эквивалентности. 4. Исключающему ИЛИ. 5. Конъюнкции.		
4	Противоречивая формула - это:	1. Формула, истинная во всех интерпретациях. 2. Формула, истинная хотя бы в одной интерпретации. 3. Формула, ложная во всех интерпретациях. 4. Формула, ложная хотя бы в одной интерпретации.		
5***	Истинностными значениями формулы $P \vee Q \leftrightarrow \neg P$ при заданном множестве значений P и Q: И И И Л Л И Л Л – являются:	1.И 2.И 3.Л 4.Л 5.И И И Л И Л Л Л И Л Л Л И Л И Л		
6	Свойство полноты аксиоматической системы означает:	1. Любая выводимая формула есть тавтология. 2. Любая тавтология выводима из системы аксиом. 3. Никакая аксиома не выводима из других аксиом. 4. Невозможно вывести отрицание тавтологии.		
7*	…..– это динамические ЭС, основанные на интеграции разнородных источников знаний, которые обмениваются между собой полученными результатами в процессе решения задач.	Вставить название типа ЭС.		
8*	…..означает, что между информационными единицами задают отношения релевантности, которые характеризуют ситуационную близость информационных единиц, то есть силу ассоциативной связи между информационными единицами.	Вставить название характеристики знаний.		
9***	Формула $\forall x \exists y\ (P(x,y) \rightarrow Q(f(x),a))$ в заданной интерпретации: D={0,1}.Оценка для a: a=1. Оценки для f: f(0)=0; f(1)=1.	1. Истинна. 2. Ложна.		

	Оценки для P и Q: P(0,0)=И;P(0,1)=И;P(1,0)=И;P(1,1)=Л;Q(0,1)=Л;Q(1,1)=И;			
10	Следующая фраза языка: «Не все целые числа являются рациональными числами»,где *C(x)* - предикат «*x* – есть целое число», *R(x)* - предикат «*x* – есть рациональное число» соответствует формуле:	*1. ¬ ∀x (¬C(x)→R(x)).* *2. ∀x (¬C(x)→R(x)).* *3. ¬ ∀x (C(x)→¬R(x)).* *4. ¬(∀x)(C(x)→R(x)).*		
11***	Пренексная нормальная форма формулы *¬(∀xP(x)→∃xQ(x))* - это.	1. *∃x (P(x)∧¬Q(x)).* 2. *∀x∃y (P(x)∨¬Q(y)).* 3. *∃x ∀y (P(x)∨¬Q(y)).* 4. *∃x (P(x)∧¬Q(x)).* 5. *∀x (P(x)∧¬Q(x)).*		
12***	Скулемовская стандартная форма формулы *∀z∃x ∀y∃w¬(P(x,y,a)∨Q(z,w,f(x,y)))* – это:	*1. ∀y ∀z (¬P(g(z),y,a)∧¬Q(z,h(z,y)),f(g(z),y))).* *2. ∀y ∀z ¬(P(g(z),y,a)∨Q(z,g(z,y),f(g(z),y))).* 3. *∀y ∀z (¬P(f(z),y,a)∧¬Q(z,f(z),f(f(z),y))).* 4. *∀y ∀z ¬(P(f(z),y,a)∨Q(z,f(z),f(f(z),y))).* *5. ∀y ∀z (¬P(b,y,a)∧¬Q(z,c,f(b,y))).*		
13	Переменная и функция унифицируются по следующему правилу:	1. Унифицируются всегда, и вместо переменной подставляется функция. 2. Унифицируются всегда, и вместо функции подставляется переменная. 3. Не унифицируются никогда. 4. Унифицируются, если переменная не является термом функции, тогда вместо переменной подставляется функция.		
14*	Пусть даны два дизъюнкта C_1 и C_2. Тогда ….. С дизъюнктов C_1 и C_2 есть логическое следствие C_1 и C_2.	Вставить пропущенное слово.		
15*	Формула *F* находится в … форме, если она имеет вид: $F_1 \vee F_2 \vee ... \vee F_n$, $n >= 1$, где каждая из $F_1, F_2, ..., F_n$ есть конъюнкция литералов.	Вставить название формы.		
16***	Следующее множества дизъюнктов: 1. *P(x,f(x),b).* 2. *¬S(x)∨ ¬P(x,f(y),z)∨S(z).* 3. *S(a).* 4. *S(b).*	1. Противоречиво. 2. Непротиворечиво.		
17*	Факт – это предложение языка Пролог, которое имеет…:	Вставить текст, определяющий структуру факта.		
18*	Раздел программы … на языке Visual Prolog содержит запрос к программе:	Вставить название раздела.		
19	Следующий фрагмент программы иллюстрирует метод…: *clauses* *repeat.* *repeat:- repeat.* *do_echo:- repeat, readchar (D), write(D), nl, char_int (D,13).* *goal* *do_echo.*	1. Обобщенное рекурсивное правило. 2. Повтора, использующего бесконечную рекурсию. 3. Использования механизма возврата. 4. Возврата после неудачи.		
20	Следующий предикат на языке	1. Определение всех вхождений элемента в		

	Пролог производит: *predicates* *p4(list, list)* *p4(list, list, list)* *clauses* *p4(L1,L2):-p4(L1,[],L2).* *p4([Head|Tail],L,L2):-* *p4(Tail,[Head|L],L2).* *p4([],L,L).*	список. 2. Удаление всех вхождений элемента из списка. 3. Удаление первого вхождения элемента из списка. 4. Объединение двух списков 5. Инвертирование списка.		
21	Стратегия поиска решения задачи «в ширину» на языке Пролог обеспечивает:	1. Простоту программирования и минимальные затраты памяти. 2. Получение первым решающего пути, минимальной длины. 3. Получение первым решающего пути, минимальной длины и простоту программирования.		
22**	Установить соответствие четырёх приведённых примеров с двумя понятиями: а) оптимизированная хвостовая рекурсия; б) неоптимизированная хвостовая рекурсия.	*1. count1(50).* *count1(N):-write(N),nl,N1=N+1,count1(N1).* *goal* *nl, count1(0).* *2. count2(50).* *count2(N):-write(N),N1=N+1,count2(N1),nl.* *goal* *count2(0).* *3. count3(50).* *count3(N):-* *N>0,!,write(N),N1=N+1,count3(N1).* *count3(N):- write("N – отрицательное число").* *goal* *nl, count3(0).* *4. count4(50).* *count4(N):-* *write(N),nl,N1=N+1,check(N1),count4(N1).* *check(Z):-Z>=0.* *check(Z):-Z<0.* *goal* *nl, count4(0).*		
23***	Найти наиболее общий унификатор (если он существует) следующего множества дизъюнктов: *{Q(a,x,f(x),c), Q(a,b,f(y),y)}.*			
24***	Привести к скулемовской стандартной форме формулу: $\forall z P(z) \wedge \forall x(\forall y Q(x,y) \leftrightarrow \exists v Q(x,v))$.			
25***	Написать программу удаления из исходного списка N элементов, начиная с M-го элемента списка.			
ИТОГО				

Примечание:

1. **Вопросы, отмеченные * требуют вставки ответа в текст вопроса,**
2. **Вопросы, отмеченные ** предполагают разделение ответов на два указанных класса,**
3. **Вопросы, отмеченные *** требуют приложения к тесту решения примера.**

Общее число баллов за тест: 40. Общее число баллов за лабораторные работы:40.

Итоговая оценка_____

Тесты рассмотрены и утверждены на заседании кафедры 23.10.2012 г.

Заведующий кафедрой ИСТ, С.А. ПРОХОРОВ
д.т.н., профессор

Составитель, к.т.н.,
доцент кафедры ИСТ И.В. ЛЁЗИНА

Примеры заданий для проведения коллоквиумов

САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ УНИВЕРСИТЕТ имени академика С.П. КОРОЛЕВА

БИЛЕТ ДЛЯ КОЛЛОКВИУМА № 1

По дисциплине *СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА*

Семестр 8 Факультет 6 Направление 230100.62

1. Характеристики знаний. Логические, сетевые, продукционные и фреймовые модели представления знаний.
2. Оценить значение формулы $\forall x \exists y(Q(x,f(a)) \vee P(f(x),y))$ исчисления предикатов первого порядка в заданной области интерпретации.
   Область: D={1, 2}.
   Оценка для a: a=1.
   Оценки для f: f(2)=2, f(1)=1.
   Оценки для P и Q:

P(1, 1)	*P(1, 2)*	*P(2, 1)*	*P(2, 2)*	*Q(1, 1)*	*Q(2, 1)*
И	*Л*	*Л*	*Л*	*И*	*Л*

3. Написать программу замены элементов списка на заданную константу с N+1-ой позиции и до конца списка.

Билет рассмотрен и утвержден на заседании кафедры . .2012 г.

Заведующий кафедрой ИСТ **С.А. ПРОХОРОВ**

Составитель, доцент кафедры ИСТ

О.П. СОЛДАТОВА

САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ УНИВЕРСИТЕТ имени академика С.П. КОРОЛЕВА

БИЛЕТ ДЛЯ КОЛЛОКВИУМА № 2
По дисциплине *СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА*

Семестр 8 Факультет 6 Направление 230100.62

1. Исчисление высказываний. Алфавит и правила построения формул. Правила вывода в исчислении высказываний. Понятие аксиоматической системы. Свойства аксиоматических систем.
2. Привести к пренексной нормальной форме формулу: $\forall x \exists y P(x,y) \rightarrow (\exists z Q(z) \rightarrow \forall z G(z))$.
3. Написать программу замены *N*-го элемента в списке на заданную константу.

Билет рассмотрен и утвержден на заседании кафедры . .2012 г.

Заведующий кафедрой ИСТ **С.А. ПРОХОРОВ**

Составитель, доцент кафедры ИСТ
О.П. СОЛДАТОВА

САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ УНИВЕРСИТЕТ имени академика С.П. КОРОЛЕВА

БИЛЕТ ДЛЯ КОЛЛОКВИУМА № 3
По дисциплине *СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА*

Семестр 8 Факультет 6 Направление 230100.62

1. Интерпретация формул в исчислении высказываний. Общезначимые и противоречивые формулы. Определение логического следствия. Основные теоремы доказательства логического следствия.
2. Привести к скулемовской стандартной форме формулу: $\forall x(\neg \forall y P(x,y)) \rightarrow (\exists z Q(z) \wedge \exists y R(y)))$.
3. Написать программу удаления первого вхождения элемента в списке.

Билет рассмотрен и утвержден на заседании кафедры . .2012 г.

Заведующий кафедрой ИСТ **С.А. ПРОХОРОВ**

Составитель, доцент кафедры ИСТ
О.П. СОЛДАТОВА

САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ УНИВЕРСИТЕТ имени академика С.П. КОРОЛЕВА

БИЛЕТ ДЛЯ КОЛЛОКВИУМА № 4
По дисциплине *СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА*

Семестр 8 Факультет 6 Направление 230100.62

1. Исчисление предикатов первого порядка. Алфавит. Понятие терма. Определение предиката. Свободные и связанные переменные. Правила построения формул в исчислении предикатов.
2. Проверить невыполнимость следующего множества дизъюнктов, используя правило резолюций:
   $P(x, y, w) \vee \neg P(x, z, v) \vee \neg P(a, b, c)$.
   $\neg P(a, b, c)$.
   $P(c, x, x)$.
   $P(x, x, c)$.
3. Написать программу замены всех вхождений элемента списка на заданную константу.

Билет рассмотрен и утвержден на заседании кафедры . .2012 г.

Заведующий кафедрой ИСТ **С.А. ПРОХОРОВ**

Составитель, доцент кафедры ИСТ
О.П. СОЛДАТОВА

САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ УНИВЕРСИТЕТ имени академика С.П. КОРОЛЕВА

БИЛЕТ ДЛЯ КОЛЛОКВИУМА № 5

По дисциплине *СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА*

Семестр 8 Факультет 6 Направление 230100.62

1. Интерпретация формул в логике предикатов первого порядка. Общезначимые и противоречивые формулы в исчислении предикатов. Системы аксиом логики предикатов. Правила вывода в исчислении предикатов.
2. Найти наиболее общий унификатор (если он существует) следующего множества дизъюнктов:
   {Q(x,h(a,y),z), Q(u,h(v,b),c).
3. Написать программу создания нового списка, путем переписывания всех элементов исходного списка, начиная с *N*-ой позиции.

Билет рассмотрен и утвержден на заседании кафедры . .2012 г.

Заведующий кафедрой ИСТ **С.А. ПРОХОРОВ**

Составитель, доцент кафедры ИСТ

О.П. СОЛДАТОВА

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Самарский государственный аэрокосмический
университет имени академика С.П. Королёва
(национальный исследовательский университет)»

СОГЛАСОВАНО

Управление образовательных программ

_____/ А.В. Дорошин /

"_____"_____ 20___ г.

УТВЕРЖДАЮ

Проректор по учебной работе

_____/ Ф.В. Гречников /

"____" _____ 20___ г.

РАБОЧАЯ ПРОГРАММА

Наименование модуля (дисциплины)

Системы искусственного интеллекта

Цикл, в рамках которого происходит освоение модуля (дисциплины)	Общепрофессиональные дисциплины
Часть цикла	Вариативная часть
Код учебного плана	230100.262-2011-О-П-4г00м
Факультет	Информатики
Кафедра	Инф. систем и технологий

Курс	4		
Семестр	8		
Лекции (СЛ)	36		
Семинарские и практические занятия (СП)	0		
Лабораторные занятия (СЛР)	48	Экзамен	8
Контроль самастоятельной работы (КСР)	0	Зачет	
Самостоятельная работа (СРС)	24		
Всего	108		

Наименование стандарта, на основании которого составлена рабочая программа:

230100 "Информатика и вычислительная техника"

Соответствие содержания рабочей программы, условий ее реализации, материально-технической и учебно-методической обеспеченности учебного процесса по дисциплине всем требованиям государственных стандартов подтверждаем.

Составители:

Солдатова Ольга Петровна, доц., к.т.н. _____
(подпись)

Заведующий кафедрой:

Прохоров Сергей Антонович, проф., д.т.н. _____
(подпись)

Рабочая программа обсуждена на заседании кафедры

Инф. систем и технологий

Протокол №___ от "____" _____ 20___ г.

Наличие основной литературы в фондах научно-технической библиотеки (НТБ) подтверждаем:

Директор НТБ _____ /_____/
(подпись) (расшифровка подписи)

Согласовано:

Декан _____ /_____/
(подпись) (расшифровка подписи)

1 Цели и задачи модуля (дисциплины), требования к уровню освоения содержания

1.1 Перечень развиваемых компетенций

ОК-1, ОК-2, ОК-6, ОК-10, ОК-11, ОК-12, ОК-13
ПК-2, ПК-3, ПК-4, ПК-5

1.2 Цели и задачи изучения модуля (дисциплины)

Целью дисциплины является приобретение студентами теоретических знаний в области разработки систем искусственного интеллекта и практических навыков логического программирования с использованием языка Prolog.

1.3 Требования к уровню подготовки студента, завершившего изучение данного модуля (дисциплины)

Студенты, завершившие изучение данной дисциплины, должны
знать: основные модели представления знаний, основные стратегии решения задач, приемы логического программирования и основные принципы, конструкции и лексику языка Prolog;
Студенты, завершившие изучение данной дисциплины, должны знать: основные модели представления знаний, основные стратегии решения задач, приемы логического программирования и основные принципы, конструкции и лексику языка Prolog;уметь: применять теоретические знания в области логического программирования на языке Prolog к решению конкретных инженерных и исследовательских задач.

1.4 Связь с предшествующими модулями (дисциплинами)

Для успешного усвоения материала дисциплины студенты должны обладать знаниями и навыками, полученными в ходе изучения следующих дисциплин:- информатика;- дискретная математика;- математическая логика и теория алгоритмов;- программирование на языке высокого уровня.

1.5 Связь с последующими модулями (дисциплинами)

Дисциплина является базовой для изучения курса «Интеллектуальные системы» магистратуры, а также является основой для разработки программных систем в ходе выполнения курсовых проектов и на этапе выполнения выпускной квалификационной работы бакалавра.

2 Содержание рабочей программы (модуля)

Семестр 1		
СЛ 0,3333 36 часов 0,9999 ЗЕТ	Активные 0	
	Интерактивные 0	

	Традиционные 1	История развития искусственного интеллекта. Классификация систем искусственного интеллекта.
		Характеристики знаний, основные модели представления знаний.
		Исчисление предикатов первого порядка как логическая система представления знаний.
		Автоматизация доказательства теорем в логике предикатов.
		Основы языка логического программирования Prolog.
		Унификация в Prolog. Вычисление цели и механизм возврата. Управление поиском решений.
		Структура программы на языке Prolog. Использование составных термов и списков.
		Повторение и рекурсия в Prolog. Оптимизация хвостовой рекурсии.
		Использование строк в Prolog. Преобразование данных.
		Представление бинарных деревьев на языке Prolog.
		Представление графов на языке Prolog.
		Основные стратегии решения задач. Поиск решения в пространстве состояний. Алгоритм A*.
		Решение игровых задач в терминах И/ИЛИ-графов. Минимаксный принцип поиска решения.
		Основы объектно-ориентированного программирования в среде Visual Prolog.
		Основы визуального программирования в среде Visual Prolog.
СП 0 0 часов 0 ЗЕТ	Активные 0	
	Интерактивные 0	
	Традиционные 0	
СЛР 0,4444 48 часов 1,3332 ЗЕТ	Активные 0,75	Изучение механизма вычисления цели, механизма возврата, рекурсии и работа со списками
		Решение логической головоломки.

		Изучение составных термов и создание простых баз данных.
		Реализация простых запросов к базе данных.
		Реализация запросов с использованием механизма внутренней динамической базы данных.
		Изучение методов представления бинарных деревьев и графов на языке Prolog и алгоритмов поиска решений.
		Решение задач с использованием бинарных деревьев или графов.
		Изучение основ объектно-ориентированного программирования в среде Visual Prolog.
		Изучение основ визуального программирования в среде Visual Prolog.
	Интерактивные 0,25	Изучение структуры программы на Prolog и основных конструкций языка.
		Коллоквиум 1.
		Коллоквиум 2.
	Традиционные 0	
КСР 0 0 часов 0 ЗЕТ	Активные 0	
	Интерактивные 0	
	Традиционные 0	
СРС 0,2222 24 часов 0,6666 ЗЕТ	Активные 0,75	Подготовка к ЛР 1.
		Подготовка к ЛР 2.
		Подготовка к ЛР 3.
		Подготовка к ЛР 4.
		Подготовка к ЛР 5.
		Подготовка к ЛР 6.
		Подготовка к ЛР 8.
		Подготовка к ЛР 9.
		Подготовка к ЛР 10.
		Подготовка к ЛР 11.
	Интерактивные 0	Подготовка к коллоквиуму 1.
		Подготовка к коллоквиуму 2.
		Подготовка к сдаче экзамена.
	Традиционные 0,25	

3 Инновационные методы обучения

1. Использование мультимедийного оборудования при проведении лекционных занятий.
2. Использование при самостоятельной подготовке электронных средств коммуникаций, в том числе специализированных сайтов и форумов.
3. Выполнение лабораторных работ с помощью современного программного обеспечения.
4. Использование тестирования для оценки знаний студентов.
5. Применение рейтинговой системы оценки знаний студентов.

4 Технические средства и материальное обеспечение учебного процесса

1. Презентационные материалы лекций в виде мультимедиа презентаций.
2. Мультимедийный проектор для демонстрации презентаций.
3. Компьютерный класс, используемый для проведения лабораторных занятий.
4. Программное обеспечение, используемое при проведении лабораторных занятий.

5 Учебно-методическое обеспечение

5.1 Основная литература

1. Адаменко, Анатолий Николаевич. Логическое программирование и Visual Prolog [Текст] : [наиболее полн. рук.] / А. Н. Адаменко, А. Кучуков. - СПб. : БХВ-Петербург, 2003. - 990 с. + 1 эл. опт. диск (CD-ROM). - (В подлиннике). - ISBN 5-94157-156-9 : 258.63
р.- 2 экз.
2. Братко, Иван. Алгоритмы искусственного интеллекта на языке PROLOG [Текст] : [пер. с англ.] / Иван Братко ; Фак. компьютер. наук и информатики Люблян. ун-та, Ин-т Йожефа Штефана. - М. [и др.] : Вильямс, 2004. - 637 с. - ISBN 5-8459-0664-4 : 538.36 р.- 3
3. Люгер, Джордж Ф.Искусственный интеллект: стратегии и методы решения сложных проблем [Текст] : [пер. с англ.] / Джордж Ф. Люгер. - 4-е изд. - М. [и др.] : Вильямс, 2005. - 863 с. - ISBN 5-8459-0437-4 : 746.09 р.- 15 экз.

5.2 Дополнительная литература

1. Андрейчиков, Александр Валентинович. Интеллектуальные информационные системы [Текст] : [учеб. для вузов по специальности "Прикладная информатика в экономике"] / А. В. Андрейчиков, О. Н. Андрейчикова. - М. : Финансы и статистика, 2006. - 423 с. - ISBN

5-279-02568-2 : 253.63 р., 233.00 р.- 5 экз.
2. Программирование на языке ПРОЛОГ [Текст] : [метод. указания к лаб. работам] / О. П. Солдатова, И. В. Лезина ; Федер. агентство по образованию, Самар. гос. аэрокосм. ун-т им. С. П. Королева. - Самара : Изд-во СГАУ, 2008. - 51 с. - 122.50 р.- 45 экз.

5.3 Электронные источники и интернет ресурсы

1. Солдатова О.П., Лёзина И.В. Логическое программирование. Учебное пособие [Электронный ресурс]. – http://www.ssau.ru/files/resources/sotrudniki/soldatova_lezina.pdf.
2.Логическое программирование и Visual Prolog [Электронный ресурс. мпакт-диск]/ А. Н. Адаменко, А. Кучуков. - СПб. : БХВ-Петербург, 2003. - 1 CD-ROM. Шифр 004/А 28
3. Электронный аналог. Программирование на языке ПРОЛОГ : [метод. указания к лаб. работам] / О. П. Солдатова, И. В. Лезина. - Самара : Изд-во СГАУ, 2008 on-line (Шифр СГАУ:004/П 784-784587)

5.4 Методические указания и рекомендации

Текущий контроль знаний студентов в семестре проводится в виде коллоквиумов и отчётов по лабораторным работам. Результатом текущего контроля является допуск или недопуск студента к экзамену по дисциплине. Основанием для допуска к экзамену является выполнение и отчет студента по всем лабораторным работам.Промежуточный контроль знаний студентов проводят в семестре в виде экзамена. Экзамен проводится согласно положению о текущем и промежуточном контроле знаний студентов, утвержденному ректором университета. Экзаменационная оценка ставится на основании письменного и устного ответов студента по экзаменационному билету.