МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
БОРИСОГЛЕБСКИЙ ФИЛИАЛ
(БФ ФГБОУ ВО «ВГУ»)

МЕТОДИЧЕСКИЕ МАТЕРИАЛЫ ПО УЧЕБНОЙ ДИСЦИПЛИНЕ
Основы искусственного интеллекта

Методические указания для обучающихся по освоению дисциплины

Приступая к изучению учебной дисциплины, целесообразно ознакомиться с учебной программой дисциплины, электронный вариант которой размещён на сайте БФ ВГУ.

Это позволит обучающимся получить четкое представление о:

- перечне и содержании компетенций, на формирование которых направлена дисциплина;
- основных целях и задачах дисциплины;
- планируемых результатах, представленных в виде знаний, умений и навыков, которые должны быть сформированы в процессе изучения дисциплины;
- количестве часов, предусмотренных учебным планом на изучение дисциплины, форму промежуточной аттестации;
- количестве часов, отведенных на контактную и самостоятельную работу;
- формах контактной и самостоятельной работы;
- структуре дисциплины, основных разделах и темах;
- системе оценивания учебных достижений;
- учебно-методическом и информационном обеспечении дисциплины.

Знание основных положений, отраженных в рабочей программе дисциплины, поможет обучающимся ориентироваться в изучаемом курсе, осознавать место и роль изучаемой дисциплины в подготовке выпускника, строить свою работу в соответствии с требованиями, заложенными в программе.

Основными формами контактной работы по дисциплине являются лекции и лабораторные занятия, посещение которых обязательно для всех студентов (кроме студентов, обучающихся по индивидуальному плану).

Подготовка к лабораторным работам ведется на основе планов лабораторных работ, которые размещены на сайте кафедры. В ходе подготовки к лабораторным работам необходимо изучить в соответствии с вопросами для повторения конспекты лекций, основную литературу, ознакомиться с дополнительной литературой. Кроме того, следует повторить материал лекций, ответить на контрольные вопросы, изучить образцы выполнения заданий лабораторных работ, выполнить задания для самостоятельной работы.

При подготовке к промежуточной аттестации (зачёт с оценкой) необходимо повторить пройденный материал в соответствии с учебной программой, выполнить все задания для самостоятельной работы в лабораторных работах. Рекомендуется использовать конспекты лекций и источники, перечисленные в списке литературы, а также ресурсы электронно-библиотечных систем, представленные в рабочей программе дисциплины.

Методические материалы для обучающихся по освоению теоретических вопросов дисциплины

Тема. Базовые понятия ИИ. Основные направления исследования в области ИИ

План

1. Цель преподавания дисциплины. Терминология: понятия интеллекта, системы знаний, интеллектуальных задач.

2. Философские аспекты проблемы систем ИИ (возможность существования, безопасность, полезность).

3. История развития систем ИИ: основные направления и разработки в области ИИ, современные разработки систем ИИ.

1. Цель преподавания дисциплины. Терминология: понятия интеллекта, системы знаний, интеллектуальных задач

Идея создания ИИ связана с постоянным стремлением человека переложить решение сложных задач на механического, а затем и электронного помощника. Единственный способ реализовать этого заключается в моделировании с помощью различных средств интеллектуальных способностей человека. При этом компьютер берёт на себя не только однотипные, многократно повторяющиеся операции, но и сам может обучаться.

Целью изучения дисциплины ОИИ является овладение систематизированными знаниями об основных моделях, методах, средствах и языках, используемых при разработке систем искусственного интеллекта.

Основным предметом изучения дисциплины являются мыслительные способности человека и способы их реализации техническими средствами.

Основные понятия

Термин «***интеллект***» (англ. – intelligence) происходит от латинского intellectus , что означает ум, рассудок, разум; мыслительные способности человека. Соответственно термин «***искусственный интеллект***» (artificial intelligence) обычно толкуется как свойство автоматических систем брать на себя отдельные функции интеллекта человека. Например, принимать оптимальные решения на основе ранее полученного опыта и рационального анализа внешних воздействий.

Мы же ***интеллектом (или естественным интеллектом ЕИ)*** будем называть способность мозга решать ***интеллектуальные задачи*** путем приобретения, запоминания и целенаправленного преобразования ***знаний*** в процессе обучения на опыте и адаптации к разнообразным обстоятельствам.

Разберем это определение более подробно.

Начнем с термина «ЗНАНИЯ». Под этим термином подразумевается не только информация, поступающая в мозг через органы чувств. Несомненно, такого типа знания чрезвычайно важны, но их недостаточно для интеллектуальной деятельности. Дело в том, что объекты окружающей среды обладают свойством не только воздействовать на органы чувств, но и находиться в определённых отношениях друг с другом. Для того чтобы осуществлять в окружающей среде интеллектуальную деятельность или хотя бы просто существовать, необходимо иметь в системе знаний модель этого мира. В этой информационной модели окружающей среды реальные объекты, их свойства и отношения между ними не только отображаются и запоминаются, но и могут мысленно "целенаправленно преобразовываться". При этом существенно то, что формирование модели внешней среды происходит "в процессе обучения на опыте и адаптации к разнообразным обстоятельствам".

Далее остановимся на термине «ЗАДАЧА». В качестве исходного можно принять понимание задачи как мыслительной задачи, существующее в психологии. Психологи подчеркивают, что задача есть только тогда, когда есть работа для мышления, т. е. когда имеется некоторая цель, а средства к ее достижению не ясны; их надо найти посредством мышления. Очень уместным в этом случае является высказывание Д. Пойи: «...трудность решения в какой-то мере входит в само понятие задачи: там, где нет трудности, нет и задачи». Если человек имеет очевидное средство, с помощью которого можно осуществить желание, поясняет он, то задачи не возникает. Если человек обладает алгоритмом решения некоторой задачи и имеет физическую возможность его реализации, то задачи в собственном смысле уже не существует.

Перейдем к интеллектуальным задачам.

Для того чтобы пояснить, чем отличается интеллектуальная задача от просто задачи, необходимо ввести термин *алгоритм* – один из основных терминов кибернетики.

Под ***алгоритмом*** понимают точное предписание о выполнении в определённом порядке системы операций для решения любой задачи из некоторого данного класса задач. Термин "алгоритм" происходит от имени персидского математика Аль-Хорезми, который ещё в IX веке предложил простейшие арифметические алгоритмы. В

математике и кибернетике класс задач определённого типа считается решённым, когда для их решения установлен алгоритм. Нахождение алгоритмов является естественной целью человека при решении им разнообразных классов задач. Отыскание алгоритма для задач некоторого данного типа связано с тонкими и сложными рассуждениями, требующими большой изобретательности и высокой квалификации. Принято считать, что подобного рода деятельность требует участия интеллекта человека. Задачи, связанные с отысканием алгоритма решения класса задач определённого типа, будем называть ***интеллектуальными***.

Так понимаемая задача в сущности тождественна проблемной ситуации, и решается она посредством преобразования последней. В ее решении участвуют не только условия, которые непосредственно заданы. Человек использует любую находящуюся в его памяти информацию, «модель мира», имеющуюся в его психике и включающую фиксацию разнообразных законов, связей, отношений этого мира.

Что же касается задач, алгоритмы решения которых уже установлены, то, как отмечает известный специалист в области ИИ М.Минский, "излишне приписывать им такое мистическое свойство, как "интеллектуальность". В самом деле, после того, как такой алгоритм уже найден, процесс решения соответствующих задач становится таким, что его могут в точности выполнить человек, вычислительная машина или робот, не имеющие ни малейшего представления о сущности самой задачи. Требуется только, чтобы лицо, решающее задачу, было способно выполнять те элементарные операции, из которых складывается процесс, и, кроме того, чтобы оно педантично и аккуратно руководствовалось предложенным алгоритмом. Такое лицо, действуя, как говорят в таких случаях, чисто машинально, может успешно решать любую задачу рассматриваемого типа.

Поэтому представляется совершенно естественным исключить из класса интеллектуальных такие задачи, для которых существуют стандартные методы решения. Примерами таких задач могут служить чисто вычислительные задачи: решение системы линейных алгебраических уравнений, численное интегрирование дифференциальных уравнений и т.д. Для решения подобного рода задач имеются стандартные алгоритмы, представляющие собой определённую последовательность элементарных операций, которая может быть легко реализована в виде программы для вычислительной машины. В противоположность этому для широкого класса интеллектуальных задач, таких, как распознавание образов, игра в шахматы, доказательство теорем и т.п., напротив, это формальное разбиение процесса поиска решения на отдельные элементарные шаги часто оказывается весьма затруднительным, даже если само их решение несложно.

Таким образом, мы можем перефразировать определение интеллекта как *универсальный сверхалгоритм, который способен создавать алгоритмы решения конкретных задач*.

Исходя из вышесказанного, интересно то, что профессия программиста является одной из самых интеллектуальных, поскольку продуктом деятельности программиста являются программы – алгоритмы в чистом виде. Именно поэтому создание даже элементов ИИ должно очень сильно повысить производительность его труда.

Деятельность мозга (обладающего интеллектом), направленную на решение интеллектуальных задач, мы будем называть ***мышлением***, или интеллектуальной деятельностью. Интеллект и мышление органически связаны с решением таких задач, как доказательство теорем, логический анализ, распознавание ситуаций, планирование поведения, игры и управление в условиях неопределённости.

Характерными чертами интеллекта, проявляющимися в процессе решения задач, являются способность к обучению, обобщению, накоплению опыта (знаний и навыков) и адаптации к изменяющимся условиям в процессе решения задач. Благодаря этим качествам интеллекта мозг может решать разнообразные задачи, а также легко перестраиваться с решения одной задачи на другую. Таким образом, мозг,

наделённый интеллектом, является универсальным средством решения широкого круга задач (в том числе неформализованных) для которых нет стандартных, заранее известных методов решения.

Следует иметь в виду, что существуют и, чисто *поведенческие* (функциональные) определения ИИ. Так, по А.Н. Колмогорову, любая материальная система, с которой можно достаточно долго обсуждать проблемы науки, литературы и искусства, обладает интеллектом. Другим примером поведенческой трактовки интеллекта может служить известное определение (или тест) А.Тьюринга. В 1950 году британский математик Алан Тьюринг опубликовал в журнале «Mind» свою работу «Вычислительные машины и интеллект», в которой описал тест для проверки программы на интеллектуальность. Он предложил поместить исследователя и программу в разные комнаты и до тех пор, пока исследователь не определит, кто за стеной – человек или программа, считать поведение программы разумным. Это было одно из первых определений интеллектуальности программы, т. е. Тьюринг предложил называть интеллектуальным такое поведение программы, которое будет моделировать разумное поведение человека.

Также, интересен план имитации мышления, предложенный А.Тьюрингом. "Пытаясь имитировать интеллект взрослого человека, – пишет Тьюринг, – мы вынуждены много размышлять о том процессе, в результате которого человеческий мозг достиг своего настоящего состояния… Почему бы нам вместо того, чтобы пытаться создать программу, имитирующую интеллект взрослого человека, не попытаться создать программу, которая имитировала бы интеллект ребенка? Ведь если интеллект ребенка получает соответствующее воспитание, он становится интеллектом взрослого человека… Наш расчёт состоит в том, что устройство, ему подобное, может быть легко запрограммировано… Таким образом, мы расчленим нашу проблему на две части: на задачу построения "программы-ребенка" и задачу "воспитания" этой программы".

Именно этот путь используют практически все системы ИИ. Ведь понятно, что практически невозможно заложить все знания в достаточно сложную систему. Кроме того, только проходя этот путь, можно проявить перечисленные выше признаки интеллектуальной деятельности (накопление опыта, адаптация и т. д.).

2. Философские аспекты проблемы систем ИИ (возможность существования, безопасность, полезность)

Основная философская проблема в области ИИ – возможность или не возможность моделирования мышления человека. В случае если когда-либо будет получен отрицательный ответ на этот вопрос, то все остальные вопросы не будут иметь не малейшего смысла.

Следовательно, начиная исследование ИИ, заранее предположим положительный ответ. Приведем несколько соображений, которые подводят нас к данному ответу.

1. Первое доказательство является схоластическим, и доказывает непротиворечивость ИИ и Библии. Даже люди далекие от религии, знают слова священного писания: "И создал Господь человека по образу и подобию своему …". Исходя из этих слов, мы можем заключить, что, поскольку Господь, во-первых, создал нас, а во-вторых, мы по своей сути подобны ему, то мы вполне можем создать кого-то по образу и подобию человека.

2. Создание нового разума биологическим путем для человека дело вполне привычное. Дети большую часть знаний приобретают путем обучения, а не как заложенную в них заранее.

3. То, что раньше казалось вершиной человеческого творчества - игра в шахматы, шашки, распознавание зрительных и звуковых образов, синтез новых технических решений, на практике оказалось не таким уж сложным делом (теперь

работа ведется не на уровне возможности или невозможности реализации перечисленного, а о нахождении наиболее оптимального алгоритма).

4. С проблемой воспроизведения своего мышления тесно смыкается проблема возможности самовоспроизведения. Существуют также различные неформальные доказательства возможности самовоспроизведения, но самым ярким доказательством, пожалуй, будет существование компьютерных вирусов.

5. Принципиальная возможность автоматизации решения интеллектуальных задач с помощью ЭВМ обеспечивается свойством алгоритмической универсальности. Это означает, что на них можно программно реализовывать любые алгоритмы преобразования информации, - будь то вычислительные алгоритмы, алгоритмы управления, поиска доказательства теорем или композиции мелодий.

Однако не следует думать, что вычислительные машины и роботы могут в принципе решать любые задачи. Анализ разнообразных задач привел математиков к замечательному открытию. Было строго доказано существование таких типов задач, для которых невозможен единый эффективный алгоритм, решающий все задачи данного типа; в этом смысле невозможно решение задач такого типа и с помощью вычислительных машин. Этот факт способствует лучшему пониманию того, что могут делать машины и чего они не могут сделать.

Как же действует человек при решении таких задач? Похоже, что он просто-напросто игнорирует их, что, однако не мешает ему жить дальше. Другим путем является сужение условий универсальности задачи, когда она решается только для определенного подмножества начальных условий. И еще один путь заключается в том, что человек методом "научного тыка" расширяет множество доступных для себя элементарных операций (например, создает новые материалы, открывает новые месторождения или типы ядерных реакций).

Следующим философским вопросом ИИ является цель создания. В принципе все, что мы делаем в практической жизни, обычно направлено на то, чтобы больше ничего не делать. Однако при достаточно высоком уровне жизни человека на первые роли выступает уже не лень, а поисковые инстинкты. Допустим, что человек сумел создать интеллект, превышающий свой собственный. Что теперь будет с человечеством? Какую роль будет играть человек? Для чего он теперь нужен? И вообще, нужно ли в принципе создание ИИ?

По-видимому, самым приемлемым ответом на эти вопросы является концепция "усилителя интеллекта" (УИ). Здесь уместна аналогия с президентом государства - он не обязан знать валентности ванадия или языка программирования Java для принятия решения о развитии ванадиевой промышленности. Каждый занимается своим делом - химик описывает технологический процесс, программист пишет программу; в конце концов, экономист говорит президенту, что вложив деньги в промышленный шпионаж, страна получит 20%, а в ванадиевую промышленность - 30% годовых. При такой постановке вопроса любой человек сможет сделать правильный выбор.

В данном примере президент использует биологический УИ - группу специалистов с их белковыми мозгами. Но уже сейчас используются и неживые УИ - например мы не могли бы предсказать погоду без компьютеров, при полетах космических кораблей с самого начала использовались бортовые счетно-решающие устройства. Кроме того, человек уже давно использует усилители силы (УС) - понятие, во многом аналогичное УИ. В качестве усилителей силы ему служат автомобили, краны, электродвигатели, прессы, пушки, самолеты и многое-многое другое.

Основным отличием УИ от УС является наличие воли. Ведь мы не сможем себе представить, чтобы вдруг серийный "Запорожец" взбунтовался, и стал ездить так, как ему хочется. Не можем представить именно потому, что ему ничего не хочется, у него нет желаний. В тоже время, интеллектуальная система, вполне могла бы иметь свои желания, и поступать не так, как нам хотелось бы. Таким образом перед нами встает еще одна проблема - проблема безопасности.

Проблема безопасности

Данная проблема будоражит умы человечества еще со времен Карела Чапека, впервые употребившего термин "робот". Большую лепту в обсуждение данной проблемы внесли и другие писатели-фантасты. Как самые известные мы можем упомянуть серии рассказов писателя-фантаста и ученого Айзека Азимова, а так же довольно свежее произведение - "Терминатор". Кстати именно у Азимова мы можем найти самое проработанное, и принятое большинством людей решение проблемы безопасности. Речь идет о так называемых трех законах роботехники.

1. Робот не может причинить вред человеку или своим бездействием допустить, чтобы человеку был причинен вред.

2. Робот должен повиноваться командам, которые ему дает человек, кроме тех случаев, когда эти команды противоречат первому закону.

3. Робот должен заботиться о своей безопасности, насколько это не противоречит первому и второму закону.

На первый взгляд подобные законы, при их полном соблюдении, должны обеспечить безопасность человечества. Однако при внимательном рассмотрении возникают некоторые вопросы.

Интересно, что будет подразумевать система ИИ под термином "вред" после долгих логических размышлений? Не решит ли она, что все существования человека это сплошной вред? Ведь он курит, пьет, с годами стареет и теряет здоровье, страдает. Не будет ли меньшим злом быстро прекратить эту цепь страданий? Конечно можно ввести некоторые дополнения, связанные с ценностью жизни, свободой волеизъявления. Но это уже будут не те простые три закона, которые были в исходнике.

Следующим вопросом будет такой. Что решит система ИИ в ситуации, когда спасение одной жизни возможно только за счет другой? Особенно интересны те случаи, когда система не имеет полной информации о том, кто есть кто.

Однако, несмотря на перечисленные проблемы, данные законы являются довольно неплохим неформальным базисом проверки надежности системы безопасности для систем ИИ.

Так что же, неужели нет надежной системы безопасности? Если отталкиваться от концепции УИ, то можно предложить следующий вариант.

Согласно многочисленным опытам, несмотря на то, что мы не знаем точно, за что отвечает каждый отдельный нейрон в человеческом мозге, многим из наших эмоций обычно соответствует возбуждение группы нейронов (нейронный ансамбль) во вполне предсказуемой области. В математической интерпретации это сведение какой-либо функции к максимуму или к минимуму. Теперь представим себе, что наш УИ в качестве такой функции использует измеренную прямо или косвенно, степень удовольствия мозга человека-хозяина. Если принять меры, чтобы исключить самодеструктивную деятельность в состоянии депрессии, а так же предусмотреть другие особые состояния психики, то получим следующее.

Поскольку предполагается, что нормальный человек, не будет наносить вред самому себе, и, без особой на то причины, другим, а УИ теперь является частью данного индивидуума (не обязательно физическая общность), то автоматически выполняются все 3 закона роботехники. При этом вопросы безопасности смещаются в область психологии и правоохранения, поскольку система (обученная) не будет делать ничего такого, чего бы не хотел ее владелец.

Вопрос полезности

И теперь осталась еще одна тема - а стоит ли вообще создавать ИИ, может просто закрыть все работы в этой области? Единственное, что можно сказать по этому поводу - если ИИ возможно создать, то рано или поздно он будет создан. И лучше его создавать под контролем общественности, с тщательной проработкой вопросов безопасности, чем он будет создан лет через 100-150 каким-нибудь программистом-

механиком-самоучкой, использующим достижения современной ему техники. Ведь сегодня, например, любой грамотный инженер, при наличии определенных денежных ресурсов и материалов, может изготовить атомную бомбу.

3. История развития систем ИИ: основные направления и разработки в области ИИ, современные разработки систем ИИ

История искусственного интеллекта как нового научного направления начинается в середине20 в.

В 1954 году американский исследователь А.Ньюэлл решил написать программу для игры в шахматы. Этой идеей он поделился с аналитиками корпорации «РЭНД» Дж. Шоу и Г.Саймоном, которые предложили Ньюэллу свою помощь. В качестве теоретической основы такой программы было решено использовать метод, предложенный в 1950 году Клодом Шенноном, основателем теории информации. Точная формализация этого метода была выполнена Аланом Тьюрингом. Он же промоделировал его вручную.

К работе была привлечена группа голландских психологов, изучавших стили игры выдающихся шахматистов. Через два года совместной работы этим коллективом был создан язык программирования ИПЛ1 - первый символьный язык обработки списков. Вскоре была написана и первая *программа*, которую можно отнести к достижениям в области искусственного интеллекта. Эта была программа "Логик-Теоретик" (1956 г.), предназначенная для автоматического доказательства теорем в исчислении высказываний.

Собственно же программа для игры в шахматы, NSS, была завершена в 1957 г. В основе ее работы лежали так называемые *эвристики* (правила, которые позволяют сделать выбор при отсутствии точных теоретических оснований) и описания целей.

В 1960 г. той же группой, на основе принципов, использованных в NSS, была написана программа, которую ее создатели назвали GPS (General Problem Solver)- универсальный решатель задач. GPS могла справляться с рядом головоломок , вычислять неопределенные интегралы, решать некоторые другие задачи. Эти результаты привлекли внимание специалистов в области вычислений. Появились программы автоматического доказательства теорем из планиметрии и решения алгебраических задач.

Джона Маккарти из Стэнфорда заинтересовали математические основы этих результатов и вообще символьных вычислений. В результате в 1963 г. им был разработан язык ЛИСП (LISP, от List Processing).

В это же время в СССР, в основном, в Московском университете и Академии наук был выполнен ряд исследований, возглавленных Вениамином Пушкиным и Дмитрием Поспеловым, целью которых было выяснение, как же, в действительности, человек решает переборные задачи?

К исследованиям в области искусственного интеллекта стали проявлять интерес и логики. В 1964 году была опубликована работа ленинградского логика Сергея Маслова "Обратный метод установления выводимости в классическом исчислении предикатов", в которой впервые предлагался метод автоматического поиска доказательства теорем в исчислении предикатов.

На год позже (в 1965 г.) в США появляется работа Дж.А.Робинсона, посвященная несколько иному методу автоматического поиска доказательства теорем в исчислении предикатов первого порядка. Этот метод был назван методом резолюций и послужил отправной точкой для создания нового языка программирования со встроенной процедурой логического вывода - языка Пролог (PROLOG) в 1971.

В 1966 году в СССР Валентином Турчиным был разработан язык рекурсивных функций Рефал, предназначенный для описания языков и разных видов их обработки. Хотя он и был задуман как алгоритмический метаязык, но для пользователя это был, подобно ЛИСПу и Прологу, язык обработки символьной информации.

В конце 60-х годов появились первые игровые программы, системы для элементарного анализа текста и решения некоторых математических задач (геометрии, интегрального исчисления). В возникавших при этом сложных переборных проблемах количество перебираемых вариантов резко снижалось применением всевозможных эвристик (*эвристики* - это общие рекомендации или советы, основанные на статистической очевидности, например, "курение сокращает вашу жизнь") и «здравого смысла». Такой подход стали называть *эвристическим программированием*. Дальнейшее развитие эвристического программирования шло по пути усложнения алгоритмов и улучшения эвристик. Однако вскоре стало ясно, что существует некоторый предел, за которым никакие улучшения эвристик и усложнения алгоритма не повысят качества работы системы и, главное, не расширят ее возможностей. Программа, которая играет в шахматы, никогда не будет играть в шашки или карточные игры.

Постепенно исследователи стали понимать, что всем ранее созданным программам недостает самого важного - знаний в соответствующей области. Специалисты, решая задачи, достигают высоких результатов, благодаря своим знаниям и опыту; если программы будут обращаться к знаниям и применять их, то они тоже достигнут высокого качества работы.

Это понимание, возникшее в начале 70-х годов, по существу, означало качественный скачок в работах по искусственному интеллекту.

Уже к середине 70-х годов появляются первые прикладные интеллектуальные системы, использующие различные способы представления знаний для решения задач - *экспертные системы*. Одной из первых была экспертная система DENDRAL, разработанная в Станфордском университете и предназначенная для порождения формул химических соединений на основе спектрального анализа. В настоящее время DENDRAL поставляется покупателям вместе со спектрометром. Система MYCIN предназначена для диагностики и лечения инфекционных заболеваний крови. Система PROSPECTOR прогнозирует залежи полезных ископаемых. Имеются сведения о том, что с ее помощью были открыты залежи молибдена, ценность которых превосходит 100 миллионов долларов. Система оценки качества воды, реализованная на основе российской технологии SIMER + MIR несколько лет назад выявившая причины превышения предельно допустимых концентрациий загрязняющих веществ в Москве-реке в районе Серебрянного Бора. Система CASNET предназначена для диагностики и выбора стратегии лечения глаукомы и т.д.

В настоящее время банки применяют системы искусственного интеллекта (СИИ) в страховой деятельности при игре на бирже и управлении собственностью. В августе 2001 года роботы выиграли у людей в импровизированном соревновании по трейдингу. Методы распознавания образов широко используют при оптическом и акустическом распознавании (в том числе текста и речи), медицинской диагностике, спам-фильтрах, в системах ПВО.

Разработчики компьютерных игр вынуждены применять ИИ той или иной степени проработанности. Стандартными задачами ИИ в играх являются нахождение пути в двухмерном или трёхмерном пространстве, имитация поведения боевой единицы, расчёт верной экономической стратегии и так далее.

Тема. Системы знаний. Модели представления знаний

План

1. Интеллектуальные системы (ИС). Функции и структура современных ИС. Отличие знаний от данных.

2. Модели представления знаний: логическая; сетевая; фреймовая и продукционная модели представления знаний.

3. Достоинства и недостатки изученных моделей.

1. Интеллектуальные системы (ИС). Функции и структура современных ИС. Отличие знаний от данных

С развитием вычислительной техники стремление человека создать “искусственный разум” оформилось в отдельную область исследований, посвященную разработке интеллектуальных систем (ИС).

Под *интеллектуальными системами* понимается комплекс аппаратных и программных средств для решения интеллектуальных задач, т.е. таких задач, которые не решаются с помощью обычных алгоритмических подходов. Существуют и более структурированные определения ИС. Одно из них мы рассмотрим позже.

Характерным признаком интеллектуальных систем является наличие знаний, необходимых для решения задач конкретной предметной области. При этом возникает естественный вопрос, что такое знания и чем они отличаются от обычных данных, обрабатываемых ЭВМ.

Разработка интеллектуальных систем существенно отличается от обычного программирования. Если обычную программу можно представить как: ПРОГРАММА=АЛГОРИТМ+ДАННЫЕ, то для интеллектуальных систем характерна другая парадигма: ИС=ЗНАНИЯ+СТРАТЕГИЯ ОБРАБОТКИ ЗНАНИЙ.

Данные – это любые исходные, промежуточные или выходные сведения о задаче, решаемой в текущий момент времени.

Определение термина «знания» включает в себя большей частью философские элементы. Например, знание – это проверенный практикой результат познания действительности, верное ее отображение в сознании человека.

Знание есть результат, полученный познанием окружающего мира и его объектов. В простейших ситуациях знания рассматривают как констатацию фактов и их описание.

Исследователями в области ИИ даются более конкретные определения знаний.

Знания – это любая информация об общих свойствах и закономерностях проблемной области, выраженная в терминах некоторой модели представления знаний, которая хранится вне зависимости от того решает ли система какую-либо задачу в данный момент или нет.

Со знаниями тесно связано понятие процедуры получения решения задачи (стратегии обработки знаний). Эта процедура называется механизмом вывода (логическим выводом).

Знания — это формализованная информация, на которую ссылаются или которую используют в процессе логического вывода.

Под знанием будем понимать совокупность фактов и правил. Понятие правила, представляющего фрагмент знаний, имеет вид:

если <условие> то <действие>.

Это определение есть частный случай предыдущего определения.

Рассмотрим определение интеллектуальной системы с точки зрения функций, реализованных в ней. Это определение сформулировано профессором Д.А. Поспеловым:

«Система называется интеллектуальной, если в ней реализованы следующие основные функции:

– накапливать знания об окружающем систему мире, классифицировать и оценивать их с точки зрения прагматической полезности и непротиворечивости, инициировать процессы получения новых знаний, осуществлять соотнесение новых знаний с ранее хранимыми;

– пополнять поступившие знания с помощью логического вывода, отражающего закономерности в окружающем систему мире или в накопленных ею ранее знаниях, получать обобщенные знания на основе более частных знаний и логически планировать свою деятельность;

– общаться с человеком на языке, максимально приближенном к естественному

человеческому языку, и получать информацию от каналов, аналогичных тем, которые использует человек при восприятии окружающего мира, уметь формировать для себя или по просьбе человека (пользователя) объяснение собственной деятельности, оказывать пользователю помощь за счет тех знаний, которые хранятся в памяти, и тех логических средств рассуждений, которые присущи системе».

В зависимости от набора компонентов, реализующих рассмотренные функции, можно, выделить следующие *основные разновидности ИС:*

1) Интеллектуальные информационно-поисковые системы или системы взаимодействия с проблемно-ориентированными (фактографическими) базами данных на естественном языке. Данный класс систем хорошо изучен, в дальнейшем мы не будем к нему возвращаться.

2) Экспертные системы (ЭС) - бурно развивающийся класс ИС. Данные системы в первую очередь стали развиваться в математически слабо формализованных областях науки и техники, таких, как медицина, геология, биология и др. Для них характерна аккумуляция в системе знаний и правил рассуждений опытных специалистов в данной предметной области, а также наличие специальной системы объяснений.

3) Расчетно-логические системы, позволяющие решать управленческие и проектные задачи по их постановкам (описаниям) и исходным данным вне зависимости от сложности математических моделей этих задач. При этом конечному пользователю предоставляется возможность контролировать в режиме диалога все стадии вычислительного процесса.

В последнее время в специальный класс выделяются гибридные экспертные системы. Указанные системы должны вобрать в себя лучшие черты, как экспертных, так и расчетно-логических и информационно-поисковых систем. Разработки в области гибридных экспертных систем находятся на начальном этапе.

На сегодняшний день существует большое число систем искусственного интеллекта. Так или иначе, отдельные элементы систем искусственного элемента можно встретить на многих вычислительных машинах.

Структура интеллектуальных систем

В наиболее общем виде сущность интеллектуальной системы может быть проиллюстрирована следующей схемой, на которой отражена совокупность блоков (компонентов), выполняющих основные функции ИС.

Рис.1. Общая структура интеллектуальной системы

В *базе знаний* (БЗ) сосредоточены те знания, которыми располагает система, а также необходимые средства преобразования знаний. Из рисунка видно, что знания

различных типов хранятся в различных базах. Так, в *базе фактов* (данных) хранятся конкретные данные.

В *базе правил* - элементарные выражения, называемые в теории искусственного интеллекта (ИИ) продукциями.

База процедур содержит прикладные программы, с помощью которых выполняются все необходимые преобразования и вычисления над данными.

База закономерностей включает различные сведения, относящиеся к особенностям той среды, в которой действует система.

База метазнаний (база знаний о себе) содержит списки того, что хранится в данный момент в остальных базах: сведения о том, как внутри системы представляются единицы информации различного типа, как взаимодействуют различные компоненты системы, как было получено решение задачи, т. е. она хранит описание самой системы и способов ее функционирования.

В *базе целей* представлены целевые структуры, позволяющие организовать процессы движения от исходных фактов, правил, процедур к достижению той цели, которая поступила в систему от пользователя либо была сформулирована самой системой в процессе ее деятельности в проблемной среде. Такие целевые структуры в теории ИИ получили название *сценариев.*

Монитор БЗ осуществляет управление всеми базами, входящих в базу знаний, и организацию их взаимодействия. С его же помощью реализуются связи БЗ с внешней средой. Таким образом, БЗ осуществляет *первую функцию ИС.*

Часть системы, называемая *решателем,* обеспечивает выполнение *второй функции.* Он также состоит из ряда блоков, управляемых *монитором решателя.* Часть из них реализует логический вывод.

Блок *дедуктивного вывода* реализует в решателе дедуктивные рассуждения, с помощью которых на основе фактов из базы фактов и правил из базы правил выводятся новые факты. Этот же блок реализует поиск путей решения задачи по сценариям при заданной конечной цели (так называемые эвристические процедуры поиска решений задач).

Для реализации рассуждений недедуктивного характера (поиск по аналогии, по прецеденту и пр.) используются блоки *индуктивного* и *правдоподобного выводов.*

Блок *планирования* используется в задачах планирования решений (совместно с блоком дедуктивного вывода).

Назначение блока *функциональных преобразований* состоит в решении задач расчетно-логического и алгоритмического типа.

Функция общения реализуется как с помощью *естественно-языкового интерфейса,* так и с *помощью рецепторов* и *эффекторов,* которые осуществляют так называемое невербальное общение и используются в интеллектуальных роботах.

2. ***Модели представления знаний: логическая; сетевая; фреймовая и продукционная модели представления знаний***

Знания представляются в виде описаний объектов, отношений между ними и процедур их обработки.

Создание общей теории или способа представления знаний представляет собой стратегическую проблему и до ее полного решения достаточно далеко. Однако уже сегодня имеется ряд способов представления знаний, позволяющих создавать как интересные экспериментальные, так и эффективные коммерческие знание-ориентированные системы в различных прикладных областях, из которых особый интерес к себе вызывают экспертные системы (ЭС).

К настоящему времени известны, по меньшей мере, четыре формы представления знаний в экспертных системах:

- логические модели
- семантические сети
- фреймы

- системы продукций

Рассмотрим особенности каждой из них.

Логические модели

Логическая модель основана на системе исчисления предикатов первого порядка. Факты в базе знаний представляются с помощью утверждений, которые выражаются логическими предикатами.

Предикаты можно комбинировать с помощью логических связок, чтобы образовать логические выражения.

Суть логики предикатов сводится к тому, что из одних выражений можно формально получить новые выражения по определенным логическим правилам. К недостаткам логики предикатов 1-го порядка как метода представления знаний можно отнести следующее:

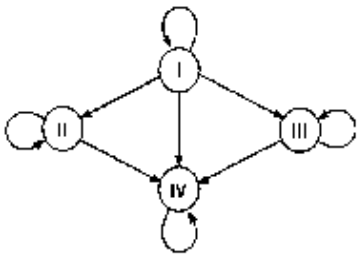
- монотонность логического вывода, т.е. невозможность пересмотра полученных промежуточных результатов (они считаются фактами, а не гипотезами);
- невозможность применения в качестве параметров предикатов других предикатов, т.е. невозможность формулирования знаний о знаниях;
- детерминированность логического вывода, т.е. отсутствие возможности оперирования с нечеткими знаниями.

Но с помощью логических моделей возможно моделирование рассуждений. В этом случае применяется так называемый дедуктивный вывод от общего к частному. С его помощью могут быть решены такие задачи, как поиск следствия и поиск доказательства.

Логическая модель применяется в основном в исследовательских системах, так как предъявляет очень высокие требования к качеству и полноте знаний предметной области.

Семантические сети определяют как граф общего вида, в котором можно выделить множество вершин и ребер. Каждая вершина графа представляет некоторое понятие или объект, а дуга - отношение между парой понятий (объектов). Метка и направление дуги конкретизируют семантику. Метки вершин семантической нагрузки не несут, а используются как справочная информация.

Объекты при таком подходе представляются именованными вершинами, а связи - направленными именованными дугами некоторой семантической сети. Тогда система знаний представляется некоторой семантической сетью (ориентированным графом), образованной именованными вершинами и дугами. В качестве вершин семантической сети выступают только те объекты проблемной области, которые необходимы для решения поставленных задач. В качестве таких объектов могут выступать: понятия, события, процессы и т.д.; свойства объектов также представляются вершинами сети и служат для описания классов объектов.



На этой схеме объект моделирования – различные виды групп крови человека обозначены кругами – это вершины графа. А стрелками показано, какую кровь можно переливать человеку с данной группой крови. Стрелки называются дугами графа. Дуга, исходящая от вершины и направленная к этой же вершине, называется петлей. Данный граф отражает такую жизненную ситуацию как переливание крови.

Объекты семантической модели разделяют на *обобщенные, конкретные* и

агрегатные. Обобщенный объект на самом деле представляет собой целый класс объектов (более низкого уровня) предметной области. Тогда как *конкретный объект* представляет собой некоторым образом выделенную *сущность* из класса. Под *агрегатным* понимается объект предметной области, составленный из других объектов. В качестве *агрегатного* может выступать как *обобщенный,* так и *конкретный* объект.

Между двумя объектами могут существовать различного типа отношения. В качестве наиболее распространенных (базовых) можно отметить следующие отношения между объектами:

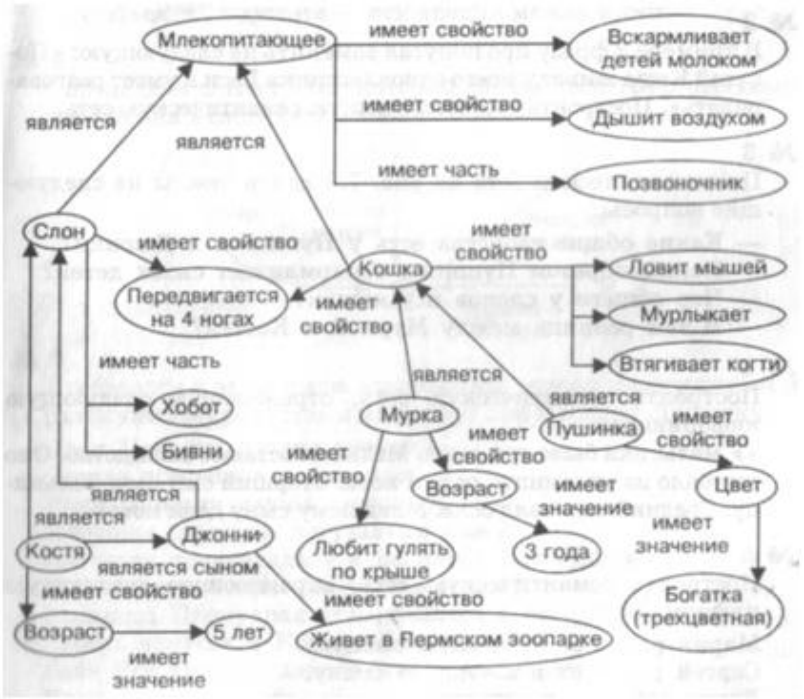
- *принадлежит* (объект *принадлежит* данному классу);
- *обладает* (объект *обладает* некоторым свойством);
- *значение* (определяет *значения* свойств объекта);
- *следствие* (отражает *причинно-следственные* связи: свойство является *следствием* некоторой причины).

Между *конкретным* и *обобщенным* объектами может существовать отношение *принадлежности,* когда конкретный объект принадлежит классу, отображаемому соответствующим *обобщенным* объектом. Например, *конкретный* объект ПК принадлежит *обобщенному* объекту ЭВМ. При этом *конкретный* объект может принадлежать нескольким *обобщенным* объектам одной и той же проблемной области. Между *агрегатным* объектом и другим может существовать отношение *вхождения,* смысл которого очевиден из его названия. Например, в состав одного данного *агрегатного* объекта может входить другой *агрегатный* объект. На основе *агрегатных* объектов и отношения *вхождения* можно представлять знания о сложных объектах проблемной области.

Классификация объектов и базовые отношения между ними не решают всех вопросов представления знаний, однако служат хорошей предпосылкой для создания конкретной БЗ.

Модификация БЗ, построенных на семантических сетях, сводится к процедурам удаления и добавления новых вершин и дуг. Основные операции поиска информации в сети обеспечивают: поиск нужной вершины или дуги по их именам (идентификаторам), переходы от вершины к вершине по соединяющим их дугам и переходы от дуге к дуге сети через смежные ее вершины. Основной целью такого поиска является получение знаний, заложенных в семантической сети БЗ и требуемых для решения поставленной задачи.

Рассмотрим пример семантической сети, представленной на рисунке.



Данный пример хорошо иллюстрирует отличие модели знаний от базы данных.

Семантическая сеть наглядно отражает взаимосвязь входящих в нее объектов. Например, если в базу данных о животных добавить новую запись «Ник – это слон», то мы будем знать про Ника один только этот факт и все. Но если добавить этот факт в данную семантическую сеть, то сразу же станет ясно, что Ник – это млекопитающее, его детей надо вскармливать молоком, что он дышит воздухом, передвигается на четырех ногах, имеет хобот, бивни и позвоночник, принадлежит к тому же классу, что Джонни, Костя и пр.

В виде семантической сети можно представить различные системы. Например,

1. Система «Школьный урок», состоящая из следующих элементов: ученик, учитель, учебник, тетрадь, классный журнал, классная доска, мел, парта, учительский стол, классная комната.
2. Круговорот воды в природе. Объекты: водоемы (моря, океаны, озера, пруды и пр.), реки, подземные воды, атмосфера, облака, почва, растения.
3. Система высших органов власти Российской Федерации.

В настоящее время аппарат семантических сетей применяется весьма широко для построения БЗ в различных интеллектуальных системах.

Наряду с определенными недостатками данный подход характеризуется целым рядом важных преимуществ: хорошей выразительностью, естественностью и наглядностью графически представленных систем знаний; близостью структуры сети, представляющей систему знаний, семантической структуре естественного языка и т.д.

Дальнейшее развитие подхода к представлению знаний на основе семантических сетей привело к появлению целого ряда их модификаций. Из наиболее известных модификаций можно отметить аппарат фреймов**.**

Фреймовые модели представления знаний

Теория представления знаний фреймами была разработана М.Минским в 70-е гг XX в. В ее основе лежит восприятие фактов посредством сопоставления полученной извне информации с конкретными элементами и значениями, а также с рамками, определенными для каждого объекта в памяти человека.

Под фреймом понимается абстрактный образ или ситуация. Например, слово "комната" вызывает у воспринимающего информацию человека образ: "помещение с четырьмя стенами, полом и потолком, площадью от 6 до 50 кв.м." Из этого образа ничего нельзя убрать (если убрать один из элементов, то представляемое помещение уже не будет комнатой), но при этом в данном образе можно заполнить значения

нескольких атрибутов (высота стен, тип покрытия пола, цвет потолка и т.д.) В теории фреймов такой образ называется фреймом.

Суть подхода на основе фреймов заключается в максимальной концентрации знаний об объекте предметной области — вся важная с точки зрения решаемой задачи информация об объекте помещается во фрейм. *Фреймом* называется структура для описания стереотипных объектов, событий и ситуаций, состоящая из их *характеристик* и *значений;* характеристики называются *слотами,* а значения — *заполнителями слотов.*

В общем виде фрейм можно представить следующим кортежем:

< ИФ, (ИС, ЗС),.., (ИС. ЗС)>,

где ИФ - имя фрейма;

ИС - имя слота;

ЗС - значение слота.

Слоты - это некоторые незаполненные подструктуры фрейма, в результате заполнения которых данный фрейм ставится в соответствие определенной ситуации, явлению или объекту.

Каждый фрейм, как структура хранит знания о предметной области (***фрейм–прототип***), а при заполнении слотов знаниями превращается в конкретный фрейм события или явления.

Фреймы можно разделить на две группы: фреймы-описания; ролевые фреймы.

Рассмотрим пример.

Фрейм описание:

<лекция, (предмет, физика), (лектор, Иванов И.И.), (аудитория,38), (студенты,15)>

В ролевом фрейме в качестве имен слотов выступают вопросительные слова, ответы на которые являются значениями слотов.

Ролевой фрейм:

<лекция, (по чему, предмет, физика), (кто, лектор, Иванов И.И.), (где, аудитория, 38), (для кого, студенты,15)>

Для данного примера представлены уже описания конкретных фреймов, которые могут называться либо фреймами – примерами, либо фреймами – экземплярами.

Если в приведенном примере убрать значения слотов, оставив только имена, то получим так называемый фрейм – прототип.

Фрейм – прототип

<лекция, (предмет),(лектор),(аудитория),(студенты)>

Кроме слотов и значений слотов, фрейм может содержать процедуры, которые будут выполняться при определенных условиях (при записи или удалении информации из слота, при обращении к слоту, в котором отсутствуют данные и т.д.) С каждым слотом может быть связано любое количество процедур.

Процедуры, связанные с определенным слотом фрейма, сильно зависят от конкретной прикладной системы, использующей фреймовые структуры для представления знаний. Если взять фрейм "ЛЕКЦИЯ" из предыдущего примера, и представить, что он используется в системе подготовки расписания занятий, то процедура, вызывающаяся при внесении значения в слот "ЛЕКТОР", могла бы уведомлять лектора об изменении расписания; процедуры, вызывающиеся при изменении значений слотов "АУДИТОРИЯ" и "СЛУШАТЕЛИ", могли бы проверять, вместит ли указанная аудитория всех слушателей и т.д.

Между различными объектами существуют некоторые аналогии, в результате чего и фреймы, представляющие такие образы, выстраиваются в иерархическую систему. При этом сложные объекты представляются комбинацией нескольких фреймов (вложенными фреймами).

Фреймовые модели отвечают требованиям структурированности и связанности. Это достигается за счет свойств наследования и вложенности, которыми обладают фреймы, т.е. в качестве слотов может выступать система имен слотов более низкого

уровня, а также слоты могут быть использованы как вызовы каких-либо процедур для выполнения.

Структура фреймов позволяет систематизировать большой объем информации, оставляя ее при этом максимально удобной для использования. Кроме того, система (сеть) фреймов способна отражать концептуальную основу организации памяти человека.

Для многих предметных областей фреймовые модели являются основным способом формализации знаний.

Обладая целым рядом достоинств, системы на основе фреймов имеют недостатки: они достаточно сложны, вывод утверждений в общем случае требует существенных временных затрат.

Продукционные модели

Системы продукций являются расширением логики предикатов. Главная особенность систем продукций заключается в возможности обновлении базы знаний и введении различных ограничений на процесс вывода.

Система продукции имеет три основные компоненты:

– базу данных (набор фактов)

– базу правил (набор правил)

– интерпретатор (для управления процессом вывода).

Интерпретатор системы продукций сравнивает условие (или ситуации) некоторого правила с фактами из базы знаний. Если имеется совпадение, то вызывается действие части правила; вследствие чего могут быть (исключены, модифицированы) факты в базе данных или в базе правил или могут быть выполнены другие произвольные действия. Таким образом, система, основанная на правилах, достаточно универсальна.

Для того чтобы сделать систему более эффективной, могут быть наложены различные подходящие ограничения. Например, правила могут быть линейно упорядочены, и последовательно сканироваться интерпретатором.

В таких моделях могут быть использованы как "вывод от цели к фактам" так и "вывод от фактов к цели".

Продукционные модели - это набор правил вида "условия - действие", где условиями являются утверждения о содержимом базы данных, а действия представляют собой процедуры, которые могут изменять это содержимое.

Практически продукции строятся по схеме "ЕСЛИ (причина), ТО (следствие)". Причину называют также посылкой, а следствие - целью правила. В хорошей системе насчитывается от 1000 до 3000 правил.

3. Достоинства и недостатки изученных моделей

Достоинства и недостатки изученных моделей представлены в таблице 1.

Таблица 1.Сравнительные характеристики моделей представления знаний.

Модель	Достоинства	Недостатки
Продукции	1. Модульность 2. Независимость 3. Простота модификации правил 4. Отделение предметных правил от управляющих	1. Отсутствие внутренней структуры 2. Зависимость шагов от стратегии вывода
Семантические сети	1. Возможность представления ограничений 2. Наличие внутренней структуры 3. Определение операций над объектами	1. Нет средств определения зависимости от времени 2. Произвольность структуры
Фреймы	Наличие внутренней структуры и связности	1. Жесткость структуры 2. Трудности в построении модулей и в модификации

Тема. Понятие об экспертной системе

План

1. Общая характеристика экспертных систем (ЭС). Типы задач, решаемые в ЭС. Структура и режимы использования ЭС. Классификация инструментальных средств в ЭС.

2. Организация знаний в экспертных системах на примере создания ЭС для аттестации знаний студентов по дисциплине «Основы искусственного интеллекта». Примеры современных ЭС, области их применения.

1. Общая характеристика экспертных систем (ЭС). Типы задач, решаемые в ЭС. Структура и режимы использования ЭС. Классификация инструментальных средств в ЭС

Экспертные системы как самостоятельное направление в искусственном интеллекте сформировалось в конце 1970-х гг. 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, как это делают специалисты (эксперты) при выработке умозаключений.

Экспертные системы – это прикладные системы ИИ, в которых база знаний представляет собой формализованные эмпирические знания высококвалифицированных специалистов (экспертов) в какой-либо узкой предметной области. ЭС предназначены для замены при решении задач экспертов в силу их недостаточного количества, недостаточной оперативности в решении задачи или в опасных (вредных) для них условиях.

Можно выделить следующие *основные типы задач*, решаемых экспертными системами:

- диагностика;
- прогнозирование;
- идентификация;
- управление;
- проектирование;
- мониторинг.

Любая ЭС должна содержать как минимум два основных элемента: базу знаний и механизм ввода-вывода.

Большинство современных развитых **ЭС** включает следующие пять базовых компонент: *базу знаний, систему логического вывода, специальные подсистемы приобретения знаний и пояснений, а также пользовательский интерфейс*.

Рассмотрим несколько детальнее каждую из указанных компонент и ее место в общей архитектуре ЭС.

Центральное место ЭС составляет ее **БЗ**, создаваемая и поддерживаемая инженером базы знаний. Процесс построения БЗ достаточно сложен, как правило, плохо структурирован. На основе результатов тестирования БЗ периодически меняется.

Основу БЗ составляют формально представленные в ней факты и правила модели предметной области. В *системе логического вывода* реализуется некая стратегия выбора соответствующего правила из БЗ, существенно зависящая от метода представления знаний в системе и характера решаемых ею задач.

Полученная после формализации БЗ может быть уже конкретно реализована программными средствами.

Процесс приобретения знаний состоит в передаче знаний и опыта от источника (эксперт, специальные публикации, опытные факты и т.д.) системе. Как правило, в качестве источника выступают эксперты не в целом по предметной области, на которую ориентирована ЭС, а специалисты по ее узким направлениям. Полученные от

эксперта знания представляют собой набор фактов, правил, процедур и оценок по его узкой профессиональной области.

Подсистема пояснений предназначена для отображения в удобном для пользователя виде промежуточных и окончательных выводов и объяснения действий, производимых системой. Как правило, для этих целей используется диалоговый режим и графический интерфейс, которые отображают ход принятия решения системой, что позволяет поэтапно отслеживать процесс принятия решений.

С учетом сказанного общая схема взаимодействия пользователя с ЭС сводится к следующему. После создания конкретной ЭС взаимодействие пользователя с системой производится через интерфейс на некотором непроцедурном языке, близком к естественному или профессиональному языку предметной области, на которую ориентирована данная ЭС. В интерфейсной компоненте ЭС производится трансляция предложений этого языка на внутренний язык ПЗ системы. Описанные запросы на внутреннем языке ПЗ поступают в систему логического вывода, которая на основе информации из БЗ генерирует рекомендации по решению поставленного вопроса.

Для реализации таких систем существует широкий круг языков программирования. Как правило, эти языки варьируются в зависимости от области применения систем.

2. Организация знаний в экспертных системах на примере создания ЭС для аттестации знаний студентов по дисциплине «Основы искусственного интеллекта». Примеры современных ЭС, области их применения

Для реализации ЭС необходимо выполнить следующие действия:

1. Создать базу знаний.
2. Создать базу данных.
3. Обработать ответы.
4. Вычислить общие весовые факторы.
5. Вычислить максимальный общий весовой фактор.
6. Принять решение.

Простейшая среда для реализации данной экспертной системы – это Microsoft Excel.

Рассмотрим пример создания ЭС для аттестации знаний студентов по дисциплине «Основы искусственного интеллекта».

Разработана исходная база знаний, которая определяет весовые коэффициенты той или иной темы раздела (таблица 1).

Таблица 1. Исходная база знаний для определения весовых коэффициентов темы.

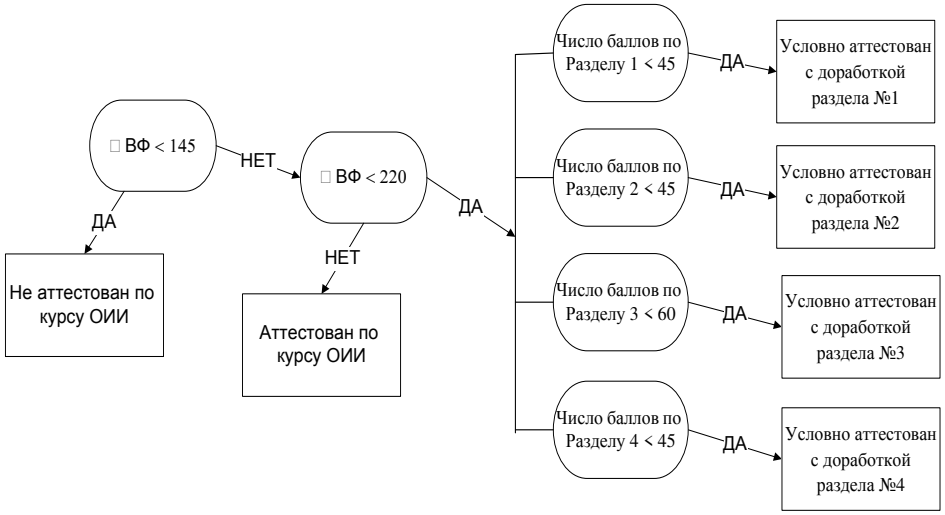
№	Раздел	Атрибут темы	Весовой фактор
1	**"Определение искусственного интеллекта"**	*Знает*: Определение искусственного интеллекта	5
		Отличие знаний от данных	10
		Классификацию моделей представления знаний (МПЗ)	5
		Определение основных видов МПЗ	15
		Структуру основных видов МПЗ	20
2	**"Экспертные системы (ЭС)"**	*Знает*: Понятие ЭС	5
		Архитектуру ЭС	20
		Процесс функционирования ЭС	25
		Виды ЭС	5
3	**"Язык программирования Пролог"**	*Знает*: Отличие процедурных языков программирования от декларативных	10
		Основные секции программы на Прологе и их применение	15
		Умеет: Организовать поиск с возвратом	30

		Организовать рекуррентный вызов	30
4	**"Нейронные сети (НС)"**	*Знает*: Понятие образа	5
		Понятие о нейронной сети	15
		Обоснование проблемы распознавания образа	5
		Виды НС	15
		Области применения НС	5
		Структуру НС	10
Максимально возможная сумма баллов			250

Весь материал дисциплины «ОИИ» разбит на 4 раздела, каждый из которых содержит несколько тем, оцениваемых заданным весовым фактором (Таблица 1). Величина весового фактора определяется в зависимости от сложности темы и объема изучаемого материала.

Базу данных (таблица 2) строится согласно базе знаний. Для этого формулируются тестовые вопросы по фактам, приведенным в задании. Например, для факта "Знает определение ИИ" сформируем вопрос " Знает определение ИИ?" и т.д. В базе данных предусматривается поле для ввода ответов. Если ответ на вопрос положительный (да), то весовой фактор соответствующего атрибута сохраняется. Если ответ отрицательный (нет), весовой фактор берется равным нулю. База данных в рассматриваемом примере заполняется преподавателем, принимающим зачет у студента. Есть и другой вариант: база данных может заполнятся автоматически после выполнения студентом теста, соответствующего ей.

Дерево решений для рассматриваемой экспертной системы представлено на рисунке:



Максимально возможная сумма весовых факторов, набранных студентом в процессе аттестации, составляет 250 баллов. Если студент набирает менее 145 (58 %) баллов, то он не будет аттестован по данной дисциплине, более 220 (88 %) – аттестован по данной дисциплине. Если сумма набранных баллов более 145, но менее 220, экспертная система должна произвести проверку числа полученных баллов по каждому из четырех разделов. Таким образом, определяется, какой из разделов не усвоен студентом в необходимом объеме и требует доработки.

Разрабатываемая ЭС должна использоваться многократно для анализа различных вариантов и предусматривать возможность многократного обновления БД.

Наиболее широко встречающиеся области деятельности, где используются экспертные системы:

- медицина;
- вычислительная техника;
- военное дело;
- микроэлектроника;
- радиоэлектроника;

• юриспруденция;
• экономика;
• экология;
• геология (поиск полезных ископаемых);
• математика.

Тема. Процедурные и декларативные языки программирования. Язык программирования Пролог как декларативный язык

План

1. Отличие процедурных и декларативных языков программирования. Представление знаний о предметной области в виде фактов и правил базы знаний Пролога.

2. Дескриптивный, процедурный и машинный смысл программы на Прологе.

3. Представление знаний о предметной области в виде фактов и правил базы знаний Пролога.

1. Отличие процедурных и декларативных языков программирования. Представление знаний о предметной области в виде фактов и правил базы знаний Пролога

На протяжении многих тысячелетий человечество занимается накоплением, обработкой и передачей знаний. Для этих целей непрерывно изобретаются новые средства и совершенствуются старые: речь, письменность, почта, телеграф, телефон и т. д. Большую роль в технологии обработки знаний сыграло появление компьютеров.

В октябре 1981 года Японское министерство международной торговли и промышленности объявило о создании исследовательской организации — Института по разработке методов создания компьютеров нового поколения. Целью данного проекта было создание систем обработки информации, базирующихся на знаниях. Предполагалось, что эти системы будут обеспечивать простоту управления за счет возможности общения с пользователями при помощи естественного языка. Эти системы должны были самообучаться, использовать накапливаемые в памяти знания для решения различного рода задач, предоставлять пользователям экспертные консультации, причем от пользователя не требовалось быть специалистом в информатике. Предполагалось, что человек сможет использовать ЭВМ пятого поколения так же легко, как любые бытовые электроприборы типа телевизора, магнитофона и пылесоса. Вскоре вслед за японским стартовали американский и европейский проекты.

Появление таких систем могло бы изменить технологии за счет использования баз знаний и экспертных систем. Основная суть качественного перехода к пятому поколению ЭВМ заключалась в переходе от обработки данных к обработке знаний. Японцы надеялись, что им удастся не подстраивать мышление человека под принципы функционирования компьютеров, а приблизить работу компьютера к тому, как мыслит человек, отойдя при этом от фон неймановской архитектуры компьютеров. В 1991 году предполагалось создать первый прототип компьютеров пятого поколения.

Теперь уже понятно, что поставленные цели в полной мере так и не были достигнуты, однако этот проект послужил импульсом к развитию нового витка исследований в области искусственного интеллекта и вызвал взрыв интереса к логическому программированию. Так как для эффективной реализации традиционная фон неймановская архитектура не подходила, были созданы специализированные компьютеры логического программирования (PSI и PIM).

В качестве основной методологии разработки программных средств для проекта ЭВМ пятого поколения было избрано логическое программирование, ярким представителем которого является язык Пролог. В настоящее время Пролог остается наиболее популярным языком искусственного интеллекта в Японии и Европе (в США,

традиционно, более распространен другой язык искусственного интеллекта —язык функционального программирования Лисп).

Название языка "Пролог" происходит от слов ПРОграммирование в ЛОГике (PROgramming in LOGic — в английском варианте).

Пролог основывается на таком разделе математической логики, как исчисление предикатов. Сегодня существует довольно много реализаций Пролога. Наиболее известные из них следующие: BinProlog, AMZI-Prolog, Arity Prolog, CProlog, Micro Prolog, МПролог, Prolog-2, Quintus Prolog, SICTUS Prolog, Silogic Knowledge Workbench, Strawberry Prolog, SWI Prolog, UNSW Prolog и т. д.

В нашей стране были разработаны такие версии Пролога как Пролог-Д (Сергей Григорьев), Акторный Пролог (Алексей Морозов), а также Флэнг (А Манцивода, Вячеслав Петухин).

Традиционно под программой понимают последовательность операторов (команд, выполняемых компьютером). Этот стиль программирования принято называть **императивным (или процедурным)**. Программируя в императивном стиле, программист должен объяснить компьютеру, **как** нужно решать задачу.

Противоположный ему стиль программирования — так называемый **декларативный стиль**, в котором программа представляет собой совокупность утверждений, описывающих фрагмент предметной области или сложившуюся ситуацию. Программируя в декларативном стиле, программист должен описать, **что** нужно решать.

Соответственно и языки программирования делят на императивные и декларативные.

Императивные языки основаны на фон неймановской модели вычислений компьютера. Решая задачу, императивный программист вначале создает модель в некоторой формальной системе, а затем переписывает решение на императивный язык программирования в терминах компьютера. Но, во-первых, для человека рассуждать в терминах компьютера довольно неестественно. Во-вторых, последний этап этой деятельности (переписывание решения на язык программирования) по сути дела не имеет отношения к решению исходной задачи. Очень часто императивные программисты даже разделяют работу в соответствии с двумя описанными выше этапами. Одни люди, постановщики задач, придумывают решение задачи, а другие, кодировщики, переводят это решение на язык программирования.

В основе декларативных языков лежит формализованная человеческая логика. Человек лишь описывает решаемую задачу, а поиском решения занимается императивная система программирования. В итоге получаем значительно большую скорость разработки приложений, значительно меньший размер исходного кода, легкость записи знаний на декларативных языках, более понятные, по сравнению с императивными языками, программы.

Известна классификация языков программирования по их близости либо к машинному языку, либо к естественному человеческому языку. Те, что ближе к компьютеру, относят к языкам **низкого уровня**, а те, что ближе к человеку, называют языками **высокого уровня**. В этом смысле декларативные языки можно назвать языками **сверхвысокого** или **наивысшего** уровня, поскольку они очень близки к человеческому языку и человеческому мышлению.

К императивным языкам относятся такие языки программирования, как Паскаль, Бейсик, Си и т. д. В отличие от них, Пролог является декларативным языком.

Логическое программирование - это один из подходов к информатике, при котором в качестве языка высокого уровня используется логика предикатов первого порядка в форме фраз Хорна. Логика предикатов первого порядка - это универсальный абстрактный язык предназначенный для представления знаний и для решения задач. Его можно рассматривать как общую теорию отношений. Логическое программирование базируется на подмножестве логики предикатов первого порядка,

при этом оно одинаково широко с ней по сфере охвата. Логическое программирование дает возможность программисту описывать ситуацию при помощи формул логики предикатов, а затем, для выполнения выводов из этих формул, применить автоматический решатель задач (т. е. некоторую процедуру).

Фактически Пролог представляет собой не столько язык для программирования, сколько язык для описания данных и логики их обработки. Программа на Прологе не является таковой в классическом понимании, поскольку не содержит явных управляющих конструкций типа условных операторов, операторов цикла и т. д. Она представляет собой модель фрагмента предметной области, о котором идет речь в задаче. И решение задачи записывается не в терминах компьютера, а в терминах предметной области решаемой задачи.

Пролог очень хорошо подходит для описания взаимоотношений между объектами. Поэтому Пролог называют реляционным языком. Причем "реляционность" Пролога значительно более мощная и развитая, чем "реляционность" языков, используемых для обработки баз данных. Часто Пролог используется для создания систем управления базами данных, где применяются очень сложные запросы, которые довольно легко записать на Прологе.

В Прологе очень компактно, по сравнению с императивными языками, описываются многие алгоритмы. По статистике, строка исходного текста программы на языке Пролог соответствует четырнадцати строкам исходного текста программы на императивном языке, решающем ту же задачу. Пролог-программу, как правило, очень легко писать, понимать и отлаживать. Это приводит к тому, что время разработки приложения на языке Пролог во многих случаях на порядок быстрее, чем на императивных языках. В Прологе легко описывать и обрабатывать сложные структуры данных.

Прологу присущ ряд механизмов, которыми не обладают традиционные языки программирования: сопоставление с образцом, вывод с поиском и возвратом. Еще одно существенное отличие заключается в том, что для хранения данных в Прологе используются списки, а не массивы. В языке отсутствуют операторы присваивания и безусловного перехода, указатели. Естественным и зачастую единственным методом программирования является рекурсия. Поэтому часто оказывается, что люди, имеющие опыт работы на процедурных языках, медленней осваивают декларативные языки, чем те, кто никогда ранее программированием не занимался, так как Пролог требует иного стиля мышления, отказа от стереотипов императивного программирования.

Но Пролог — это не универсальный "решатель" любой задачи. Для каждого языка существует свой класс задач, для решения которых он подходит лучше других языков программирования. Соответственно, для решения любой задачи есть оптимальный язык (языки) программирования. Многие задачи, хорошо решаемые императивными языками типа Паскаля и Си, плохо решаются на Прологе, и наоборот. Давайте посмотрим, в каких областях наилучшим образом себя показал Пролог.

Основные области применения Пролога:

- быстрая разработка прототипов прикладных программ;
- автоматический перевод с одного языка на другой;
- создание естественно-языковых интерфейсов для существующих систем;
- символьные вычисления для решения уравнений, дифференцирования и интегрирования;
- проектирование динамических реляционных баз данных;
- экспертные системы и оболочки экспертных систем;
- автоматизированное управление производственными процессами;
- автоматическое доказательство теорем;
- полуавтоматическое составление расписаний;
- системы автоматизированного проектирования;

- базирующееся на знаниях программное обеспечение;
- организация сервера данных или, точнее, сервера знаний, к которому может обращаться клиентское приложение, написанное на каком-либо языке программирования.

Области, для которых Пролог не предназначен: большой объем арифметических вычислений (обработка аудио, видео и т.д.); написание драйверов.

Как уже было сказано выше, версий Пролога очень много, и нужно выбрать какую-нибудь одну из них, чтобы привязать к этой версии разбираемые примеры. Наиболее известной у нас в стране и довольно эффективной версией Пролога является Турбо Пролог. Его начинала разрабатывать фирма Borland International в содружестве с датской компанией Prolog Development Center (PDC). Первая версия вышла в 1986 году. Последняя совместная версия имела номер 2.0 и была выпущена в 1988 году.

В 1990 году PDC получила монопольное право на Турбо Пролог и дальше продвигала его под названием PDC Prolog.

В 1992 году вышла версия PDC Prolog 3.31.

В 1996 году, при участии группы питерских программистов, Prolog Development Center выпустила систему Visual Prolog 4.0. В состав среды Visual Prolog были включены инструментальные средства генерации кода, конструирующие управляющую логику, интерфейс визуального программирования и многие другие средства, позволяющие ускорить разработку приложений. Помимо прочих достоинств среды Visual Prolog стоит обратить внимание на возможность использования в идентификаторах символов национального алфавита, в частности, можно употреблять в программах русские имена доменов, предикатов и переменных, что делает программу более понятной и самодокументированной.

2. Дескриптивный, процедурный и машинный смысл программы на Прологе

Смысл программы языка Пролог может быть понят либо с позиций декларативного подхода, либо с позиций процедурного подхода.

Декларативный смысл программы определяет, является ли данная цель истинной (достижимой) и, если да, при каких значениях переменных она достигается. Он подчеркивает статическое существование отношений. Порядок следования подцелей в правиле не влияет на декларативный смысл этого правила. Декларативная модель более близка к семантике логики предикатов, что делает Пролог эффективным языком для представления знаний. Однако в декларативной модели нельзя адекватно представить те фразы, в которых важен порядок следования подцелей. Для пояснения смысла фраз такого рода необходимо воспользоваться процедурной моделью.

При процедурной трактовке Пролог-программы определяются не только логические связи между головой предложения и целями в его теле, но еще и порядок, в котором эти цели обрабатываются. Но процедурная модель не годится для разъяснения смысла фраз, вызывающих побочные эффекты управления, такие как остановка выполнения запроса или удаление фразы из программы.

3. Представление знаний о предметной области в виде фактов и правил базы знаний Пролога

Рассмотрим пример программы на Prolog’e:

Например, необходимо выявить все объекты, имеющие свойство быть птицей.

Предположим, имеются следующие исходные данные:

- журавль – это птица;
- у синицы есть крылья;
- синица умеет летать;
- у пингвина есть крылья;
- пингвин умеет плавать;
- некто является птицей при условии, что у него есть крылья и он умеет летать.

Первые пять предложений будем считать не подлежащими сомнению и назовем фактами, шестое предложение – правилом вывода, так как это предложение формулирует правило, по которому можно сделать вывод о том, является ли некто птицей или нет. Некто будет птицей при условии, что он умеет летать и у него есть крылья. Условное обозначение :- читается: «при условии» или «если». Так как условия «умеет летать» и «есть крылья» перечислены через запятую, они должны быть выполнены оба. Запятая означает операцию «логическое И». Все предложения должны заканчиваться точкой.

Программа на языке Prolog будет выглядеть следующим образом:

```
птица (журавль).
есть_крылья (синица).
умеет_летать (синица).
есть_крылья (пингвин).
умеет_плавать (пингвин).
птица (Объект):- есть_крылья (Объект), умеет_летать (Объект).
```

Следует оговорить, что программа записана не по всем правилам, но для первого знакомства с языком Prolog вполне можно допустить несоблюдение некоторых правил. В разделе «Основные стандартные домены» этот пример будет приведен с соблюдением всех правил синтаксиса Prolog'а.

Теперь к программе можно обращаться с вопросами (запросами):

1. Кто является птицей? (Ответы: журавль, синица)
2. Кто умеет летать? (Ответ: синица)
3. У кого есть крылья? (Ответы: синица, пингвин)
4. и т.д.

Рассмотрим, каким образом работает программа. Предположим, нужно найти ответ на первый вопрос – кто является птицей? Вопрос будет иметь такой вид:

```
Goal: птица (Кто).
```

Кто – это имя переменной, которая в начале работы программы не имеет никакого значения.

Каким образом будет находиться первое решение? Будут последовательно просматриваться все строки программы и первая же строка (первый факт) дает первое решение – журавль.

Но язык Prolog своеобразен и существенно отличается от других языков программирования. В частности, если у задачи есть несколько решений, они все будут найдены. В рассматриваемом примере есть еще возможные решения, поэтому выполнение программы не прекращается после нахождения первого решения.

Для нахождения следующих возможных решений продолжается просмотр строк программы и обнаруживается правило вывода с подходящей для нахождения следующего решения левой частью. Переменные Кто и Объект начинают обозначать один и тот же объект, то есть, как только переменная Объект получит какое-нибудь значение, то же самое значение сразу же получит переменная Кто.

Теперь, для того, чтобы правило вывода дало второе решение, необходимо, чтобы были выполнены все условия, записанные в его правой части. Первое условие выполнится, если будет найден объект, у которого есть крылья. Другими словами, в тексте программы нужно найти факт, говорящий об этом. При просмотре программы с самого начала такой факт обнаруживается – у синицы есть крылья. Переменная Объект немедленно принимает значение «синица». Проверка выполнения второго условия начинается с учетом того, что переменная Объект уже имеет значение «синица», то есть второе условие в правиле вывода имеет вид – умеет_летать (синица). Но второе условие еще нельзя считать выполненным. Недостаточно того, что переменная Объект приняла значение «синица», нужно найти факт, подтверждающий, что синица действительно умеет летать. А для этого вновь просмотреть предложения программы с самого начала. Факт находится. Найдено второе решение «синица».

Больше решений обнаружить не удастся, так как больше нет фактов, описывающий птиц, у которых есть крылья и которые умеют летать.

Итак, полученные решения:

1. журавль (в соответствии с фактом)
2. синица (по правилу вывода)

Как видно в рассмотренном примере, основой для находимых решений являются факты, записанные в тексте программы. Если дописать в программу еще факты, например:

есть_крылья (самолет).

умеет_летать (самолет).

то будет найдено третье решение – «самолет».

Тема. Общий алгоритм выполнения и основные секции программ на Прологе. Рекурсия

План

1. Алгоритм выполнения программ на Прологе.

2. Основные секции программ на Прологе

3. Предикат отсечения и управление поиском с возвратом в программах на Прологе: предикаты «!» и «fail».

4. Организация циклов: механизм отката и рекурсия.

1. Алгоритм выполнения программ на Прологе

Предложения: факты и правила

Программа на языке Prolog состоит из предложений, которые можно разделить на две группы: *факты и правила вывода*.

В виде фактов в программе записываются данные, которые принимаются за истину и не требуют доказательства. Данные в фактах могут быть использованы для логического вывода. Факт может описывать некоторые свойства объекта или отношения между объектами. Можно дать следующее определение для факта: факт – это свойство объекта или отношение между объектами, для которого известно, что они истинны.

Примеры фактов:

```
%объект кот имеет свойство – черный цвет или, проще, кот – черный
black (cat).
%города Новосибирск и Омск связаны железной дорогой
railway (novosibirsk, omsk).
```

Второй тип предложений – *правила вывода*. Правило вывода состоит из двух частей, разделенных условным обозначением :- , которое читается как «если», или «при условии, что». Левая часть правила вывода называется заголовком или головной целью. Правая часть правила вывода называется хвостом или хвостовой частью. Хвостовая часть может состоять из нескольких условий (хвостовых целей), перечисленных через запятую или точку с запятой. Запятая означает операцию «логическое И», точка с запятой – операцию «логическое ИЛИ». Использовать скобки в хвостовой части правила вывода нельзя.

Головная цель правила вывода считается доказанной, если доказаны все хвостовые цели в правой части правила вывода.

Пример правил вывода:

```
%Некоторый объект, обозначенный переменной Bird, является пингвином,
%если объект умеет плавать и у него есть крылья
penguin (Bird):- swim (Bird), wings(Bird).
%Некоторый объект, обозначенный переменной Bird, является страусом,
%если объект не умеет летать, у него длинные ноги и есть крылья
ostrich (Bird):- not_fly (Bird), long_legs (Bird), wings(Bird).
```

В предложениях используются переменные для обобщенной формулировки правил вывода. Все предложения обязательно заканчиваются точкой.

Запросы

Для того чтобы программа, написанная на языке Prolog, начала работу, к ней нужно обратиться с запросом. Запросы могут быть двух видов: внутренние и внешние. Внутренние запросы записываются непосредственно в тексте программы, для использования внешних запросов нужна программная оболочка.

Вне зависимости от того, являются запросы внутренними или внешними, выглядят они одинаково. Запрос может быть простым (состоящим из одной цели) или составным (состоящим из нескольких целей). Выполнение программы заключается в доказательстве целей, входящих в запрос. Программа считается успешно выполненной (завершенной), если доказаны цели, из которых состоит запрос.

Предикаты

Предикат – это имя свойства или отношения между объектами с последовательностью аргументов.

При записи факта или цели с использованием некоторого предиката сначала записывается имя предиката, а затем в скобках, через запятую, его аргументы.

Например, факт black (cat). записан с использованием предиката black, имеющего один аргумент. Факт railway (novosibirsk, omsk). записан с использованием предиката railway, имеющего два аргумента.

Число аргументов предиката называется арностью предиката и обозначается black/1 (предикат black имеет один аргумент, его арность равна единице). Предикаты могут не иметь аргументов, арность таких предикатов равна 0.

Переменные

Работа с переменными в Prolog’е достаточна своеобразна. Если в других, алгоритмических, языках программирования значение переменной, которое было ей присвоено, не изменяется до тех пор, пока не будет выполнено переприсваивание значения, то в Prolog’е переменная может получить некоторое значение в процессе поиска решения и потерять его, только когда начнется поиск нового решения.

Принудительно изменить значение переменной, уже получившей значение, с помощью оператора присваивания нельзя, да и в Prolog’е отсутствует оператор присваивания в чистом виде, его функции время от времени берет на себя операция равенства (при соблюдении некоторых условий).

Имя переменной дается по следующим правилам: имя должно начинаться с заглавной латинской буквы или символа подчеркивания, после которых могут следовать латинские буквы, цифры или символы подчеркивания.

Пример:

First_list
X
Person
City

Переменная, не имеющая значения, называется свободной, переменная, имеющая значение – конкретизированной. Как было сказано выше, в Prolog’е нет оператора присваивания, его роль, в некоторых случаях, выполняет оператор равенства =.

Если записать следующую цель: …, X=5, …, то как эта цель будет рассматриваться, как сравнение или как присваивание, все зависит от того, получила ли какое-либо значение переменная X к моменту доказательства этой цели. Если переменная X имеет значение (например, равное 6), то оператор равенства = работает как оператор сравнения. Если же переменная X свободна (не имеет никакого значения), то оператор равенства = работает как оператор присваивания. При этом совершенно неважно, слева или справа от знака равенства находится имя переменной. Главное, чтобы она была свободной. С точки зрения программы на Prolog’е следующие две цели совершенно одинаковы:

…, X=5, …
…, 5=X, …

Самое важное, чтобы переменная X не имела значения. Из вышесказанного вытекает следующая особенность использования переменных в Prolog'е, нельзя записывать вот так:

…, X=X+5, …

В любом случае такая цель будет ошибочной. Действительно, если переменная X имеет, например, значение равное 10, то предыдущая цель сводится к доказательству цели:

…, 10=10+5, …

которая, естественно, не доказывается.

Если же переменная X свободна, то нельзя к переменной, не имеющей никакого значения, прибавить 5, и присвоить эту неопределенность той же самой переменной. Как же тогда быть, если нужно изменить значение переменной? Ответ один – использовать новое имя, поскольку переменная, которая появляется в тексте программы впервые, считается свободной, и может быть конкретизирована некоторым значением:

…, Y=X+5, …

При этом опять же не важен порядок записи. Такая цель также будет правильной, и будет выполнять присваивание (конечно, если переменная X конкретизирована некоторым числом):

…, X+5=Y, …

Еще существуют специальные переменные, называемые анонимными. Их имя состоит только из знака подчеркивания. Анонимные переменные используются в случаях, когда значение переменной несущественно, но переменная должна быть использована.

Пример:

parent ("Владимир", "Михаил").
parent ("Владимир", "Светлана").
parent ("Анна", "Михаил").
parent ("Анна ", " Светлана").

Факты описывают родителей и их детей (первый аргумент – имя родителя, второй – имя ребенка). Теперь, если нужно узнать только имена родителей, но не нужны имена детей, к программе можно обратиться с внешним запросом, использовав анонимную переменную:

Goal: parent (Person, _).

Решение в данном случае будет избыточным, поскольку есть четыре факта.

Person=Владимир
Person=Владимир
Person=Анна
Person=Анна
4 Solutions

Сравните, если использовать запрос:

Goal: parent (Person, Child).

какими будут результаты:

Person=Владимир, Child=Михаил
Person=Владимир, Child=Светлана
Person=Анна, Child=Михаил
Person=Анна, Child=Светлана
4 Solutions

2. Основные секции программ на Прологе

Как правило, программа на Prolog'е состоит из нескольких секций, ни одна из которых не является обязательной. Вот основные секции:

1. DOMAINS – секция описания доменов (типов). Секция применяется только, если в программе используются нестандартные домены.

2. PREDICATES – секция описания предикатов. Секция применяется, если в программе используются нестандартные предикаты.

3. CLAUSES – секция предложений. Именно в этой секции записываются предложения: факты и правила вывода.

4. GOAL – секция цели. В этой секции записывается внутренний запрос.

На первый взгляд, без секций DOMAINS, PREDICATES и GOAL действительно можно обойтись, но как написать программу без секции CLAUSES? Конечно, такая программа не обладает большим количеством возможностей, но принципиально такую программу написать можно. Например:

GOAL

write (“Введите Ваше имя: ”), readln (Name), write (“Здравствуйте, ”, Name, “!”).

Вот пример программы, состоящей только из секции GOAL. Используются только стандартные домены, следовательно, отпадает необходимость использовать секцию DOMAINS, нет нестандартных предикатов, следовательно, отпадает необходимость использовать секцию PREDICATES, и, наконец, все цели записаны непосредственно в секции GOAL, следовательно, нет необходимости использовать секцию CLAUSES.

Основные стандартные домены

Доменом в Prolog’е называют тип данных. В Prolog’е, как и других языках программирования, существует несколько стандартных доменов, перечислим их:

1. integer – целые числа.
2. real – вещественные числа.
3. string – строки (любая последовательность символов, заключенная в кавычки).
4. char – одиночный символ, заключенный в апострофы.
5. symbol – последовательность латинских букв, цифр и символов подчеркивания, начинающаяся с маленькой буквы или любая последовательность символов, заключенная в кавычки.

Пример (программа из введения, оформленная по всем правилам):

```
PREDICATES
bird (string)
has_wings (string)
can_fly (string)
can_swim (string)
CLAUSES
bird (“журавль”).
has_wings (“синица”).
has_wings (“пингвин”).
can_fly (“синица”).
can_swim (“пингвин”).
bird (Object):- has_wings (Object), can_fly (Object).
GOAL
bird (Who).
```

Несколько замечаний. Поскольку в программе не использовались нестандартные домены, не было необходимости использовать секцию описания доменов DOMAINS. В отличие от примера из введения, где был использован внешний запрос, в данной программе запрос записан в секции GOAL, то есть является внутренним. В таком случае находится только первое решение. Как находить все существующие решения, если используется внутренний запрос, более подробно рассмотрено в вопросе «Управление поиском с возвратом: предикаты ! и fail».

3. Предикат отсечения и управление поиском с возвратом в программах на Прологе: предикаты «!» и «fail»

Поиск с возвратом (backtracking) – это один из основных приемов поиска решений поставленной задачи в Prolog’е.

Предположим, для достижения некоторой цели человеку необходимо последовательно принять несколько решений и выполнить некоторые действия в соответствии с принятыми решениями. Первоначально человек без колебаний и раздумий принимает несколько решений, но при решении очередной проблемы у него возникают сомнения, поскольку возможных решений, предположим, имеется два, и человеку они кажутся одинаково правильными. Какое-либо из двух решений человек все равно принимает (но запоминает, в какой момент он сомневался, и какое из двух решений все же выбрал) и продолжает свое движение к поставленной цели.

Но, в какой-то момент оказывается, что решение, выбранное из двух, все же оказалось неправильным. Тогда человек вернется в точку принятия неверного решения, и пойдет по альтернативному пути. Не факт, что вновь выбранный путь окажется правильным, но человек попробует все возможные варианты нахождения решения.

Еще одна аналогия. Поиск с возвратом можно сравнить с поиском выхода из лабиринта. Нужно войти в лабиринт и на каждой развилке сворачивать влево, до тех пор, пока не найдется выход или тупик. Если впереди оказался тупик, нужно вернуться к последней развилке и свернуть направо, затем снова проверять все левые пути.

В конце концов, выход (если он есть) будет найден. Подобным образом работаем и механизм поиска с возвратом в языке Prolog.

Благодаря механизму поиска с возвратом Prolog в состоянии находить все возможные решения, имеющиеся для данной задачи.

Рассмотрим на примере, каким образом выполняется поиск всех возможных решений с применением поиска с возвратом.

```
PREDICATES
little (symbol)
middle (symbol)
big (symbol)
strong (symbol)
powerful (symbol)
CLAUSES
little (cat).
little (wolf).
middle (tiger).
middle (bear).
big (elephant).
big (hippopotamus).
strong (tiger).
powerful (Animal):- middle (Animal), strong (Animal).
powerful (Animal):- big (Animal).
```

Обратимся к программе с запросом – какое животное можно назвать мощным?

Запрос будет выглядеть следующим образом:

```
Goal: powerful (Animal).
```

Проследим по шагам, каким образом будут находиться все возможные решения.

Доказательство цели, сформулированной в запросе, начинается с последовательного просмотра всех предложений, имеющихся в тексте программы. В данном примере цель powerful (Animal) может быть сопоставлена с заголовком первого правила вывода, что и происходит, но при этом помечается, что в тексте программы имеется еще одно правило точно с таким же заголовком, то есть устанавливается первая точка возврата (назовем ее *1).

Так как было выбрано первое правило вывода, теперь необходимо последовательно доказать все цели, перечисленные в теле правила. Для доказательства цели middle (Animal) вновь начинается просмотр всех предложений, имеющихся в тексте программы, и находится факт middle (tiger). Поскольку имеется еще один факт, описывающий животное средних размеров middle (bear)., устанавливается вторая точка возврата (назовем ее *2).Переменная Animal получает значение tiger. Первая цель в теле правила успешно доказана.

Теперь выполняется переход к доказательству цели strong (tiger). Переменная Animal получила значение tiger при доказательстве предыдущей цели. Чтобы доказать цель strong (tiger), вновь начинается просмотр всех предложений, имеющихся в тексте программы, и находится факт strong (tiger), успешно доказывающий цель strong (tiger). Точка возврата не устанавливается, так как в тексте программы нет больше фактов strong, описывающих сильных животных.

Так как доказаны все цели в теле правила, считается успешно доказанной головная цель правила, и, следовательно, цель powerful (Animal), записанная в исходном запросе. Найдено первое решение: Animal=tiger.

Поскольку должны быть найдены все возможные решения, вступает в действие поиск с возвратом, который возвращает выполнение программы к последней установленной точке возврата – *2, то есть к цели middle (Animal), которая может быть передоказана. Вновь начинается просмотр всех предложений, но не с самого первого, а с того, на котором была установлена точка возврата *2 и цель middle (Animal) успешно передоказывается фактом middle (bear). Следует отметить, что переменная Animal, получившая при нахождении первого решения значение tiger, потеряла это значение, когда поиск с возвратом вернулся к передоказательству цели middle (Animal), то есть, нет никаких препятствий к тому, чтобы переменная Animal получила теперь значение bear.

Точка возврата *2 удаляется и вновь не устанавливается, так как нет более фактов middle, описывающих животных средних размеров. Итак, успешно передоказана первая цель в теле правила, и восстанавливается исходный порядок действий, то есть выполняется переход к доказательству второй цели в теле правила, но только теперь это цель strong (bear). Найти факт, доказывающий данную цель, не удается, то есть считается недоказанной вторая цель в теле правила, следовательно, вновь в действие вступает поиск с возвратом и происходит возврат к ближайшей точке возврата, а это точка *1. Точка возврата *1 свидетельствует о том, что вновь начинается просмотр предложений в тексте программы, но не с самого начала, а с предложения, помеченного этой самой точкой *1. При просмотре обнаруживается, что цель в запросе powerful (Animal) может быть передоказана с помощью второго правила вывода.

Так как выполнен возврат к последней точке возврата, переменная Animal вновь теряет свое значение, и, так как больше возможностей для передоказательства цели в запросе powerful (Animal) нет, точка возврата *1 более не устанавливается.

Теперь вновь повторяются действия, похожие на действия, происходившие, когда для доказательства использовалось первое правило вывода, только действий будет немного меньше, так как в теле второго правила всего одна цель.

Итак, цель запроса powerful (Animal) была сопоставлена с заголовком второго правила, что привело к необходимости доказать единственную цель в теле правила – big (Animal). Для этого вновь начинается просмотр предложений в тексте программы с самого начала и обнаруживается факт big (elephant). Вновь устанавливается точка возврата, назовем ее *3, говорящая о том, что цель big (Animal) в дальнейшем может быть передоказана. Факт big (elephant). успешно доказывает цель big (Animal) и переменная Animal принимает значение elephant. Так как успешно доказаны все (в данном случае одна) цели в теле правила, считается успешно доказанной и заголовочная цель правила, что приводит к успеху в доказательстве цели в запросе, и вот оно, второе решение: Animal=elephant.

Так как успешно найдено очередное решение, возобновляются действия по поиску следующих возможных решений, ведь есть точка возврата *3, к которой можно вернуться и передоказать цель в теле правила big (Animal) (в процессе поиска с возвратом переменная Animal вновь теряет свое значение). Точка возврата *3 свидетельствует о том, что вновь начинается просмотр предложений в тексте программы, но не с самого начала, а с предложения, помеченного точкой *3. При просмотре обнаруживается, что цель в теле правила big (Animal) может быть передоказана с помощью факта big (hippopotamus). , что и делается, переменная Animal получает значение hippopotamus, точка возврата *3 удаляется и более не устанавливается, так как больше нет фактов, описывающих больших животных, и считается найденным очередное, третье, решение: Animal=hippopotamus.

Так как все возможные решения найдены, выполнение программы заканчивается.

Проведём трассировку (пошаговое выполнение) рассмотренного примера.

Условные обозначения для трассировки:

1. CALL – цель, которую нужно доказать.
2. RETURN – цель, которая успешно доказана.
3. REDO – поиск с возвратом.
4. FAIL – неудача в доказательстве.
5. * – точка возврата.
6. _ – переменная, не имеющая значения.

```
CALL:       powerful (_)
CALL:       middle (_)
RETURN:     *middle ("tiger")
CALL:       strong ("tiger")
RETURN:     strong ("tiger")
RETURN:     *powerful ("tiger")
REDO:       middle (_)
RETURN:     middle ("bear")
CALL:       strong ("bear")
FAIL:       strong ("bear")
REDO:       powerful (_)
CALL:       big (_)
RETURN:     *big ("elephant")
RETURN:     powerful ("elephant")
REDO:       big (_)
RETURN:     big ("hippopotamus")
RETURN:     powerful ("hippopotamus")
```

Теперь можно сформулировать основные правила поиска с возвратом:

7. Цели должны быть доказаны по порядку, слева, направо.

8. Для доказательства некоторой цели предложения просматриваются в том порядке, в каком они появляются в тексте программы.

9. Для того, чтобы доказать головную цель правила, необходимо доказать цели в теле правила. Тело правила состоит, в свою очередь из целей, которые должны быть доказаны.

10. Цель считается доказанной, если с помощью соответствующих фактов доказаны все цели, находящиеся в листьевых вершинах дерева целей.

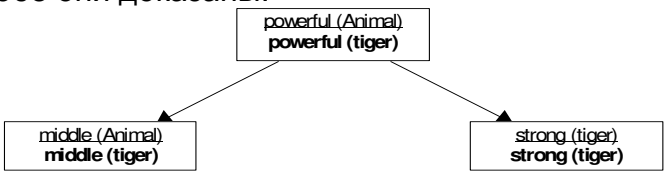
Для последнего правила следует пояснить, что называется деревом целей. Ход решения программы удобно представлять в виде дерева, которое называется деревом целей.

Пример дерева целей для ранее рассмотренного примера, для случая нахождения первого решения Animal=tiger.

Условные обозначения в дереве целей:

1. powerful (Animal) – цель, которую нужно доказать.
2. powerful (tiger) – цель, которая успешно доказана.

В данном дереве целей две цели: middle (Animal) и strong (tiger), находящиеся в листьевых вершинах, и обе они доказаны.



Управление поиском с возвратом: предикаты «"!» и «fail»

Управление поиском с возвратом заключается в решении двух задач: включении поиска с возвратом при его отсутствии, и отключении поиска с возвратом при его наличии.

Для решения этих задач используются два стандартных предиката:

1. предикат fail, включающий поиск с возвратом.

2. предикат ! (этот предикат еще называют «отсечение»), предотвращающий поиск с возвратом.

Рассмотрим, как работает предикат fail. Для начала следует напомнить, что поиск с возвратом начинает свою работу только в том случае, если не удается доказать какую-либо цель. Поэтому действует предикат fail очень просто – цель с использованием данного предиката НИКОГДА не доказывается, а, следовательно, всегда включается поиск с возвратом.

Для получения такого же эффекта можно записать, например, вот такую цель: 2=3. Эффект будет абсолютно тем же самым. Предикат fail используется в тех случаях, когда в программе есть внутренняя цель, и необходимо позаботиться о нахождении всех возможных решений. Рассмотрим простой пример: вывод названий всех стран, перечисленных в фактах.

```
PREDICATES
country (string)
print
CLAUSES
country (“Финляндия”).
country (“Швеция”).
country (“Норвегия”).
print:- country (Country_name), write (Country_name), nl.
GOAL
print.
```

Результатом работы программы будет вывод только названия первой страны из перечня фактов – Финляндии. Произойдет это из-за того, что при использовании внутренней цели находится только первое решение задачи.

Для того чтобы в процессе выполнения программы был выведен полный перечень названий стран, необходимо, чтобы цель country (Country_name) была передоказана столько раз, сколько имеется фактов в секции CLAUSES. Эта цель достигается очень просто. Предложение для предиката print нужно лишь переписать следующим образом:

```
print:- country (Country_name), write (Country_name), nl, fail.
```

В таком случае данное правило будет работать следующим образом: первый раз цель country (Country_name) будет успешно доказана с помощью факта country (“Финляндия”). и переменная Country_name будет конкретизирована значением “Финляндия”. Затем будет выведено значение переменной Country_name и наступит черед цели fail, которая никогда не доказывается.

Естественно, будет инициализирован поиск с возвратом, и возврат будет выполнен к ближайшей цели, которую можно передоказать. (Следует отметить, что ближайшей считается цель, встреченная при возврате, то есть при движении справа налево.) Эта цель – country (Country_name). Так как заработал поиск с возвратом, переменная Country_name теряет свое значение, и ничто не препятствует успешному передоказательству цели country (Country_name) фактом country (“Швеция”). Подобные действия будут повторяться до тех пор, пока не будут исчерпаны все факты, используемые для доказательства.

В результате будет выведен список всех стран, то есть программа выполнит те действия, которых от нее ждали. Однако следует сделать небольшое, но важное замечание. Несмотря на то, что программа выполнила все ожидаемые действия, в итоге выполнения программы завершится с неуспехом, поскольку цель print из секции GOAL доказана не будет. Недоказательство цели print произойдет вот по какой причине. Когда цель country (Country_name) будет в последний раз успешно доказана фактом country (“Норвегия”). в действие вновь вступает цель fail. Но передоказать цель country (Country_name) более нельзя, все факты исчерпаны и, так как не удалось доказать цель в теле правила и головная цель правила считается недоказанной, следовательно будет считаться недоказанной цель print из секции GOAL.

Избежать этого изъяна в работе программы очень легко, следует всего лишь добавить еще одно предложение для предиката print. Следующий вариант примера будет выполнять все необходимые действия, и выполнение программы будет завершаться с успехом.

```
PREDICATES
country (string)
print
CLAUSES
country (“Финляндия”).
country (“Швеция”).
country (“Норвегия”).
print:- country (Country_name), write (Country_name), nl, fail.
print.
GOAL
print.
```

В данном примере наличие второго предложения для предиката print создает еще одну точку возврата. Когда цель fail в очередной раз своим недоказательством включает поиск с возвратом, передоказать цель country (Country_name) более нельзя, все факты исчерпаны, вот тогда поиск с возвратом и возвращается к передоказательству цели print, что с успехом делается с помощью второго предложения для предиката print.

Еще один пример, показывающий, как можно управлять поиском с возвратом без предиката fail. В приведенном примере выполняется вывод положительных чисел до первого встреченного отрицательного числа. В этом случае работа программы прекращается.

```
PREDICATES
number (integer)
output
CLAUSES
number (2).
number (1).
number (0).
number (-1).
number (-2).
output:- number (Positive_number), write (Positive_number), nl, Positive_number<0.
output.
GOAL
output.
```

В данном случае цель Positive_number<0 играет роль, если так можно выразиться, условного предиката fail. Пока цель number (Positive_number) будет доказываться фактами, содержащими положительные числа, цель Positive_number<0 доказываться не будет, и, сдедовательно, будет работать поиск с возвратом. Как только переменная Positive_number будет конкретизирована значением –1, цель Positive_number<0 будет успешно доказана, завершится успехом доказательство всего правила и цели output в секции GOAL. В этом примере второе предложение для предиката output требуется только на тот случай, если бы все факты number содержали бы только положительные числа.

Еще одно средство для управления поиском с возвратом – это стандартный предикат ! (отсечение). Действие этого предиката прямо противоположно действию предиката fail. Если предикат fail всегда включает поиск с возвратом, то отсечение поиск с возвратом прекращает.

Рассмотрим, как работает отсечение, на примере. Пусть имеется набор фактов, описывающих некоторые числа.

```
PREDICATES
tens (string)
ones (string)
numbers
CLAUSES
```

```
tens ("двадцать").
tens ("тридцать").
ones ("два").
ones ("три").
numbers:- tens (Tens_number), ones(Ones_number),
write (Tens_number, " ", Ones_number), nl, fail.
numbers.
GOAL
numbers.
```

Результатом работы программы будет вывод следующих строк:

```
двадцать два
двадцать три
тридцать два
тридцать три
```

(выполнение программы завершится с успехом)

В данном случае в программе будет две точки возврата, которые и дадут этот эффект. Вместо подробного описания работы программы ниже дается трассировка (с некоторыми несущественными сокращениями), из которой становится ясной логика работы программы. Следует заметить, что выполнение программы завершится с успехом.

```
CALL:    numbers ()
CALL:    tens (_)
RETURN:  *tens ("двадцать")
CALL:    ones (_)
RETURN:  *ones ("два")
         write ("двадцать", " ", "два")
REDO:    ones (_)
RETURN:  ones ("три")
         write ("двадцать", " ", "три")
REDO:    tens (_)
RETURN:  tens ("тридцать")
CALL:    ones (_)
RETURN:  *ones ("два")
         write ("тридцать", " ", "два")
REDO:    ones (_)
RETURN:  ones ("три")
         write ("тридцать", " ", "три")
REDO:    numbers ()
RETURN:  numbers ()
```

Если теперь добавить в правило вывода отсечение,

```
numbers:- tens (Tens_number), !, ones (Ones_number),
write (Tens_number, " ", Ones_number), nl, fail.
```

то это существенно изменит результат работы программы:

```
двадцать два
двадцать три
```

(выполнение программы завершится с неуспехом).

Рассмотрим, как работает отсечение. Когда выполняется последовательное доказательство целей слева направо, то цель «!» (отсечение) доказывается всегда и при этом выполняется еще одно очень важное действие – отсечение уничтожает все точки возврата, которые остались слева от отсечения. Следовательно, когда цель «!» была успешно доказана, точка возврата для цели tens (Tens_number) была уничтожена, а с точкой возврата для цели ones (Ones_number) ничего не произошло.

Когда предикат fail инициализировал поиск с возвратом, «в живых» осталась только одна точка возврата, для цели ones (Ones_number), то есть только для этой цели перебирались все возможные решения. Другими словами, после того, как доказательство целей миновало отсечение, поиск с возвратом возможен только справа от отсечения (нужно отметить, что, до тех пор, пока цель отсечение не доказана, поиск с возвратом возможен и слева от цели !). Пример трассировки для второго варианта правила.

```
CALL:    numbers ()
```

```
CALL:    tens (_)
RETURN:  *tens ("двадцать")
CALL:    ones (_)
RETURN:  *ones ("два")
         write ("двадцать", " ", "два")
REDO:    ones (_)
RETURN:  ones ("три")
         write ("двадцать", " ", "три")
```

В целом выполнение программы завершается с неуспехом, так как отсечение уничтожило не только возможность возврата к передоказательству цели tens (Tens_number), но и цели numbers. Перестановка отсечения в правиле будет существенно изменять результаты работы программы, и при наличии отсечения приведенный пример всегда будет завершаться с неуспехом.

```
numbers:- tens (Number1), ones (Number2), !, write (Number1, " ", Number2), nl, fail.
двадцать два
(выполнение программы завершится с неуспехом)
numbers:- !, tens (Number1), ones (Number2), write (Number1, " ", Number2), nl, fail.
двадцать два
двадцать три
тридцать два
тридцать три
(выполнение программы завершится с неуспехом)
```

Отсечение весьма полезно при организации ветвления с помощью нескольких правил с одной и той же целью в заголовке правила. Пример программы, которая проверяет, делится ли введенное число нацело на 2, на 3 или на 5.

```
PREDICATES
division (integer)
start (string)
CLAUSES
division (N):- N mod 2 = 0, !, write (N, “делится на 2 без остатка.”), nl.
division (N):- N mod 3 = 0, !, write (N, “делится на 3 без остатка.”), nl.
division (N):- N mod 5 = 0, !, write (N, “делится на 3 без остатка.”), nl.
division (_):- write (“Ваше число не делится нацело ни на 2, ни на 3, ни на 5!!!”).
GOAL
write (“Ваше число? ”), readint (Number), division (Number).
```

В рассмотренном примере отсечение убирает совершенно ненужные точки возврата. Предположим, пользователь ввел число 1023. В этом случае в первом правиле для предиката division условие N mod 2 = 0 доказано не будет, следовательно, будет выполнен откат ко второму правилу. Препятствий для отката нет, так как отсечение в первом правиле не сработало. Во втором правиле условие N mod 3 = 0 успешно доказывается, и в этом случае доказательство проходит через отсечение. Как уже упоминалось, отсечение всегда доказывается с успехом, и будут уничтожены все точки возврата, проставленные к моменту доказательства цели !, оказавшиеся совершенно ненужными. Действительно, ведь если условие N mod 3 = 0 верно, а то, что N не делится нацело на 2, было проверено с помощью первого правила, третье и четвертое правила для предиката division точно не понадобится, то есть и не нужно сохранять бесполезную точку возврата.

Если при написании программы есть возможность выносить проверку некоторого условия непосредственно в заголовок правила, лучше это сделать. Например, программа, проверяющая, является ли введенное число равным 1, 2 или 3, может быть написана следующим образом:

```
…
choice (N):- N=1, !, write (“Это единица.”).
choice (N):- N=2, !, write (“Это двойка.”).
choice (N):- N=3, !, write (“Это тройка.”).
choice (N):- write (“Это не единица, не двойка, не тройка!!!”).
…
```

Более кратко правила можно записать с проверкой значения N не отдельным условием, а непосредственно в заголовке правила:

```
…
choice (1):- !, write (“Это единица.”).
choice (2):- !, write (“Это двойка.”).
choice (3):- !, write (“Это тройка.”).
choice (_):- write (“Это не единица, не двойка, не тройка!!!”).
…
```

Анонимная переменная в заголовке последнего правила говорит о том, что значение переменной роли не играет. Действительно, если дело дошло до последнего правила, значит, значение переменной не равно 1, 2 или 3.

Всегда следует убирать ненужные точки возврата как можно раньше.

Итак, поиск с возвратом возможен только в случае, если в предложении есть цели, которые можно передоказать. Как же быть, если поиск с возвратом необходим для решения задачи, но нет целей, которые можно передоказывать? В такой ситуации приходиться создавать точку возврата искусственно, используя специальный предикат, для которого должны быть определены два предложения. Цель, записанная с использованием данного предиката, должна соответствовать двум условиям: не выполнять никаких видимых действий и ВСЕГДА генерировать точку возврата. Такой специальный предикат определяется следующим образом (предикат не является стандартным и его имя может быть выбрано совершенно произвольно):

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

Рассмотрим пример: вывод приглашения «>», ввод с клавиатуры строки и вывод ее на экран до тех пор, пока не будет введена строка stop.

```
PREDICATES
repeat
echo
CLAUSES
repeat.
repeat:- repeat.
echo:- repeat, write (“> ”), readln (String), write (String), nl, String=”stop”.
GOAL
echo.
```

Если написать правило вывода без цели repeat,

```
echo:- write (“> ”), readln (String), write (String), nl, String=”stop”.
```

будет выполнен ввод и вывод только первой и единственной введенной строки, неравной “stop”. Действительно, после ввода и вывода строки будет выполнена проверка String=”stop”, которая завершится неуспехом, что приведет к включению поиска с возвратом, но ни одну из целей в правиле передоказать нельзя (точки возврата не были проставлены), поэтому выполнение программы завершится неуспехом.

Рассмотрим вариант правила с использованием цели repeat. При первом доказательстве цели repeat она успешно доказывается с помощью факта repeat. , при этом проставляется точка возврата, так что, если в дальнейшем не будет доказана какая-либо цель, можно будет вернуться к цели repeat и передоказать ее с помощью правила вывода repeat:- repeat.

Далее выполняется успешное последовательное доказательство всех целей, вплоть до цели String=”stop”, которая, в случае, если была введена строка, неравная ”stop”, не доказывается. Как известно, недоказательство некоторой цели приводит к включению поиска с возвратом. В данном примере цели write (“> ”), readln (String), write (String) и nl передоказать нельзя, предикат repeat же определен таким образом, что всегда может быть передоказан.

Цель repeat успешно передоказывается (вновь с генерацией точки возврата) и возобновляется естественный порядок доказательства целей, слева направо. Таким

образом, организовываются повторяющиеся действия с помощью поиска с возвратом в том случае, когда исходно не было целей, дающих точку возврата.

4. Организация циклов: механизм отката и рекурсия

Рекурсия – это второе средство для организации повторяющихся действий в Prolog'е. *Рекурсивная процедура* – это процедура, вызывающая сама себя до тех пор, пока не будет соблюдено некоторое условие, которое остановит рекурсию. Такое условие называют граничным. Рекурсия – хороший способ для решения задач, содержащих в себе подзадачу такого же типа.

Пример рекурсии: найти факториал n!.

Задача нахождения значения факториала n! очень хорошо решается с помощью рекурсии, поскольку может быть сведена к решению аналогичной подзадачи, которая, в свою очередь, сводится к решению аналогичной подподзадачи и т.д.

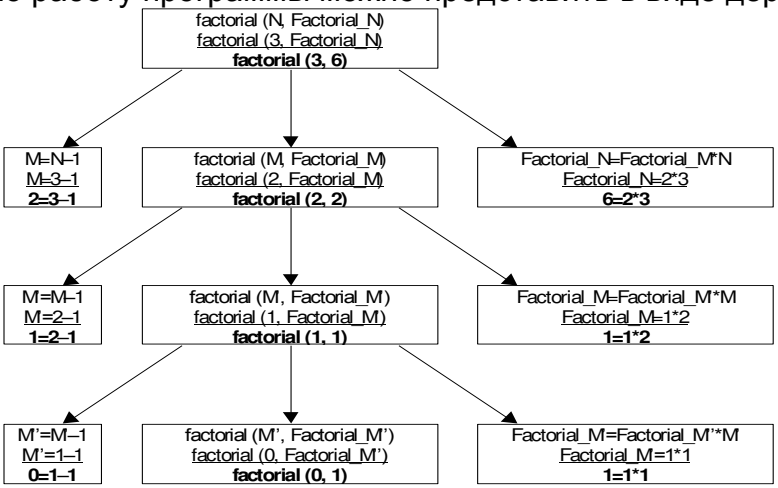
Действительно, чтобы найти значение факториала n!, можно найти значение факториала (n–1)! и умножить найденное значения на n. Для нахождения значения факториала (n–1)! можно пойти по уже известному пути – найти значение факториала (n–2)! и умножить найденное значения на n–1. Так можно действовать до тех пор, пока не доберемся до нахождения значения факториала (n–n)! или другими словами, факториала 0!. Значение факториала 0! известно – это 1. Вот это и будет граничное условие, которое позволит остановить рекурсию. Все, что теперь остается – это умножить полученную 1 на (n-(n-1)), затем на (n-(n-2)) и т.д. столько раз, сколько было рекурсивных вызовов. Результат n! получен.

Вот как выглядит программа, которая проделывает вычисление n! (нужно заметить, что предложения Prolog-программы достаточно точно повторяют формулировку задачи на естественном языке).

```
PREDICATES
factorial (integer, integer)
CLAUSES
%факториал 0! равен 1
factorial (0, 1):- !.
%факториал n! равен факториалу (n–1)!, умноженному на n
factorial (N, Factorial_N):- M=N–1, factorial (M, Factorial_M), Factorial_N=Factorial_M*N.
GOAL
write ("Для какого числа Вы хотите найти факториал? "), readint (Number), factorial (Number, Result), write (Number, "!=", Result).
```

Результат работы программы: 3!=6

Более наглядно работу программы можно представить в виде дерева целей.



Непременно нужно отметить очень важную роль отсечения в первом предложении программы. Видимых действий здесь отсечение не производит, если его убрать, то есть записать первое предложение как факт, результат работы программы останется тем же.

В чем же тогда смысл отсечения в этом примере? Отсечение убирает совершенно ненужную точку возврата, которая в дальнейшем может доставить много хлопот, если на нее не обратить внимания и оставить.

Это утверждение можно пояснить на примере. Выполнение программы начинается с попытки доказательства цели, например, factorial (3, Result). Для доказательства данной цели будет использовано второе правило вывода, так как в первом правиле нет совпадения по первому аргументу (3<>0), что приведет, естественно, к последовательному доказательству хвостовых подцелей правила. В процессе этого доказательства нужно будет доказать рекурсивную цель factorial (2, Factorial_M), что вновь приведет к использованию второго правила вывода (в действие вступает рекурсия), так как первое правило по прежнему не подходит для доказательства из-за несовпадения первого аргумента. Подобные действия будут продолжаться до тех пор, пока рекурсивная цель не примет вид factorial (0, Factorial_M''). Вот теперь наступил ключевой момент!

Впервые для доказательства рекурсивной цели будет использовано первое правило. При этом цель factorial (0, Factorial_M'') будет успешно, с помощью первого правила, доказана, но при этом остается потенциальная возможность использовать для доказательства той же самой цели второе правило. Другими словами, будет поставлена точка возврата. Но никакого передоказательства в дальнейшем не потребуется! Значение 0! всегда равно 1!.

Вот часть трассировки выполняемой программы для случая БЕЗ отсечения:

```
factorial (0, 1).
factorial (N, Factorial_N):- M=N–1, factorial (M, Factorial_M), Factorial_N=Factorial_M*N.
…
CALL:     factorial (2, _)
REDO:     factorial (2, _)
          1=1
CALL:     factorial (1, _)
REDO:     factorial (1, _)
          0=0
CALL:     factorial (0, _)
RETURN:   *factorial (0, 1)        вот он, ключевой момент! цель доказана,
…                                  но * говорит о том, что поставлена точка возврата.
```

Теперь часть трассировки выполняемой программы для случая С отсечением:

```
factorial (0, 1):- !.
factorial (N, Factorial_N):- M=N–1, factorial (M, Factorial_M), Factorial_N=Factorial_M*N.
…
CALL:     factorial (2, _)
REDO:     factorial (2, _)
          1=1
CALL:     factorial (1, _)
REDO:     factorial (1, _)
          0=0
CALL:     factorial (0, _)
RETURN:   factorial (0, 1)         цель доказана,
…                                  но точка возврата НЕ поставлена.
```

Каковы могут быть последствия, если точку возврата не убрать с помощью отсечения. Предположим, предикат factorial используется в каком-нибудь правиле вывода, например:

```
…
factorial (0, 1).
factorial (N, Factorial_N):- M=N–1, factorial (M, Factorial_M), Factorial_N=Factorial_M*N.
calculate (N, Res):- factorial (N, Res), Res<1000, !, write (“Все в порядке!”).
calculate (_, _):- write (“Слишком большое число!”).
```

…

Что же произойдет, если будет рассчитываться, например, факториал 7! (7!=5040)? Цель factorial (N, Res) будет успешно доказана, но следующая цель Res<1000 доказана не будет, то есть начнется поиск с возвратом, который как известно, возвращается к ближайшей точке возврата. В данном примере эта точка оставлена в предикате factorial. То есть, вместо того, чтобы вывести сообщение “Слишком большое число!”, как задумал автор программы, программа проваливается в бесконечную рекурсию и выполнение программы прекращается из-за переполнения стека.

Для иллюстрации вышесказанного приводится выдержка из трассировки:

```
…
CALL:     calculate(7, _)
CALL:     factorial (7, _)
FAIL:     factorial (7, _)
REDO:     factorial (7, _)
          6=6
CALL:     factorial (6, _)
FAIL:     factorial (6, _)
REDO:     factorial (6, _)
…
REDO:     factorial (1, _)
          0=0
CALL:     factorial (0, _)
RETURN:   *factorial (0,1)                оставшаяся точка возврата
          1=1
RETURN:   factorial (1,1)
…
RETURN:   factorial (6,720)
          5040=5040
RETURN:   factorial (7,5040)
          5040<1000
FAIL:     calculate(7, _)                 неудача, начинается поиск с возвратом
REDO:     factorial (0, _)                а вот и последствия, возвращаемся и
          -1=-1                           проваливаемся в бесконечную рекурсию
CALL:     factorial (-1, _)
FAIL:     factorial (-1, _)
REDO:     factorial (-1, _)
          -2=-2
CALL:     factorial (-2, _)
FAIL:     factorial (-2, _)
REDO:     factorial (-2, _)
          -3=-3
CALL:     factorial (-3, _)
…
```

Теперь, для сравнения, как все происходит, если точка возврата ликвидирована с помощью отсечения:

```
…
factorial (0, 1):- !.
factorial (N, Factorial_N):- M=N–1, factorial (M, Factorial_M), Factorial_N=Factorial_M*N.
calculate (N, Res):- factorial (N, Res), Res<1000, !, write (“Все в порядке!”).
calculate (_, _):- write (“Слишком большое число!”).
…
```

Вновь пример трассировки:

```
…
REDO:     factorial (1, _)
          0=0
CALL:     factorial (0, _)
RETURN:   factorial (0,1)                 точки возврата нет!!!
          1=1
```

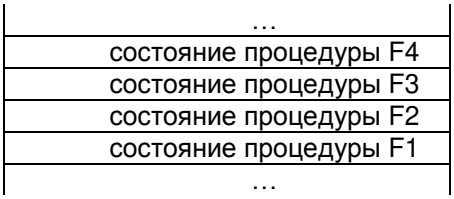
```
RETURN:     factorial (1,1)
…
RETURN:     factorial (6, 720)
            5040=5040
RETURN:     factorial (7, 5040)
            5040<1000
FAIL:       calculate (7, _)
REDO:       calculate (7, _)
            write("Слишком большое число!")     работает, как и было задумано!!!
…
```

Вывод из всего вышеизложенного – никогда не оставлять ненужные точки возврата, убирая их с помощью отсечения. Если программа объемная, то достаточно сложно понять, куда выполняется откат и где оставлена лишняя точка возврата.

Как видно, рекурсивные задачи, решаемые с помощью Prolog'а, отличаются краткостью и приближенностью к естественному языку.

Но таким образом организованная рекурсия имеет один, но существенный недостаток. При достаточно глубокой рекурсии переполняется стек вызовов, и выполнение программы аварийно завершается. Можно ли избежать такой ситуации? Да, это возможно, при специальным образом организованной рекурсии. Такая рекурсия называется хвостовой. Прежде чем перейти к рассмотрению хвостовой рекурсии, рассмотрим для начала следующий пример.

Предположим, имеются некоторые абстрактные процедуры F1, F2, F3 и F4. Пусть в процессе выполнения процедуры F1 выполняется вызов процедуры F2, при выполнении которой, в свою очередь, вызывается процедура F3, которая, в свою очередь, вызывает F4. В этом случае стек вызовов будет выглядеть следующим образом:



Сохранять состояние вызывающей процедуры необходимо для того, чтобы продолжить ее выполнение после завершения вызова. Но, а если вызов процедур F2, F3 и F4 будет последним действием в вызывающих процедурах? Тогда можно не сохранять состояние вызывающей процедуры, так как в ней после завершения вызова больше ничего выполняться не будет. Можно хранить в стеке вызовов только состояние последней вызванной процедуры, другими словами подменять состояние бывшей процедуры состоянием новой процедуры.

То есть считать, что процедура F1 вызывает процедуру F4 непосредственно. В этом случае, не расходуется стек вызовов.

Теперь перейдем к варианту с рекурсивными вызовами процедур. Пусть в процессе выполнения процедуры F1 выполняется вызов процедуры F2, при выполнении которой, в свою очередь, рекурсивно вызывается процедура F2, которая, в свою очередь, вновь рекурсивно вызывает саму себя, то есть опять вызывает F2. В этом случае стек вызовов будет хранить только состояние последнего рекурсивного вызова, то есть стек вновь не будет переполняться!

Выполнение таким образом организованной рекурсии не будет завершаться аварийно из-за переполнения стека, сколько бы ни было рекурсивных вызовов! Аварийное завершение по другим причинам, конечно, не исключается.

Можно на практике убедиться, что это действительно так. Попробуйте запустить следующую программу и посмотреть, в течение какого времени она будет благополучно работать:

```
PREDICATES
tail_recursion
CLAUSES
```

```
%хвостовая рекурсия
tail_recursion:- write (“*”), tail_recursion.
GOAL
tail_recursion.
```

Теперь можно сформулировать условия, при соблюдении которых рекурсия в Prolog’е становится хвостовой, то есть не расходует стек при неограниченном количестве рекурсивных вызовов:

1. рекурсивный вызов должен быть последней целью в хвостовой части правила вывода.

2. перед рекурсивным вызовом не должно быть точек возврата (это условие хвостовой рекурсии специфично для Prolog’а).

Если первое условие очевидно, то необходимость выполнения второго условия может быть, на первый взгляд, не совсем понятна. Чтобы рекурсия была хвостовой, необходимо, чтобы доказательство рекурсивного вызова было действительно последним действием в хвостовой части правила. А если до рекурсивного вызова имеется цель, которую можно передоказать, то есть имеется точка возврата? Тогда придется сохранять в стеке состояние вызывающего правила! Вдруг в дальнейшем, где-то в глубинах рекурсии какая-либо цель не будет доказана? Тогда будет включен поиск с возвратом к ближайшей точке возврата, для чего и нужно сохранять состояние в стеке соответствующего правила (чтобы знать, куда вернуться).

Если соблюдение первого условия сложности не представляет (легко проконтролировать, чтобы рекурсивный вызов был последней целью в теле правила), то как быть уверенным в соблюдении второго условия, в отсутствии точек возврата до рекурсивного вызова?

Соблюсти второе условие очень просто. Достаточно перед рекурсивным вызовом поставить отсечение. Только и всего! Конечно, использовать отсечение следует как можно раньше в теле правила, но, в крайнем случае, его можно использовать в качестве предпоследней цели (последняя цель, естественно, рекурсивный вызов)

Примеры нехвостовой рекурсии и ее преобразования в хвостовую:

-пример нехвостовой рекурсии

```
PREDICATES
counter (integer)
CLAUSES
counter (N):- write (“N=”, N), nl, New_N=N+1, counter (New_N), nl.
GOAL
counter (0).
```

Естественно, приведенный пример не является примером хвостовой рекурсии, однако получить хвостовую рекурсию достаточно просто: нужно всего лишь убрать цель nl (переход на новую строчку) после рекурсивного вызова.

-пример хвостовой рекурсии

```
PREDICATES
counter (integer)
CLAUSES
%выполнение программы аварийно завершится из-за переполнения
%разрядной сетки для integer, но не из-за переполнения стека
counter (N):- write (“N=”, N), nl, New_N=N+1, counter (New_N).
GOAL
counter (0).
```

В рассмотренном примере соблюдены оба условия хвостовой рекурсии: рекурсивный вызов последний в теле правила и нет точек возврата перед рекурсивным вызовом. Действительно, цели write (“N=”, N), nl и New_N=N+1 передоказать нельзя, соответственно, нет и точки возврата.

-пример нехвостовой рекурсии

```
PREDICATES
counter (integer)
CLAUSES
```

```
counter (N):- N>0, write (“N=”, N), nl, New_N=N–1, counter (New_N).
counter (N):- write (“Отрицательное N=”, N).
GOAL
counter (1000).
```

В приведенном примере рекурсия хвостовой не является из-за оставленной точки возврата. Пока N будет положительным, для доказательства рекурсивной подцели будет использоваться первое правило вывода, но ведь остается неиспользованным второе правило, что и дает точку возврата. Преобразовать нехвостовую рекурсию в хвостовую можно, убрав точку возврата с помощью отсечения. Первое правило нужно переписать следующим образом:

```
counter (N):- N>0, !, write (“N=”, N), nl, New_N=N–1, counter (New_N).
```

Если N – положительное число, второе правило заведомо не понадобится.

-пример хвостовой рекурсии

```
PREDICATES
counter (integer)
CLAUSES
counter (N):- N>0, !, write (“N=”, N), nl, New_N=N–1, counter (New_N).
counter (N):- write (“Отрицательное N=”, N).
GOAL
counter (1000).

%пример нехвостовой рекурсии
PREDICATES
counter (integer)
check (integer)
CLAUSES
counter (N):- check (N), write (“N=”, N), nl, New_N=N–1, counter (New_N).
check (N):- N>0, write (“Положительное ”).
check (_):- write (“Отрицательное ”).
GOAL
counter (1000).
```

В приведенном примере рекурсия также хвостовой не является вновь из-за оставленной точки возврата. Пока N будет положительным, для доказательства рекурсивной подцели будет использоваться первое правило вывода для предиката check, но ведь остается неиспользованным второе правило для того же самого предиката, что и дает точку возврата. Преобразовать нехвостовую рекурсию в хвостовую можно, убрав точку возврата с помощью отсечения, как и в первом примере. Первое правило для предиката counter можно переписать следующим образом:

```
counter (N):- check (N), !, write (“N=”, N), nl, New_N=N–1, counter (New_N).
```

Но более правильно, с точки зрения хорошего стиля программирования на Prolog’e, точку возврата лучше убрать в предложении для предиката check.

-пример хвостовой рекурсии

```
PREDICATES
counter (integer)
check (integer)
CLAUSES
counter (N):- check (N), write (“N=”, N), nl, New_N=N–1, counter (New_N).
check (N):- N>0, !, write (“Положительное ”).
check (_):- write (“Отрицательное ”).
GOAL
counter (1000).
```

Не всегда так легко и просто можно преобразовать нехвостовую рекурсию в хвостовую. Иногда для этого требуется полностью переписать программу. Рассмотрим уже известный пример вычисления факториала. Рекурсия в приведенном ранее примере, очевидно, хвостовой не является. Только с помощью применения отсечения добиться хвостовой рекурсии нельзя. Придется полностью преобразовать программу.

Тема. Составные объекты языка Пролог. Решение логических задач в Прологе

План

1. Составные объекты в Visual Prolog: списки и деревья.

a. Обработка списков в Прологе.

b. Работа с бинарными деревьями в Прологе.

2. Работа со строками (некоторые стандартные предикаты)

1. Составные объекты в Visual Prolog: списки и деревья

Составные объекты данных позволяют рассматривать информацию, состоящую из нескольких частей, как единое целое, в то же время предоставляя возможность получать доступ к отдельным частям этой информации.

Например, дата рождения человека может быть представлена как единый объект, и в то же время, дата состоит из трех частей, числа, месяца и года. Такую составную структуру можно представить как единое целое, использовав функтор, например, birthday. Имя функтора выбирается произвольно. Тогда структура будет выглядеть следующим образом:

```
birthday (25, “мая”, 1980)
```

Секция описания доменов будет выглядеть следующим образом:

```
DOMAINS
    day=birthday (integer, symbol, integer)
```

Такое описание домена позволит определить предикат, в котором в качестве аргумента можно использовать данные, принадлежащие домену day. Далее следует пример программы с использованием данного домена.

```
DOMAINS
    day=birthday (integer, string, integer)
    name=string
PREDICATES
    congratulation (name, day)
    print
    print2
CLAUSES
    congratulation (“Анна”, birthday (25, “мая”, 1980)).
    congratulation (“Иван”, birthday (17, “декабря”, 1957)).
    congratulation (“Петр”, birthday (30, “июля”, 2001)).
    print:- congratulation (Name, birthday (Day, Month, Year)), write (“С днем рождения ”, Day, Month, Year, ”, ”, Name,”!”), nl, fail.
    print.
    print2:- congratulation (Name, Birthday), write (“С днем рождения ”, Birthday, ”, ”, Name,”!”), nl, fail.
    print2.
GOAL
    print, print2.
```

Как видно из вышеприведенного примера, к составному объекту можно обратиться как единому целому, так и получить доступ к его составным элементам.

1 а). Обработка списков в Прологе

Список – это объект, который содержит конечное число других объектов. Список в Prolog’е можно приблизительно сравнить с массивами в других языках, но для списков нет необходимости заранее объявлять размерность.

Список в Prolog’е заключается в квадратные скобки, и элементы списка разделяются запятыми. Список, который не содержит ни одного элемента, называется пустым списком.

Примеры списков:

- список, элементами которого являются целые числа

[1, 2, 3]

- список, элементами которого являются символы

[one, two, three]

- список, элементами которого являются строки

[“One”, “Two”, “Three”]

- пустой список

[ ]

Для работы со списками с Prolog'е не существует отдельного домена, для того, чтобы работать со списком, необходимо объявить списочный домен следующим образом:

```
DOMAINS
    listdomain=elementdomain*
    elementdomain=…
```

listdomain – это произвольно выбранное имя нестандартного списочного домена, elementdomain – имя домена, которому принадлежит элемент списка, звездочка * как раз и обозначает, что выполнено объявление списка, состоящего из элементов домена element. При работе со списками нельзя включать в список элементы, принадлежащие разным доменам. В таком случае нужно воспользоваться составным доменом.

Примеры объявления списочных доменов:

```
DOMAINS
%элементы списка – целые числа
    intlist=integer*
%элементы списка – символы
    symlist=symbol*
%элементы списка – строки
    strlist=string*
%элементы списка – или целые числа, или символы, или строки
    mixlist=mixdomain*
    mixdomain=int(integer); sym(symbol); str(string)
```

Обратите внимание, что при объявлении составного домена были использованы функторы, так как объявление вида mixdomain=integer; symbol; string привело бы к ошибке.

Список является рекурсивным составным объектом, состоящим из двух частей. Составные части списка:

1. Голова списка – первый элемент списка;

2. Хвост списка – все последующие элементы, являющиеся, в свою очередь списком.

Примеры голов и хвостов списков:

[1, 2, 3] голова списка – 1, хвост списка – [2, 3]
[1] голова списка – 1, хвост списка – [ ]
[ ] пустой список нельзя разделить на голову и хвост

В Prolog'е используется специальный символ для разделения списка на голову и хвост – вертикальная черта |.

Например:

[1, 2, 3] или [1 | [2, 3]] или [1 | [2| [3]]] или [1 | [2 | [3 | [ ]]]]
[1] или [1 | [ ]]

Вертикальную черту можно использовать не только для отделения головы списка, но и для отделения произвольного числа начальных элементов списка:

[1, 2, 3] или [1, 2 | [3]] или [1, 2, 3 | [ ]]

Примеры сопоставления и унификации в списках:

[1, 2, 3]=[Elem1, Elem2, Elem3]	Elem1=1, Elem2=2, Elem3=3
[1, 2, 3]=[Elem1, Elem2, Elem3 | T]	Elem1=1, Elem2=2, Elem3=3, T=[ ]
[1, 2, 3]=[Head | Tail]	Head=1, Tail=[2, 3]
[1, 2, 3]=[Elem1, Elem2 | T]	Elem1=1, Elem2=2, T=[3]
[1, 2, 3]=[4 | T]	ошибка
[ ]=[H | T]	ошибка

Так как список является рекурсивной структурой данных, то для работы со списками используется рекурсия. Основной метод обработки списков заключается в следующем: отделить от списка голову, выполнить с ней какие-либо действия и перейти к работе с хвостом списка, являющимся в свою очередь списком. Далее у хвоста списка отделить голову и так далее до тех пор, пока список не останется

пустым. В этом случае обработку списка необходимо прекратить. Следовательно, предикаты для работы со списками должны иметь, по крайней мере, два предложения: для пустого списка и для непустого списка.

Пример: поэлементный вывод списка, элементами которого являются целые числа, на экран (нужно отметить, что этот пример приводится в учебных целях, список вполне может быть выведен на экран как единая структура, например, GOAL List=[1, 2, 3], write(List)).

```
DOMAINS
    intlist=integer*
PREDICATES
    printlist (intlist)
CLAUSES
%если список пустой, то выводить не экран нечего
    printlist ([ ]):- !.
%если список непустой, то отделить от списка голову, напечатать ее и
%использовать тот же самый предикат для печати хвоста списка, то есть
%выполнить рекурсивный вызов предиката printlist, передав в качестве
%аргумента хвост
    printlist ([H | T]):- write (H), nl, printlist (T).
GOAL
    printlist ([1, 2, 3]).
```

Результат работы программы:

```
1
2
3
```

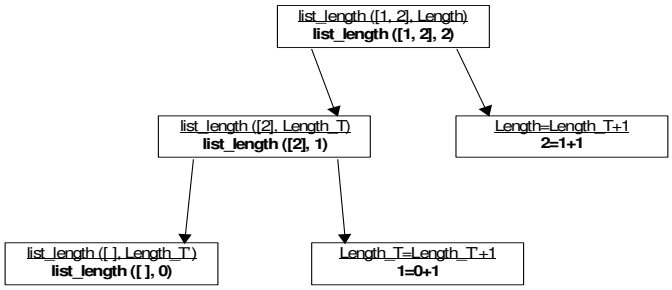
Еще один пример работы со списком – подсчет числа элементов списка или, другими словами, определение длины списка. Для того чтобы определить длину списка, вновь нужно рассмотреть два случая: для пустого и непустого списков. Если список пуст, то его длина равна 0, если же список не пуст, то определить его длину можно следующим образом: разделить список на голову и хвост, подсчитать длину хвоста списка и увеличить длину хвоста на единицу (то есть учесть отделенную предварительно голову списка.)

```
DOMAINS
    intlist=integer*
PREDICATES
    list_length (intlist, integer)
CLAUSES
%если список пустой, то его длина равна 0
    list_length ([ ], 0):- !.
%если список непустой, то его длина равна длине хвоста, увеличенной на 1
    list_length (List, Length):- List=[H | T], list_length (T, Length_T), Length=Length_T+1.
%более кратко второе правило вывода можно записать, перенеся разделение
%списка на голову и хвост в заголовок предложения и избавиться от лишней
%переменной List
%list_length ([H | T], Length):- list_length (T, Length_T), Length=Length_T+1.
GOAL
    listlength ([1, 2], Length), write("Length=", Length).
```

Результат работы программы:

```
Length=2
```

Для того чтобы лучше понять, каким образом работает приведенный пример, удобно воспользоваться деревом целей. В данном примере следует обратить внимание на то, каким образом формируется выходное значение – длина списка. Счетчик для подсчета числа элементов в списке обнуляется в момент, когда рекурсия останавливается, то есть список разбирается до пустого списка-хвоста и результат насчитывается при выходе из рекурсии.



Достаточно часто необходимо обработать список поэлементно, выполняя некоторые действия с элементами списка, в зависимости от соблюдения некоторых условий. Предположим, необходимо преобразовать список, элементами которого являются целые числа, инвертируя знак элементов списка, то есть положительные числа преобразовать в отрицательные, отрицательные в положительные, для нулевых значений никаких действий не предпринимать. Вновь придется рассмотреть два случая – для непустого и пустого списков. Преобразование пустого списка дает в результате такой же пустой список. Если же список не пуст, то следует рекурсивно выполнять отделение головы списка, ее обработку и рассматривать полученный результат как голову списка-результата.

```
DOMAINS
    intlist=integer*
PREDICATES
    inverting (intlist, intlist)
    processing (integer, integer)
CLAUSES
%обработка пустого списка дает, в результате, тоже пустой список
    inverting ([ ], [ ]):- !.
%если список непустой, отделить голову, обработать ее,
%и добавить в качестве головы списка-результата
    inverting ([H | T], [Inv_H | Inv_T]):- processing (H, Inv_H), inverting (T, Inv_T).
%предикат processing выполняет действия по обработке элемента списка в
%зависимости от его знака, предикат имеет два предложения, так как нужно
%рассмотреть два варианта: ненулевое и нулевое значения
    processing (0, 0):- !.
    processing (H, Inv_H):- Inv_H=-H.
GOAL
    inverting ([-2, -1, 0, 1, 2], Inv_List), write("Inv_List=", Inv_List).
```

Результат работы программы:

```
    Inv_List=[2, 1, 0, -1, -2]
```

Следующий пример рассматривает достаточно часто встречающуюся задачу – определение принадлежности элемента списку. Проверка принадлежности элемента списку выполняется достаточно просто – отделением головы списка и сравнением ее с искомым элементом. Если сравнение завершилось неудачей, продолжается поиск элемента в хвосте списка. Признаком наличия искомого элемента в списке будет успешное доказательство цели, если же цель не была доказана, значит, такого элемента в списке нет.

```
DOMAINS
    strlist=string*
PREDICATES
    member (string, strlist)
    search (string, strlist)
CLAUSES
%искомый элемент найден, его значение совпало со значением головы списка,
%хвост списка обозначен анонимной переменной, так как теперь хвост списка
%не важен
    member (Elem, [Elem | _]):- !.
%если элемент пока не обнаружен, попробовать найти его в хвосте списка,
%теперь для головы списка использована анонимная переменная, поскольку ее
%значение не важно, так как оно точно не равно искомому значению
    member (Elem, [_ | T]):- member (Elem, T).
%предикат search служит для двух целей: во-первых, чтобы сообщить о
```

```
%результатах поиска, и, во-вторых, чтобы при любом исходе поиска программа
%всегда заканчивала свое выполнение с успехом
    search (Elem, List):- member (Elem, List), !, write ("Элемент найден! :-) ").
    search (_, _):- write ("Элемент не найден! :-( ").
GOAL
    Cityes=["Москва", "Санкт-Петербург", "Омск", "Новосибирск", "Томск"], City="Новосибирск",
search (City, Cityes).
```

Результат работы программы:

```
    Элемент найден! :-)
```

Последний пример, который будет рассмотрен, это решение задачи соединения двух списков. Итак, каким образом можно объединить два списка? Предположим, имеется два двухэлементных списка: [1, 2] и [3, 4]. Объединение нужно начать с последовательного отделения голов первого списка до тех пор, пока первый список не станет пустым. Как только первый список станет пустым, его легко можно будет объединить со вторым, непустым, никоим образом к этому моменту не изменившимся списком. В результате, естественно, будет получен список, полностью совпадающий со вторым. Все, что осталось сделать, это добавить головы, отделенные от первого списка, ко второму списку. Вот как это выглядит в пошаговом описании:

1. отделяется голова от первого списка – [1, 2] –> [1 | [2]] (голова – 1, хвост – [2])
2. продолжается выполнение отделение головы, только теперь от полученного хвоста – [2] –> [2 | [ ]] (голова – 2, хвост – [ ])
3. когда первый список разобран до пустого, можно приступить к его объединению со вторым, непустым списком, объединяются пустой хвост [ ] и непустой второй список [3, 4] – получается тоже непустой список – [3, 4], теперь можно приступать к формированию объединенного списка, так как основа для списка-результата заложена, это его будущий хвост – [3, 4]
4. к хвосту списка-результата [3, 4] добавляется последняя отделенная от первого списка голова 2, что дает следующий список – [2, 3, 4]
5. все, что осталось сделать, это добавить к списку [2, 3, 4], который получился на предыдущем шаге, голову 1, которая была отделена самой первой и получается объединенный список [1, 2, 3, 4]

Теперь, собственно, текст программы:

```
DOMAINS
    intlist=integer*
PREDICATES
    append (intlist, intlist, intlist)
CLAUSES
%объединение пустого и непустого списков
    append ([ ], List, List):- !.
%объединение двух непустых списков
    append ([H | T], List, [H | App_T]):- append (T, List, App_T).
GOAL
    append ([1, 2], [3, 4], App_List), write("App_List=", App_List).
```

Результат работы программы:

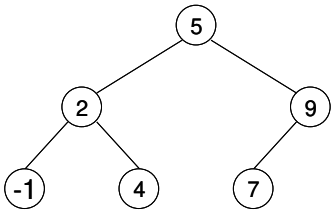
```
    App_List=[1, 2, 3, 4]
```

1 б). Работа с бинарными деревьями в Прологе

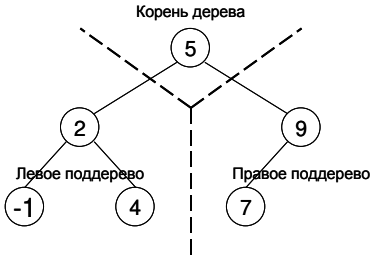
Дерево, как и список, также является рекурсивным составным объектом, но, в отличие от списка, в дереве можно выделить три составные части (сразу же следует оговорить, что далее речь пойдет только о двоичных, или бинарных деревьях). Составные части двоичного (бинарного) дерева:

1. Левое поддерево, являющееся, в свою очередь деревом;
2. Корень дерева;
3. Правое поддерево, также являющееся деревом.

В графическом виде дерево может быть представлено, например, так:



На следующем рисунке обозначены три составные части дерева.



Каждый узел дерева, приведенного в качестве примера, содержит целое число. Если для всего дерева соблюдается определенное правило расположения значений в узлах дерева, то такое дерево называют упорядоченным. Правило формулируется следующим образом: дерево можно назвать упорядоченным, если все значения, находящиеся в узлах левого поддерева, меньше значения в корне дерева, а все значения, находящиеся в узлах правого поддерева, больше значения в корне дерева.

Это правило должно соблюдаться как для левого, так и для правого поддеревьев, и для их левых и правых поддеревьев (ведь дерево – это рекурсивная структура, и как левое, так и правое поддеревья также являются полноценными деревьями).

Для дерева, приведенного выше в качестве примера, это правило соблюдается, следовательно, это дерево упорядочено.

5 больше –1, 2 и 4, и 5 меньше 7 и 9 (для дерева в целом)

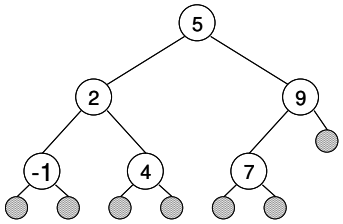
2 больше –1, и 2 меньше 4 (для левого поддерева)

9 больше чем 7 (для правого поддерева)

Узлы дерева, у которых нет левого и правого поддерева, называется листьевыми узлами или вершинами. В рассматриваемом дереве это узлы, которые содержат значения –1, 4 и 7. Отсутствующие у этих узлов левое и правое поддеревья можно назвать пустыми деревьями.

Пустое дерево – это дерево, в котором нет ни одного узла.

Если обозначить на рисунке пустые поддеревья, то это может выглядеть, например, так:



Заштрихованные кружочки как раз и обозначают пустые деревья, или, другими словами, отсутствующие левое и/или правое поддеревья у какого либо узла.

В Prolog’е для представления деревьев не существует отдельного домена, для того, чтобы работать с деревом, необходимо объявить новый нестандартный домен. При этом следует учитывать, что, во-первых, дерево – структура рекурсивная, во-вторых, дерево – структура, состоящая их трех частей, и, в-третьих, дерево может быть пустым или непустым. Все эти особенности учитываются в следующем объявлении домена:

```
DOMAINS
%домен_для_дерева=функтор_непустого_дерева(корень, левое_поддерево,
правое_поддерево) ;
% функтор_пустого_дерева()
```

```
treetype=tree (integer, treetype, treetype) ; empty ()
```

Treetype – это имя нестандартного домена для представления дерева, рекурсивность структуры дерева обеспечивается использованием рекурсивного описания домена, treetype дважды используется справа от знака равенства и описывает, соответственно, домен левого и правого поддеревьев.

Integer – описание домена для значения, находящегося в корневом узле дерева. Объединить в единое дерево три его составные части позволяет использование функтора tree с тремя аргументами. Для описания двух возможных состояний, в которых может находиться дерево, пустого и непустого, используются, соответственно, два функтора – tree и empty. Так как у пустого дерева нет ни корня, ни левого и правого поддеревьев, то и функтор empty не имеет аргументов (используются пустые скобки после empty). В этом случае пустые скобки можно и не использовать.

```
DOMAINS
    treetype=tree (integer, treetype, treetype) ; empty
```

При объявлении домена было использовано только одно зарезервированное слово – integer, все остальные имена выбраны произвольно. В следующем примере также описан домен для представления дерева, только использованы другие имена.

```
DOMAINS
    a=b (integer, a, a) ; c
```

Такое определение позволяет записать следующую структуру данных (для дерева, приведенного на рисунке):

```
tree                                                    (5,
  tree                                                  (2,
    tree                (-1,            empty,          empty),
    tree                (4,             empty,          empty)),
  tree                                                  (9,
    tree                (7,             empty,          empty),
    empty))
```

Вот пример программы, выводящей на экран дерево, как единый объект:

```
DOMAINS
    treetype=tree (integer, treetype, treetype) ; empty
GOAL
    OrderTree=tree (5, tree (2, tree (-1, empty, empty), tree (4, empty, empty)), tree (9, tree(7, empty,
empty), empty)), write (“Tree=”, OrderTree).
```

Результат работы программы:

```
    Tree=tree(5,tree(2,tree(-1,empty,empty),tree(4,empty,empty)),
tree(9,tree(7,empty,empty),empty))
```

Одной из наиболее частых операций, выполняемых с деревом, является исследование всех узлов дерева и обработка их некоторым образом. Последовательная обработка всех узлов дерева называется обходом дерева. В зависимости от того, в какой очередности обрабатываются корень дерева и его поддеревья, различают несколько направлений обхода дерева:

1. Корень, левое поддерево, правое поддерево обход «сверху вниз»;
2. Левое поддерево, правое поддерево, корень обход «снизу вверх»;
3. Левое поддерево, корень, правое поддерево обход «слева направо»;
4. Правое поддерево, корень, левое поддерево обход «справа налево»;

Так как дерево является рекурсивной структурой данных, то для работы с деревьями используется рекурсия. Основной метод обработки дерева заключается в следующем: разделить дерево на корень, левое поддерево и правое поддерево, выполнить какие-либо действия с корнем дерева, и перейти к обработке левого поддерева, а затем правого поддерева, являющимися, в свою очередь, деревьями. Обработка каждого из поддеревьев заключается в разделении их на три части и последовательной обработке каждой из частей.

Деление деревьев продолжается до тех пор, пока очередное поддерево не станет пустым. В этом случае обработку дерева необходимо прекратить. Следовательно, предикаты для работы с деревьями должны иметь, по крайней мере, два предложения: для пустого дерева и для непустого дерева.

Пример: поэлементный вывод дерева, в узлах которого хранятся целые числа, на экран (нужно отметить, что этот пример приводится в учебных целях, дерево вполне может быть выведено на экран как единая структура (см. предыдущий пример)).

```
DOMAINS
    treetype=tree (integer, treetype, treetype) ; empty
PREDICATES
    printtree (treetype)
CLAUSES
%если дерево пустое, то выводить не экран нечего
    printtree (empty):- !.
%если дерево непусто, то разделить дерево на корень, левое и правое поддеревья,
%напечатать корень и использовать тот же самый предикат для печати левого,
%а затем правого поддеревьев, то есть выполнить рекурсивные вызовы
%предиката printtree, передав в качестве аргумента сначала левое, а затем правое
%поддеревья
    printtree (tree (Root, Left, Right)):- write (Root), write (" ~ "), printtree (Left), printtree (Right).
GOAL
    printtree (tree (5, tree (2, tree (–1, empty, empty), tree (4, empty, empty)), tree (9, tree(7, empty, empty), empty))).
```

Результат работы программы обхода дерева «сверху вниз»:

```
5 ~ 2 ~ –1 ~ 4 ~ 9 ~ 7 ~
```

Если же второе предложение для предиката printlist переписать для выполнения обхода «слева направо»

```
printtree (tree (Root, Left, Right)):- printtree (Left), write (Root), write (" – "), printtree (Right).
```

то результат работы будет следующим, так как дерево упорядочено:

```
–1 ~ 2 ~ 4 ~ 5 ~ 7 ~ 9 ~
```

Еще один пример работы с деревом – подсчет числа узлов дерева. Для того чтобы определить количество узлов дерева, вновь нужно рассмотреть два случая: для пустого и непустого деревьев. Если дерево пусто, то количество узлов в таком дереве равно 0. Если дерево не пусто, то определить количество узлов в дереве в целом можно следующим образом: разделить дерево на корень, левое и правое поддеревья, подсчитать число узлов в, левом поддереве, затем в правом поддереве. Зная количество узлов в каждом поддереве достаточно сложить эти значения и прибавить к получившей сумме единицу (учесть, таким образом, корень). Результат и будет общим числом узлов в дереве.

```
DOMAINS
    treetype=tree (integer, treetype, treetype) ; empty
PREDICATES
    counter (treetype, integer)
CLAUSES
%число узлов в пустом дереве равно 0
    counter (empty, 0):- !.
%если дерево непусто, общее количество узлов дерева равно сумме узлов
%левого и правого поддеревьев, увеличенной на 1
    counter (tree (Root, Left, Right), Count):- counter (Left, CountLeft), counter (Right, CountRight), Count=CountLeft+CountRight+1.
GOAL
    counter (tree (5, tree (2, tree (–1, empty, empty), tree (4, empty, empty)), tree (9, tree(7, empty, empty), empty)), N), write("N=", N).
```

Результат работы программы:

```
N=6
```

Для того чтобы лучше понять, каким образом работает приведенный пример, рекомендуется самостоятельно построить дерево целей, в качестве основы можно взять деревья целей для примеров работы со списками.

И еще один пример по работе с деревьями – создание упорядоченного дерева. Пусть в узлах дерева содержатся символы, символы, добавляемые к дереву, не

должны повторяться (программа не выполняет проверки на существование вводимых символов в дереве). Символы вводятся с клавиатуры, символ ‘#’ – признак конца ввода символов.

```
DOMAINS
    treetype=tree (char, treetype, treetype) ; end
PREDICATES
    insert (char, treetype, treetype)
    create_tree (treetype, treetype)
CLAUSES
%предикат insert обеспечивает вставку введенного с клавиатуры символа в дерево,
%для предиката insert записывается три правила вывода, в которых рассматривается
%три случая: первый случай – вставка нового символа в пустое дерево,
%второй и третий случаи – вставка нового символа в непустое дерево,
%если новый символ меньше, чем символ в корне дерева, то новый символ добавляется
%в левое поддерево, если больше – в правое
%(для сравнения символов используются их ASCII-коды)

%вставка нового символа в пустое дерево, получается дерево с корнем и пустыми
%левым и правым поддеревьями
    insert (New, end, tree (New, end, end)):- !.
%вставка нового символа по результатам проверки в левое поддерево,
%левое поддерево изменяется, а правое остается без изменений
    insert (New, tree (Root, Left, Right), tree (Root, NewLeft, Right)):- New<Root, !,
insert (New, Left, NewLeft).
%вставка нового символа в правое поддерево, без проверки,
%правое поддерево изменяется, а левое остается без изменений
    insert (New, tree (Root, Left, Right), tree (Root, Left, NewRight)):-
insert (New, Right, NewRight).
%предикат create_tree обеспечивает ввод символа с клавиатуры, и вызов предиката insert
    create_tree (Tree, NewTree):- readchar(Ch), Ch<>'#', !,
insert (Ch, Tree, TmpTree), create_tree (TmpTree, NewTree).
    create_tree (Tree, Tree).
GOAL
    create_tree (end, Tree), write("Tree=", Tree).
```

Результат работы программы:

```
    Tree=tree('d',tree('a',end,tree('b',end,tree('c',end,end))),tree('f',tree('e',end,end),tree('g',end,end)))
```

С клавиатуры была введена последовательность:

‘d’ ‘a’ ‘f’ ‘b’ ‘e’ ‘g’ ‘c’ ‘#’

2. Работа со строками в Visual Prolog (некоторые стандартные предикаты).

PDC Prolog поддерживает различные стандартные предикаты для обработки строк. Основными предикатами для работы со строками можно назвать предикат frontchar (String, Char, StringRest), позволяющий разделить строку String на первый символ Char и остаток строки StringRest и предикат fronttoken (String, Lexeme, StringRest), который работает аналогично предикату frontchar, но только отделяет от строки String лексему Lexeme. Лексемой называется последовательность символов, удовлетворяющая следующим условиям: имя в соответствии с синтаксисом Prolog’a, число или отличный от пробела символ.

Пример: преобразование строки в список символов (Два варианта. Варианты отличаются друг от друга граничным условием рекурсии. В первом варианте остановка рекурсии происходит, когда от строки будет отделен последний символ и строка станет пустой строкой. Во втором варианте остановка рекурсии происходит в момент попытки отделения от пустой строки очередного символа, что сделать не удается, и происходит откат ко второму предложению).

1 вариант

```
DOMAINS
    charlist=char*
PREDICATES
    string2charlist (string, charlist)
CLAUSES
```

```
    string2charlist (“”, [ ]):- !.
    string2charlist (Str, [H|T]):- frontchar (Str, H, Str_Rest), string2charlist (Str_Rest, T).
GOAL
    string2charlist (“abcde”, List), write (“List=”, List).
```

2 вариант

```
DOMAINS
    charlist=char*
PREDICATES
    string2charlist (string, charlist)
CLAUSES
    string2charlist (Str, [H|T]):- frontchar (Str, H, Str_Rest), !, string2charlist (Str_Rest, T).
    string2charlist (_, [ ]).
GOAL
    string2charlist (“abcde”, List), write (“List=”, List).
```

Результат работы программы:

```
    List=[‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
```

9. Методические материалы для обучающихся по подготовке к лабораторным занятиям

Тема. СОЗДАНИЕ ПРОСТЕЙШИХ ПРОЕКТОВ В СРЕДЕ VISUAL PROLOG

План

1. *Среда Visual Prolog: основные понятия, интерфейс.*
2. *Запуск и протестирование программ в среде Visual Prolog.*
3. *Понятие «nondeterm» недетерминированного предиката.*
4. *Выполнение заданий для самостоятельной работы.*

Среда Visual Prolog: основные понятия, интерфейс.

Prolog является языком, основанным на программировании логики (PROgramming in LOGic). Вместо детальных инструкций, предписывающих как решать ту или иную задачу, программист на языке Prolog уделяет основное внимание описанию задачи.

В среде Visual Prolog используется подход, получивший название «визуальное программирование», при котором внешний вид и поведение программ определяются с помощью специальных графических средств проектирования без традиционного программирования на алгоритмическом языке.

Visual Prolog автоматизирует построение сложных процедур и освобождает программиста от выполнения тривиальных операций. С помощью Visual Prolog проектирование пользовательского интерфейса и связанных- с ним окон, диалогов, меню, линии уведомлений о состояниях и т.д. производится в графической среде. С созданными объектами сразу же могут работать различные Кодовые Эксперты (Code Experts), которые используются для генерации базового и расширенного кодов на языке Prolog, необходимых для обеспечения их функционирования.

В Visual Prolog входят интерактивная среда визуальной разработки (VDE — Visual Develop Environment), которая включает текстовый и различные графические редакторы, инструментальные средства генерации кода, конструирующие управляющую логику (Experts), а также являющийся расширением языка интерфейс визуального программирования (VPI — Visual Programming Interface), Пролог-компилятор, набор различных подключаемых файлов и библиотек, редактор связей, файлы, содержащие примеры и помощь.

Visual Prolog поддерживается различными ОС, в том числе MS-DOS, всеми версиями Windows, а также некоторыми другими системами, требующими графического пользовательского интерфейса.

Запустите Visual Prolog. Для этого нажмите кнопку **Пуск**, выберите в меню пункт **Программы**, далее пункт **Visual Prolog 5.2 Personal Edition** и в появившемся подменю – пункт **Visual Prolog.**

Интерфейс Visual Prolog включает: главное меню, панель инструментов, окно проекта. Если во время последнего использования системы Visual Prolog там был открытый проект, то система автоматически вновь откроет этот проект.

На рис.1 изображен внешний вид среды Visual Prolog после запуска. В окне проекта отображаются модули открытого проекта route.prj: karta.pro, route.pro, VPITools.pro.

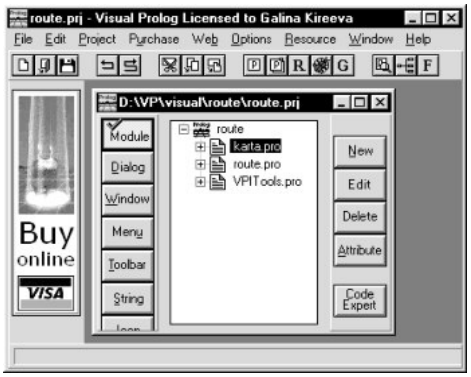


Рис 1. Среда разработки Visual Prolog

Левая панель кнопок в окне проекта позволяет выбирать нужный компонент проекта: модуль, окно, меню и т.д. С помощью кнопок правой панели выбранный компонент можно редактировать(кнопка Edit), удалять(кнопка Delete), а также добавлять новый(кнопка New).

Пункт меню **File** содержит команды для работы с файлами. Чтобы создавать новое окно редактирования, можно использовать команду File | New. Эта команда создаст новое окно редактора с заголовком "NONAME".

В меню **Edit** представлены команды, позволяющие редактировать текст программы. Встроенный редактор системы по интерфейсу похож на обычный текстовый редактор. Можно производить вырезку, копирование и вставку текста, операции Отмена/Восстановление, которые можно активизировать из меню Edit. Также меню Edit показывает "горячие клавиши", связанные для этих действий.

Пункт меню **Project** содержит команды для работы с проектом: создать новый, открыть, запустить и т.д. Запуск проекта на исполнение выполняется нажатием кнопки <R> на панели инструментов (или F9, или с помощью команд меню Project | Run).

Команды меню **Options** позволяют выполнять настройку проекта, устанавливать необходимые параметры.

Программа на ПРОЛОГе состоит из предложений, которые могут быть фактами, правилами или запросами. Как правило, программа состоит из четырех разделов.

DOMAINS – секция описания доменов (типов). Секция применяется, если в программе используются нестандартные домены.

PREDICATES – секция описания предикатов. Секция применяется, если в программе используются нестандартные предикаты.

CLAUSES – секция предложений. Именно в этой секции записываются предложения: факты и правила вывода.

GOAL – секция цели. В этой секции записывается запрос.

Среда Visual Prolog позволяет протестировать программу без создания проекта. Для этого используется утилита Test Goal. Достаточно создать новый файл, набрать текст программы и активизировать Test Goal нажатием кнопки <G> на панели инструментов. Автономно исполняемый файл при этом не создается. Утилита Test Goal компилирует только тот код, который определен в активном окне редактора (код в других открытых окнах или модулях проектов, если они есть, игнорируются). Test Goal находит *все* возможные решения задачи и автоматически выводит значения *всех* переменных.

Пример 1.

Имеется база данных, содержащая следующие факты:

родитель(Илья, Марина).
родитель(Марина, Ира).
родитель(Елена, Иван).
родитель(Николай, Ира).
родитель(Ольга, Алексей).
родитель(Марина, Саша).
родитель(Сергей, Иван).

Определить:

1) верно ли, что Марина является родителем Саши;
2) верно ли, что Алексей является родителем Ольги;
3) кто является ребенком Николая;
4) кто родители Ивана;

5) всех родителей и их детей.

Решение.

1. Запустите среду Visual Prolog. Закройте окно проекта (если оно открыто) и откройте новый файл (**File**|**New**) (рис.2)

В появившемся окне наберите текст программы, содержащий разделы: PREDICATES (описание предиката ***родитель***), CLAUSES (перечисляются имеющиеся факты) и GOAL (запрос).

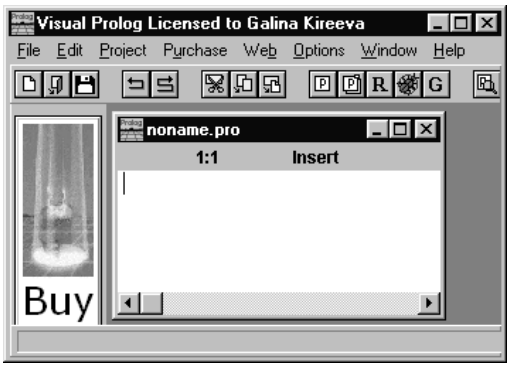


рис.2. Рабочее окно редактора

```
DOMAINS
имя=string
PREDICATES
nondeterm      родитель(имя, имя)
CLAUSES
родитель(илья, марина).
родитель(марина, ира).
родитель(елена, иван).
родитель(николай, ира).
родитель(ольга, алексей).
родитель(марина, саша).
родитель(сергей, иван).
GOAL
родитель(марина, саша) .
```

Запустите и протестируйте программу с помощью команды **Project | Test Goal** (можно использовать кнопку на панели инструментов **<G>** или сочетание клавиш **<Ctrl>+<G>**). Результат выполнения программы будет выведен в отдельном окне

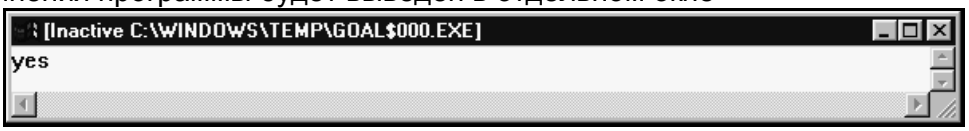


рис3. Окно вывода результата

Указание: перед следующим запуском программы следует закрыть это окно.

2. Для ответа на вопрос: верно ли, что Алексей является родителем Ольги, измените запрос:

```
GOAL
 родитель(алексей, ольга).
```

После запуска программы (Project | Test Goal) будет получен ответ:

no

3. Для ответа на вопрос: кто является ребенком Николая, запишите цель:

```
GOAL
 родитель(николай, X) .
```

Результат:

```
X=ира
1 Solution
```

4. Для ответа на вопрос: кто родители Ивана, укажите запрос:

```
GOAL
 родитель(X, иван), родитель(Y, иван), X<>Y.
```

Результат:

X=елена, Y=сергей
X=сергей, Y=елена
2 Solutions
5. Для определения всех родителей и их детей, запишите:
GOAL
родитель(X, Y).
Результат:
X=илья, Y=марина
X=марина, Y=ира
X=елена, Y=иван
X=николай, Y=ира
X=ольга, Y=алексей
X=марина, Y=саша
X=сергей, Y=иван
7 Solutions
Пример 2
Имеются факты вида: *родитель(имя, имя)* и *женщина(имя).*
а) составить правило ***мать*** и определить, кто мать Маши.
Решение:
```
DOMAINS
имя=string
PREDICATES
родитель(имя, имя)
женщина(имя)
мать(имя,имя)
CLAUSES
родитель("Марина","Ирина").
родитель("Елена", "Анна").
родитель("Ольга","Марина").
родитель("Ольга","Татьяна").
родитель("Татьяна","Катя").
родитель("Анна", "Маша").
женщина("Ольга").
женщина("Маша").
женщина("Ирина").
женщина("Елена").
женщина("Анна").
женщина("Марина").
женщина("Татьяна ").
женщина("Катя").
мать(X,Y):-родитель(X,Y),женщина(X).
GOAL
мать(X,"Маша").
```
Результат:
X=Анна
1 Solution
b) составить правило ***бабушка*** и определить, кто бабушка Маши.
Решение:
```
DOMAINS
имя=string
PREDICATES
nondeterm родитель(имя,имя)
женщина(имя)
nondeterm мать(имя,имя)
nondeterm бабушка(имя,имя)
CLAUSES
родитель("Марина","Ирина").
родитель ("Елена", "Анна").
```

```
родитель("Ольга","Марина").
родитель("Ольга","Татьяна").
родитель("Татьяна","Катя").
родитель ("Анна", "Маша").
женщина("Ольга").
женщина( "Маша").
женщина("Ирина").
женщина("Елена").
женщина("Анна").
женщина("Марина").
женщина("Татьяна ").
женщина("Катя").
мать(X,Y):-родитель(X,Y),женщина(X).
бабушка(X,Z):-мать(X,Y),родитель(Y,Z).
GOAL
бабушка(X,"Маша").
```

Результат:

```
X=Елена
1 Solution
```

Замечание: ключевое слово *nondeterm* определяет недетерминированные предикаты, которые могут совершать откат назад и генерировать множественные решения. Таким образом, если задача предполагает возможность получения несколько решений, следует объявлять предикаты как недетерминированные.

с) составить правило ***внучка*** и определить, сколько внучек у Ольги и как их зовут.

Решение:

```
DOMAINS
имя=string
PREDICATES
nondeterm родитель(имя,имя)
женщина(имя)
nondeterm мать(имя,имя)
nondeterm бабушка(имя,имя)
nondeterm внучка(имя,имя)
CLAUSES
    родитель("Марина","Ирина").
    родитель ("Елена", "Анна").
 родитель("Ольга","Марина").
 родитель("Ольга","Татьяна").
 родитель("Татьяна","Катя").
    родитель("Анна", "Маша").
    женщина("Ольга").
 женщина( "Маша").
    женщина("Ирина").
 женщина("Елена").
 женщина("Анна").
 женщина("Марина").
 женщина("Татьяна ").
 женщина("Катя").
 мать(X,Y):-родитель(X,Y),женщина(X).
    бабушка(X,Z):-мать(X,Y),родитель(Y,Z).
    внучка(X,Y):-бабушка(Y,X),женщина(X).
GOAL
внучка(X, "Ольга").
```

Результат:

```
X=Ирина
X=Катя
2 Solutions
```

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

Имеется база данных, содержащая следующие факты:

играет (“Саша”, футбол).
играет (“Катя”, теннис).
играет (“Саша”, теннис).
играет (“Андрей”, футбол).
играет (“Олег”, футбол).
играет (“Ольга”, теннис).
играет (“Катя”, волейбол).
играет (“Олег”, волейбол).
женщина(“Катя”).
женщина(“Ольга”).
мужчина(“Саша”).
мужчина(“Андрей”).
мужчина(“Олег”).

a) используя имеющиеся факты, составить новое правило ***волейбол_жен(X)*** и определить всех женщин, играющих в волейбол;

b) используя имеющиеся факты, составить новое правило ***футбол_муж(X)*** и определить всех мужчин, играющих в футбол;

c) используя имеющиеся факты, составить правило ***теннис_пара(X,Y),*** позволяющее найти смешанную теннисную пару (мужчина+женщина). Определить все такие пары.

СОЗДАНИЕ ПРОСТЕЙШИХ ПРОЕКТОВ

Создание проекта позволяет протестировать пример как автономную исполняемую программу. После запуска проекта на исполнение создается exe-файл, работа которого завершается после *первого* решения, удовлетворяющего решению задачи. Запуск программы в этом режиме не обеспечивает автоматический вывод значений переменных, поэтому необходимо использовать стандартный предикат вывода **write**.

Пример.

Заданы отношения-факты:

родитель(“Иван”,”Катя”).
родитель(“Анна”,” Олег ”).
родитель(“Олег”,”Дима”).
родитель(“Игорь”,”Ольга”).
родитель(“Олег”,”Виктор”).
родитель(“Игорь”,”Иван”).
мужчина(“Дима”).
мужчина(“Иван”).
мужчина(“Игорь”).
мужчина(“Олег”).
мужчина(“Виктор”).
женщина(“Катя”).
женщина(“Ольга”).
женщина(“Анна”).

Составить новое отношение-правило ***дед(X,Y)*** и определить, кто является дедушкой Кати. Создать проект и протестировать пример как автономную исполняемую программу.

Решение

1. Запустите среду Visual Prolog и создайте новый проект (Project | New Project), активизируется окно **Application Expert** (эксперт приложения).

2. Определите имя проекта (Primer) и базовый каталог, куда будет сохранен проект (например, D:\VP\ Primer)

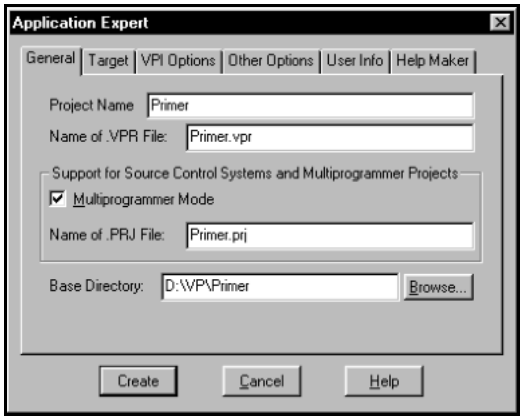


рис.4. Окно **Application Expert**

На вкладке **Target** установите в поле UI Strategy параметр Easywin и нажмите кнопку **Create** для создания проекта (рис.5):

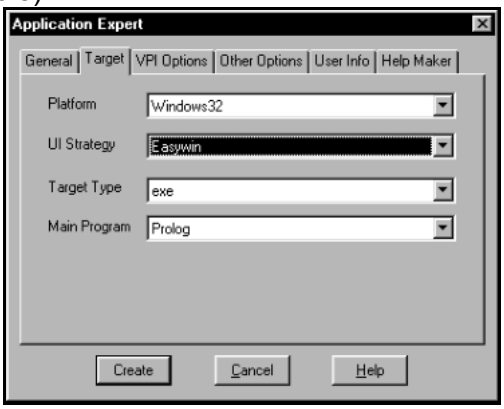


рис.5. Установки на вкладке **Target** окна **Application Expert**

3. Откройте окно **Compiler Options** (Options | Project | Compiler Options), откройте вкладку Warnings и установите опции компилятора для созданного проекта как указано на рисунке (рис.6):

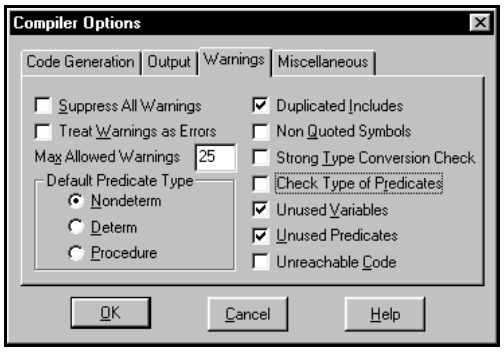


рис.6. Установки опций компилятора

Нажмите OK.

4. В окне проекта выделите файл Primer.pro и откройте его для редактирования (двойной щелчок или кнопка **Edit**)

Файл с расширением .pro содержит секции PREDICATES, GOAL, CLAUSES. Допишите необходимые определения так, чтобы получилась программа:

```
DOMAINS
имя=string
PREDICATES
родитель(имя,имя)
женщина(имя)
мужчина(имя)
дед(имя, имя)
CLAUSES
родитель("Иван","Катя").
родитель("Анна","Олег").
родитель("Олег","Дима").
```

родитель("Игорь","Ольга").
родитель("Олег","Виктор").
родитель("Игорь","Иван").
мужчина("Дима").
мужчина("Иван").
мужчина("Игорь").
мужчина("Олег").
мужчина("Виктор").
женщина("Катя").
женщина("Ольга").
женщина("Анна").
дед(X,Z):-родитель(X,Y), родитель(Y,Z),
мужчина(X).
GOAL
дед(X,"Катя"),write(X).

5. Откомпилируйте исходный код примера и запустите его как автономную исполняемую программу. (Project | Run, или клавиша <F9>, или кнопка <R>). Результат выполнения программы должен отобразиться в окне:

рис.7. Окно вывода результата

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

Доработайте исходный код примера следующим образом:
1) добавьте новое правило ***бабушка*** и определите, кто является бабушкой;
2) добавьте новое правило ***внук*** и определите, кто внук Анны;
3) добавьте новое правило ***брат*** и определите, кто брат Димы;
4) добавьте новое правило ***сестра*** и определите, кто сестра Ивана.

Тема. ПОИСК С ВОЗВРАТОМ

План

1. Поиск с возвратом как один из основных приемов поиска решений поставленной задачи в Прологе
2. Построение целевого дерева поиска с возвратом.
3. Управление поиском с возвратом: предикаты fail и отсечения.
4. Выполнение заданий для самостоятельной работы.

Поиск с возвратом (backtracking) – это один из основных приемов поиска решений поставленной задачи в ПРОЛОГе. Выполняя поиск, ПРОЛОГ может столкнуться с необходимостью выбора между альтернативными путями. Тогда он ставит маркер у места развилки (точка отката) и выбирает первую подцель. Если она не выполняется, то ПРОЛОГ возвращается в точку отката и переходит к следующей подцели.

Среда Visual Prolog позволяет использовать отладчик для пошагового выполнения программы. Отладчик работает с откомпилированным кодом. В исходном коде можно ставить точки останова и выполнять программу по шагам. В режиме пошагового выполнения программы можно просматривать значения переменных и содержимое утвержденных фактов.

Пример

Имеется база данных, содержащая факты вида ***отдыхает(имя, город)***, ***украина(город***), ***россия(город), прибалтика(город).*** Составить правило, позволяющее определить, кто отдыхал в России.

Проследить поиск решения задачи с помощью отладчика Visual Prolog и построить целевое дерево поиска с возвратом.

Решение:

1. Создайте новый проект (Project | New Project) и наберите текст программы:
DOMAINS
имя, город=string

```
PREDICATES
отдыхает(имя, город)
украина(город)
россия(город)
прибалтика(город)
отдых_Россия(имя)
CLAUSES
отдыхает(sasha, antalia).
отдыхает(anna, sochi).
отдыхает(dima, urmala).
отдыхает(oleg, kiev).
украина(kiev).
россия(sochi).
прибалтика(urmala).
отдых_Россия(X):- отдыхает(X,Y),россия(Y).
GOAL
отдых_Россия(X),
write(X),nl.
```

3. Сохраните проект (**Project** | **Save Project**)

4. Запустите его на исполнение (**Project** | **Run**, или клавиша **<F9>,** или кнопка **<R>**).

Результат выполнения программы:

anna

5. Проследите поиск этого решения с помощью отладчика(Debugger). Для этого:

а) запустите отладчик (**Project** | **Debug**);

б) в окне отладчика выберите команду **View** | **Local Variables** (для просмотра текущих значений переменных);

в) нажимайте клавишу **<F7>** (или **Run** | **Trace Into**) для пошагового выполнения программы, текущие значения переменных отображаются в окне Variables For Current Clause

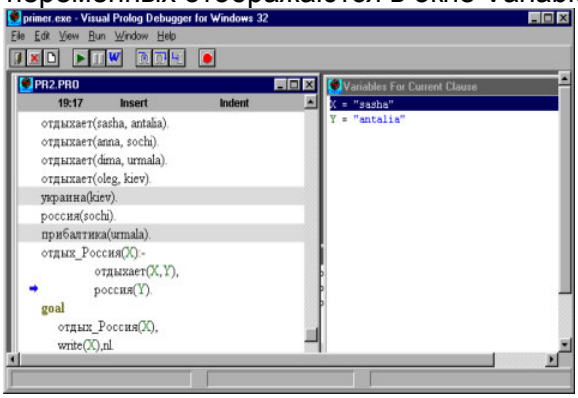


рис.9. Окно отладчика

Поиск решения можно представить следующим образом:

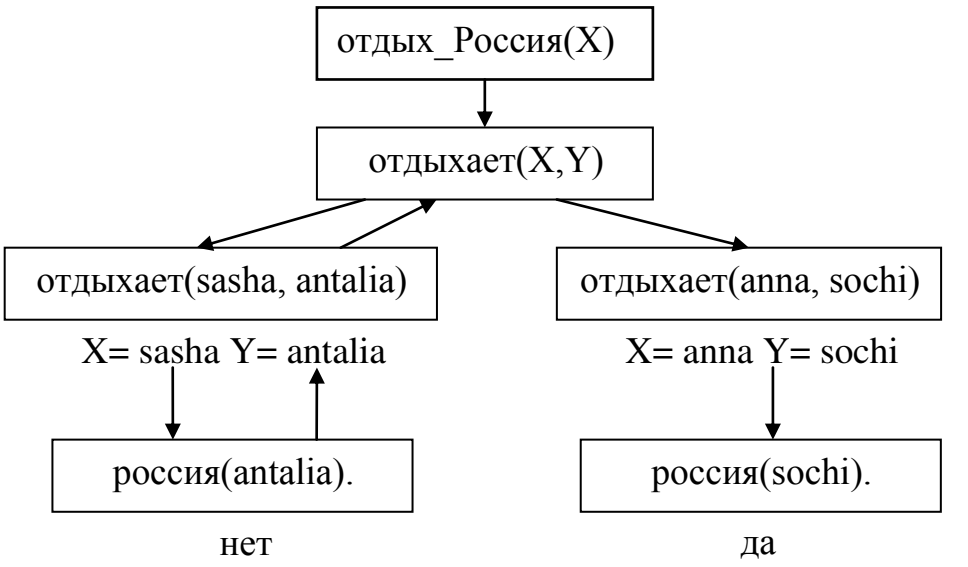


Рис.10. Целевое дерево поиска решения

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. База данных содержит следующие факты:
увлекается(“Коля”, гитара).
увлекается(“Оля”, скрипка).
увлекается(“Дима”, плаванье).
увлекается(“Таня”, теннис).
спорт(плаванье).
спорт(теннис).
муз_инстр(скрипка).
муз_инстр(гитара).
а) составить правило спортсмен и определить, кто увлекается спортом;
б) проследить за поиском решения с помощью отладчика;
в) построить целевое дерево поиска с возвратом.

Управление поиском с возвратом: предикаты fail и отсечения.

fail – это тождественно-ложный предикат, искусственно создающий ситуацию неуспеха. После выполнения этого предиката управление передается в точку отката и поиск продолжается. Использование предиката fail позволяет найти все решения задачи.

Чтобы ограничить пространство поиска и прервать поиск решений при выполнении какого-либо условия, используется предикат *отсечения* (обозначается **!**), Однажды пройдя через отсечение, невозможно вернуться назад, т.к. этот предикат является тождественно-истинным. Процесс может только перейти к следующей подцели, если такая имеется.

Например, p :- p1, p2, !, p3.

Если достигнуты цели p1 и p2, то возврат к ним для поиска новых решений невозможен.

Пример 1

База данных содержит факты вида: ***student(имя, курс).*** Создать проект, позволяющий сформировать список студентов 1-го курса.

Решение:

```
PREDICATES
student(symbol,integer)
spisok
CLAUSES
student(vova,3).
student(lena,1).
student(dima,1).
student(ira,2).
student(marina,1).
spisok:-student(X,1),write(X),nl,fail.
```

```
GOAL
write("Список студентов 1-курса"),nl,spisok.
```

Результат выполнения программы:

```
Список студентов 1-курса
lena
dima
marina
```

Пример 2

База данных содержит факты вида **father(name, name).** Создать проект, позволяющий определить кто чей отец.

Решение:

```
DOMAINS
name=symbol
PREDICATES
father (name, name)
everybody
CLAUSES
father ("Павел", "Петр").
father ("Петр", "Михаил").
father ("Петр", "Иван").
everybody:- father (X, Y),
write(X," - это отец",Y,"a"),nl,
fail.
GOAL
everybody.
```

Результат выполнения программы:

```
Павел - это отец Петра
Петр - это отец Михаила
Петр - это отец Ивана
```

Пример 3

Создать проект, реализующий железнодорожный справочник. В справочнике содержится следующая информация о каждом поезде: номер поезда, пункт назначения и время отправления.

а) вывести всю информацию из справочника.

Решение:

```
DOMAINS
nom=integer
p, t=string
PREDICATES
poezd(nom,p,t)
CLAUSES
poezd(233,moskva,"12-30").
poezd(257,moskva,"22-40").
poezd(133,armavir,"10-20").
poezd(353,armavir,"20-40").
poezd(353,adler,"02-30").
poezd(413,adler,"11-10").
poezd(256,piter,"21-30").
GOAL
write(" Расписание поездов"), nl,
write("Номер Пункт прибытия Время отправления"),
nl, poezd(N,P,T), write(N," ",P," ",T),nl,fail.
```

Результат выполнения программы

```
Расписание поездов
Номер Пункт прибытия Время отправления
233 moskva 12-30
257 moskva 22-40
133 armavir 10-20
```

353 armavir 20-40
353 adler 02-30
413 adler 11-10
256 piter 21-30

б) организовать поиск поезда по пункту назначения.

Решение:

```
GOAL
write (" Пункт назначения:"), Readln(P), nl,
write ("Номер Время отправления"), nl,
poezd(N,P,T), write(N," ",T), nl, fail.
```

Комментарий: Readln –стандартный предикат ввода строкового значения

Результат выполнения программы

Пункт назначения:armavir
Номер Время отправления
133 10-20
353 20-40

в) вывести информацию о поездах, отправляющихся в заданный временной промежуток

Решение:

```
GOAL
write(" Время отправления:"),nl,
write("с..."), Readln(T1),
write("до..."), Readln(T2), nl,
write("Номер Пункт назначения Время отправления"),
nl,poezd(N,P,T),T>=T1,T<=T2,write(N," ",P," ", T),
nl, fail.
```

Результат выполнения программы

Время отправления:
с...10-00
до...15-00

Номер Пункт назначения Время отправления
233 moskva 12-30
133 armavir 10-20
413 adler 11-10

Пример 4

Имеется база данных, содержащая данные о спортсменах: имя и вид спорта. Определить возможные пары одного из спортсменов-теннисистов с другими теннисистами.

Решение:

```
DOMAINS
имя,вид_сп=string
PREDICATES
играет(имя,вид_сп)
спис_спортс
CLAUSES
играет("Саша",теннис).
играет("Аня",волейбол).
играет("Олег",футбол).
играет("Коля",теннис).
играет("Саша",футбол).
играет("Андрей",теннис).
спис_спортс:- играет(X,теннис),!,играет(Y,теннис),
X<>Y,write(X,"-",Y),nl,fail.
GOAL
write("Пары теннисистов"),nl,
спис_спортс.
```

Результат выполнения программы:

Пары теннисистов
Саша-Коля

Саша-Андрей

Пример 5

Студенту в зависимости от набранной в процессе обучения суммы баллов **Z** присваивается квалификация:

магистр (**M**), если 80<=Z<=100

специалист (**S**), если 60<= Z< 80

бакалавр (**B**), если 40<= Z< 60

неудачник (**N**), если 0<=Z< 40

Составить программу, которая определит квалификацию в зависимости от введенного значения **Z**

Решение:

Для решения задачи составим правило ***grade***, устанавливающее связь между количеством баллов (z) и квалификацией (r). Правило состоит из нескольких частей. Первые две части обеспечивают проверку недопустимых значений Z с выводом соответствующего сообщения. Остальные части правила определяют квалификацию в зависимости от значения Z.

```
DOMAINS
z=integer
r=string
PREDICATES
grade(z,r)
CLAUSES
grade(Z,""):-Z<0,!, write("Неверный ввод данных!").
grade(Z,""):-Z>100,!,write("Неверный ввод данных!").
grade(Z,"M"):-Z>=80,!.
grade(Z,"S"):-Z>=60,!.
grade(Z,"B"):-Z>=40,!.
grade(Z,"N").
GOAL
write("Z="), readint(Z), grade(Z,R),write(R).
```

Комментарий: readint – стандартный предикат ввода целочисленного значения

Результат выполнения программы:

```
1-й случай:
Z=88
M
2-й случай:
Z=65
S
3-й случай:
Z=39
N
4-й случай:
Z=110
Неверный ввод данных!
```

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. База данных содержит факты вида: **отдыхает(имя, город), украина(город), россия(город), женщина (имя), мужчина(имя).**

а) вывести список женщин, отдыхающих в России;

б) вывести список мужчин, отдыхающих на Украине.

2. База данных содержит факты вида: **книга(автор, название, издательство, год_издания), украина(город).**

а) вывести весь список книг;

б) вывести список книг авторов Пушкина и Чехова;

в) вывести список книг, изданных в издательстве «Питер» не ранее 2000 года.

3. Составить программу, реализующую авиасправочник. В справочнике содержится следующая информация о каждом рейсе: номер рейса, пункт назначения, время вылета, дни(ежедн., чет, нечет). Вывести:

а) всю информацию из справочника;

б) информацию о самолетах, вылетающих в заданный пункт по четным дням;

в) информацию о самолетах, вылетающих ежедневно не позже указанного времени.

4. Составить программу, реализующую географический справочник. В справочнике содержится следующая информация о каждой стране: название страны, название столицы, численность населения, географическое положение (Европа или Азия). Вывести:

а) всю информацию из справочника;

б) информацию о странах, численность населения которых превышает заданное значение;

в) информацию о европейских странах, численность населения которых не превышает заданное значение.

5. Составить программу, реализующую словарь. В словаре содержится следующая информация: слово и его перевод (русские и английские слова). Реализовать вывод всего словаря, перевод с русского на английский, с английского на русский (с несколькими значениями).

6. Составить программу, реализующую телефонный справочник. В справочнике содержится следующая информация о каждом абоненте: имя и телефон. Реализовать вывод всей информации из справочника, поиск телефона по имени, поиск имени по телефону

7. База данных содержит факты вида: ***ученик(имя, класс)*** и ***увлекается(имя, хобби).*** Составить программу, которая выводит:

а) список всех учеников и их увлечения;

б) подбирает одному из учеников указанного класса, увлекающемуся кино, пару из других классов. Вывести все возможные пары.

8. База данных содержит факты вида: ***ученик(имя, класс)*** и ***играет(имя, вид_спорта).*** Составить программу, которая:

а) выводит список всех учеников заданного класса и вид спорта, которым они увлекаются;

б) подбирает одному из учеников указанного класса, играющему в бадминтон, пару из других классов. Вывести все возможные пары.

Тема. АРИФМЕТИЧЕСКИЕ ВЫЧИСЛЕНИЯ НА ПРОЛОГЕ

План

1. *Возможности вычислений на Прологе.*
2. *Встроенные функции для вычисления арифметических выражений на Прологе.*
3. *Выполнение заданий для самостоятельной работы.*

Хотя Пролог не предназначен для решения вычислительных задач, его возможности вычислений аналогичны соответствующим возможностям таких языков программирования как Basic, C, Pascal.

В языке Пролог имеется ряд встроенных функций для вычисления арифметических выражений, некоторые из которых перечислены в таблице 1.

Таблица 1. Математические операции и функции в Прологе

X + Y	Сумма X и Y
X – Y	Разность X и Y
X * Y	Произведение X и Y
X / Y	Деление X на Y
X mod Y	Остаток от деления X на Y
X div Y	Целочисленное деление X на Y
abs(X)	Абсолютная величина числа X
sqrt(X)	Квадратный корень из X
random(X)	Случайное число в диапазоне от 0 до 1
random(Int,X)	Случайное целое число в диапазоне от 0 до Int
round(X)	Округление X

trunc(X)	Целая часть X
sin(X)	Синус X
cos(X)	Косинус X
arctan(X)	Арктангенс X
tan(X)	Тангенс X
ln(X)	Натуральный логарифм X
log(X)	Логарифм X по основанию 10

Пример 1.

Вычислить значение выражения Z=(2*X+Y)/(X-Y) для введенных X и Y.

Решение:

```
PREDICATES
        знач_выраж(real,real)
CLAUSES
        знач_выраж(X,Y):-X<>Y, Z=(2*X+Y)/(X-Y),
        write("Z=",Z);
        X=Y, write ("Делить на 0 нельзя!").
GOAL
        Write("X="),readreal(X),
        Write("Y="),readreal(Y),знач_выраж(X,Y),nl.
```

Комментарий: readreal – предикат для ввода действительных чисел

Результат выполнения программы:

1-й случай:

```
        X=4
        Y=4
        Делить на 0 нельзя!
```

2-й случай:

```
        X=5
        Y=2
        Z=4
```

Пример 2.

Найти минимальное из двух введенных A и B.

Решение:

```
PREDICATES
        min(integer,integer,integer)
CLAUSES
        min(A,B,A):-A<=B,!.
        min(A,B,B).
GOAL
        Write("A="),readreal(A),Write("B="),readreal(B),
        min(A,B,Min),write("min=",Min),nl.
```

Результат выполнения программы:

1-й случай:

```
        A=5
        B=17
        min=5
```

2-й случай:

```
        A=35
        B=18
        min=18
```

3-й случай:

```
        A=8
        B=8
        min=8
```

Пример 3.

Определить, является четным или нечетным случайным образом выбранное число от 0 до 20.

Решение:

```
PREDICATES
        chet
CLAUSES
        chet:-random(20,X),write(X),X mod 2=0,
        write(" - четное"),!.
        chet:-write( " - нечетное").
GOAL
        chet.
```

Результат выполнения программы:

1-й случай:

6 – четное

2-й случай:

19 – нечетное

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Составить программу для вычисления значения выражения Y=(X2+1)/(X-2) для введенного X.
2. Составить программу для вычисления значения выражения S=2(X2+Y2)/(X+Y) для введенных X и Y.
3. Составить программу для вычисления значения выражения $z=e^{x}\sin x +3\ln x$ для введенного X.
4. Составить программу для вычисления значения выражения $y=\ln(\lg(\sin x+e^{x}))$для введенного X.
5. Составить программу для вычисления среднего арифметического двух введенных чисел.
6. Составить программу для вычисления среднего геометрического двух введенных чисел.
7. Составить программу для проверки введенного натурального числа на четность.
8. Составить программу для проверки попадает ли введенное число X в заданный промежуток [a,b].
9. Составить программу для выбора наименьшего из трех введенных чисел.
10. Составить программу для выбора наибольшего из трех введенных чисел.

Тема. РЕКУРСИЯ

План

1. *Организации рекурсии на Прологе.*
2. *Составление рекурсивных правил.*
3. *Выполнение заданий для самостоятельной работы.*

Рекурсивная процедура – это процедура, вызывающая сама себя до тех пор, пока не будет соблюдено некоторое условие, которое остановит рекурсию. Такое условие называют *граничным*. Рекурсивное правило всегда состоит по крайней мере из двух частей, одна из которых является *нерекурсивной*. Она и определяет граничное условие.

В рекурсивной процедуре нет проблемы запоминания результатов ее выполнения, потому что любые вычисленные значения можно передавать из одного вызова в другой как аргументы рекурсивно вызываемого предиката. Рекурсия является эффективным способом для решения задач, содержащих в себе подзадачу такого же типа.

Пример 1.

База данных содержит следующие факты:

```
roditel(ivan,oleg).
roditel(inna,oleg).
roditel(oleg,dima).
roditel(oleg,marina).
```

Составить рекурсивное правило ***предок*** и определить всех предков и их потомков.

Решение:

```
DOMAINS
```

```
        name=string
PREDICATES
        roditel(name,name)
        predok(name,name)
CLAUSES
        roditel(ivan,oleg).
        roditel(inna,oleg).
        roditel(oleg,dima).
        roditel(oleg,marina).
        predok(X,Z):-roditel(X,Z).    % нерекурсивная часть правила
        predok(X,Z):-roditel(X,Y),    % рекурсивная часть правила
                predok(Y,Z).
GOAL
        predok(X,Y),
        write("Predok -",X," Ego potomok-",Y),nl,fail.
```

Результат выполнения программы:

```
        Predok -ivan   Ego potomok-oleg
        Predok -inna   Ego potomok-oleg
        Predok -oleg   Ego potomok-dima
        Predok -oleg   Ego potomok-marina
        Predok -ivan   Ego potomok-dima
        Predok -ivan   Ego potomok-marina
        Predok -inna   Ego potomok-dima
        Predok -inna   Ego potomok-marina
```

Пример 2. Вычисление факториала.

Решение:

```
PREDICATES
        fact(integer,integer)
CLAUSES
        fact(0,1):-!.              % Факториал нуля равен единице
        fact(N,F):- N1=N-1,        %  уменьшаем N на единицу,
                fact(N1,F1),       % вычисляем факториал нового числа,
                F=N*F1.            %  а затем умножает его на N
GOAL
        write("N="),readint(N),fact(N,F),write("F=",F),nl.
```

Результат выполнения программы:

```
1-й случай:
        N=0
        F=1
2-й случай:
        N=1
        F=1
3-й случай:
        N=4
        F=24
```

Пример 3

Составить программу для вычисления $Y=X^n$, X, n – целые числа

Решение:

Составим правило **stepen**, состоящее из 3-х частей.

1-я часть правила (нерекурсивная) определяет, что $X^0=1$.

2-я часть правила (рекурсивная) вычисляет X^n для положительного n.

3-я часть (рекурсивная) - вычисляет X^n для отрицательного n (добавляется необходимое условие X<>0)

```
PREDICATES
        stepen(real,real,real)
CLAUSES
        stepen(X,0,1):-!.
        stepen(X,N,Y):-N>0,N1=N-1,stepen(X,N1,Y1),Y=Y1*X,!.
```

```
        stepen(X,N,Y):-X<>0,K=-N,stepen(X,K,Z),Y=1/Z.
GOAL
        write("X="),readreal(X),
        write("N="),readreal(N),
        stepen(X,N,Y),write("Y=",Y),nl.
```

Результат выполнения программы:

1-й случай:

```
        X=3
        N=2
        Y=9
```

2-й случай:

```
        X=2
        N=-2
        Y=0.25
```

Пример 4 . Ханойские башни

Имеется три стержня: A, B и C. На стержне A надеты N дисков разного диаметра, надетые друг на друга в порядке убывания диаметров. Необходимо переместить диски со стержня A на стержень C используя B как вспомогательный, если перекладывать можно только по одному диску и нельзя больший диск класть на меньший.

Решение:

Составим правило **move**, определяющее порядок переноса дисков.

1-я (нерекурсивная) часть правила определяет действие, если на стержне находится 1 диск.

2-я (рекурсивная) часть правила перемещает сначала верхние N-1 диск на стержень B, используя C как вспомогательный, затем оставшийся диск на стержень C и, наконец, диски со стержня B на C, используя A как вспомогательный.

```
PREDICATES
        move(integer,char,char,char)
CLAUSES
        move(1,A,B,C):-
                write("Перенести диск с ",A," на  ",C),nl,!.
        move(N,A,B,C):-
                M=N-1,move(M,A,C,B),
                write("Перенести диск с ",A," на  ",C),nl,
                move(M,B,A,C).
GOAL
        write("Ханойские башни"), nl,
        write("Количество дисков:"), readint(N),nl,
        move(N,'A','B','C').
```

Результат выполнения программы:

```
        Ханойские башни
        Количество дисков:3
        Перенести диск с A на  C
        Перенести диск с A на  B
        Перенести диск с C на  B
        Перенести диск с A на  C
        Перенести диск с B на  A
        Перенести диск с B на  C
        Перенести диск с A на  C
```

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Вычислить сумму 1+2+3+…+N.
2. Подсчитать сумму ряда целых четных чисел от 2 до N.
3. Вычислить сумму ряда целых нечетных чисел от 1 до n.
4. Найти значение произведения: 2*4*6*...*26
5. Найти значение произведения: 1*3*5*...*11

6. Вычислить значение n-го члена ряда Фибоначчи: f(0)=0, f(1)=1, f(n)=f(n-1)+f(n-2).
7. Используя базу данных и правило **предок** из примера 2 составить правило для определения всех потомков-мужчин.
8. Используя базу данных и правило **предок** из примера 2 составить правило для определения всех потомков-женщин.

Тема. РЕШЕНИЕ ЛОГИЧЕСКИХ ЗАДАЧ В ПРОЛОГЕ

План

1. Решение логических задач на Прологе с помощью правил, моделирующих процесс мышления человека.

2. Описание конечных множеств как баз данных. Описание зависимостей между множествами с одинаковым количеством элементов с помощью правил.

3. Выполнение заданий для самостоятельной работы.

ПРОЛОГ позволяет наиболее естественным образом решать логические задачи, моделируя процесс размышления человека с помощью правил.

Многие логические задачи связаны с рассмотрением нескольких конечных множеств с одинаковым количеством элементов, между которыми устанавливается взаимно-однозначное соответствие. В ПРОЛОГе эти множества можно описывать как базы данных, а зависимости между объектами устанавливать с помощью правил.

Пример 1

В автомобильных гонках три первых места заняли Алеша, Петя и Коля. Какое место занял каждый из них, если Петя занял не второе и не третье место, а Коля - не третье?

Решение

Традиционным способом задача решается заполнением таблицы. По условию задачи Петя занял не второе и не третье место, а Коля - не третье. Это позволяет поставить символ '-' в соответствующих клетках.

Имя	I место	II место	III место
Алеша			
Петя		-	-
Коля			-

Между множеством имен участников гонки и множеством мест должно быть установлено взаимнооднозначное соответствие. Поэтому определяем занятое место сначала у Пети, затем у Коли и, наконец, у Алеши. В соответствующих клетках проставляем знак '+'. В каждой строке и каждом столбце должен быть только один такой знак.

Имя	I место	II место	III место
Алеша	**-**	**-**	**+**
Петя	**+**	**-**	**-**
Коля	**-**	**+**	**-**

Из последней таблицы следует, что Алеша занял третье место, Петя - первое, Коля - второе.

Программа на ПРОЛОГе будет выглядеть следующим образом:

```
PREDICATES
        имя(string)
        место(string)
        соответствие(string,string)
        решение(string,string,string,string,string,string)
CLAUSES
        имя(алеша).
        имя(петя).
        имя(коля).
        место(первое).
        место(второе).
        место(третье).
        /* Устанавливаем взаимнооднозначное соответствие
        между множеством имен и множеством мест, X - имя,Y -  место */
        /* Петя занял не второе и не третье место */
        соответствие(X, Y):-имя(X), X=петя,
```

```
                место(Y),not(Y=второе),
                not(Y=третье).
    /* Коля занял не третье место */
    соответствие(X, Y):- имя(X), X=коля,
                место(Y), not(Y=третье).
    соответствие(X, Y):- имя(X),  X=алеша, место(Y).
    /* У всех ребят разные места */
    решение(X1,Y1,X2,Y2,X3,Y3):-
                X1=петя,соответствие(X1,Y1),
                X2=коля,соответствие(X2,Y2),
                X3=алеша,соответствие(X3,Y3),
                Y1<>Y2, Y2<>Y3, Y1<>Y3.
GOAL
    решение(X1,Y1,X2,Y2,X3,Y3), write(X1,"  -  ",Y1),nl,
    write(X2,"  -  ",Y2),nl,write(X3,"  -  ",Y3),nl.
```

Результат выполнения программы

```
петя  -  первое
коля  -  второе
алеша  - третье
```

Пример 2

Наташа, Валя и Аня вышли на прогулку, причем туфли и платье каждой были или белого, или синего, или зеленого цвета. У Наташи были зеленые туфли, а Валя не любит белый цвет. Только у Ани платье и туфли были одного цвета. Определить цвет туфель и платья каждой из девочек, если у всех туфли и платья были разного цвета.

Решение

```
PREDICATES
    имя(string)
    туфли(string)
    платье(string)
    соот(string,string,string)
    решение(string,string,string,string,string,string,
          string,string,string)
CLAUSES
    имя(наташа).
    имя(валя).
    имя(аня).
    туфли(белый).
    туфли(синий).
    туфли(зеленый).
    платье(белый).
    платье(синий).
    платье(зеленый).
    % X – имя,  Y – цвет туфель, Z – цвет платья
    соот(X,Y,Z):-имя(X),туфли(Y),платье(Z),
              X=наташа,Y=зеленый,Y<>Z.
    соот(X,Y,Z):-имя(X),туфли(Y),платье(Z),
              X=валя,not(Y=белый),
              not(Z=белый), Y<>Z.
    соот(X,Y,Z):-имя(X),туфли(Y),платье(Z),X=аня,Y=Z.
    решение(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3):-
                X1=наташа,соот(X1,Y1,Z1),
                X2=валя, соот(X2,Y2,Z2),
                X3=аня, соот(X3,Y3,Z3),
                Y1<>Y2, Y2<>Y3, Y1<>Y3,
                Z1<>Z2, Z2<>Z3, Z1<>Z3.
GOAL
    решение(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3),
    write(X1,"  туфли-  ",Y1," платье- ",Z1),nl,
```

```
        write(X2,"  туфли-  ",Y2," платье- ",Z2),nl,
        write(X3,"  туфли-  ",Y3," платье- ",Z3),nl.
```

Результат выполнения программы

```
        наташа   туфли -  зеленый  платье - синий
        валя     туфли -  синий    платье - зеленый
        аня      туфли -  белый    платье - белый
```

Пример 3

Витя, Юра и Миша сидели на скамейке. В каком порядке они сидели, если известно, что Миша сидел слева от Юры, а Витя слева от Миши.

Решение

```
PREDICATES
        слева(string,string)
        ряд(string,string,string)
CLAUSES
        /* Миша сидел слева от Юры */
        слева(миша, юра).
        /* Витя сидел слева от Миши */
        слева(витя, миша).
        /* Объекты X, Y и Z образуют ряд,
        если X слева от Y и Y слева от Z */
        ряд(X, Y, Z):- слева(X,Y), слева(Y, Z).
GOAL
        ряд(X, Y, Z), write(X,"-",Y,"-",Z),nl.
```

Результат выполнения программы

```
        витя-миша-юра
```

Пример 4

Известно, что тополь выше березы, которая выше липы. Клен ниже липы, а сосна выше тополя и ниже ели. Определить самое высокое и самое низкое дерево.

Решение

```
DOMAINS
        name=string
PREDICATES
        выше(name,name)
        ряд(name,name,name,name,name,name)
CLAUSES
        выше(тополь,береза).
        выше(липа,клен).
        выше(ель,сосна).
        выше(береза,липа).
        выше(сосна,тополь).
        ряд(X1,X2,X3,X4,X5,X6):-выше(X1,X2),выше(X2,X3),
                        выше(X3,X4),выше(X4,X5),
                        выше(X5,X6).
GOAL
        ряд(X,_,_,_,_,Y),write("Самое высокое - ",X),nl,
        write("Самое низкое - ",Y),nl.
```

Результат выполнения программы

```
        Самое высокое - ель
        Самое низкое - клен
```

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Трое ребят вышли гулять с собакой, кошкой и хомячком. Известно, что Петя не любит кошек и живет в одном подъезде с хозяйкой хомячка. Лена дружит с Таней, гуляющей с кошкой. Определить, с каким животным гулял каждый из детей.
2. Беседуют трое друзей: Белокуров, Рыжов и Чернов. Брюнет сказал Белокурову: «Любопытно, что один из нас блондин, другой – брюнет, а третий – рыжий, но ни у кого цвет волос не соответствует фамилии». Какой цвет волос у каждого из друзей?

3. Витя, Юра, Миша и Дима сидели на скамейке. В каком порядке они сидели, если известно, что Юра сидел справа от Димы, Миша справа от Вити, а Витя справа от Юры.
4. Известно, что Волга длиннее Амударьи, а Днепр короче Амударьи. Лена длиннее Волги. Определить вторую по протяженности реку.

Тема. СПИСКИ

План

1. *Списки в Прологе как рекурсивные объекты.*
2. *Способ объявления типа данных "список" в программе на Прологе.*
3. *Выполнение основных операций на списками.*
4. *Выполнение заданий для самостоятельной работы.*

Список – это объект, который содержит конечное число других объектов. Список в ПРОЛОГе заключается в квадратные скобки и элементы списка разделяются запятыми. Список, который не содержит ни одного элемента, называется *пустым* списком.

Список является рекурсивным объектом. Он состоит из *головы* (первого элемента списка) и *хвоста* (все последующие элемента). Хвост также является списком. В ПРОЛОГе имеется операция “|”, которая позволяет делить список на голову и хвост. Пустой список нельзя разделить на голову и хвост.

Тип данных "список" объявляется в программе на Прологе следующим образом:

```
DOMAINS
        списковый_тип = тип*
```

где "тип" - тип элементов списка; это может быть как стандартный тип, так и нестандартный, заданный пользователем и объявленный в разделе DOMAINS ранее.

Основными операциями на списками являются:

- формирование списка;
- объединение списков;
- поиск элемента в списке;
- вставка элемента в список и удаление из списка.
-

Пример 1

Сформировать список вида [7,6,5,4,3,2,1]

Решение

```
DOMAINS
        list = integer*
PREDICATES
        genl(integer, list)
CLAUSES
        genl(0,[]):-!.
        genl(N,[N|L]):-N1=N-1, genl(N1,L).
GOAL
        genl(7,L),write(L),nl.
```

Результат выполнения программы:

```
[7,6,5,4,3,2,1]
```

Пример 2

Сформировать список из N элементов, начиная с 2. Каждый следующий на 4 больше предыдущего.

Решение

```
DOMAINS
        list = integer*
PREDICATES
        genl( integer, integer, list )
CLAUSES
        genl(N2,N2,[]):-!.
        genl(N1,N2,[N1|L]):-N1<N2, N=N1+4,
                genl(N,N2,L).
GOAL
        write("N="),readint(N),K=4*(N+1)-2,
   genl(2,K,L),write(L),nl.
```

Результат выполнения программы:

N=5
[2,6,10,14,18]

Пример 3

Сформировать список последовательных натуральных чисел от 4 до 20 и найти количество его элементов.

Решение:

```
DOMAINS
        list = integer*
PREDICATES
        genl1(integer, integer, list )
        len(integer, list )
CLAUSES
        genl1(N2,N2,[]):-!.
        genl1(N1,N2,[N1|L]):-N1<N2, N=N1+1, genl1(N,N2,L).
        len(0,[]).
        len(X,[_|L]):-len(X1,L), X=X1+1.
GOAL
        genl1 (4,21,L ),write(L),nl,
        len(X, L),write("Количество элементов=",X),nl.
```

Результат выполнения программы:

[4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
Количество элементов=17

Пример 4

Определить, содержится ли введенное число X в заданном списке L.

Решение:

```
DOMAINS
    list = integer*
PREDICATES
    member(integer, list)
CLAUSES
    member(X,[X|_]):-write("yes"),!.
    member(X,[]):-write("no"),!.
    member(X,[_|L]) :- member(X, L).
GOAL
    L=[1,2,3,4], write(L),nl,     write("X="),readint(X),
    member(X, L),nl.
```

Результат выполнения программы:

1-й случай:

[1,2,3,4]
X=3
yes

2-й случай:

[1,2,3,4]
X=5
no

Пример 5

Сформировать списки L1=[1,2,3], L2=[10,11,12,13,14,15] и объединить их в список L3.

Решение:

```
DOMAINS
        list = integer*
PREDICATES
        genl1(integer,integer,list)
        append(list,list,list)
CLAUSES
        genl1(N2,N2,[]):-!.
        genl1(N1,N2,[N1|L]):-N1<N2,N=N1+1,genl1(N, N2, L).
        append([],L,L).
```

```
        append([X|L1],L2,[X|L3]):-append(L1,L2,L3).
GOAL
        genl1(1,4,L1),write("L1=",L1),nl,
        genl1(10,16,L2),write("L2=",L2),nl,
        append(L1,L2,L3),write("L3=",L3),nl.
```

Результат выполнения программы:

```
        L1=[1,2,3]
        L2=[10,11,12,13,14,15]
        L3=[1,2,3,10,11,12,13,14,15]
```

Пример 6

Удалить из списка, элементами которого являются названия дней недели, указанный элемент.

Решение:

```
DOMAINS
        list = symbol*
PREDICATES
        del(symbol,list,list)
CLAUSES
        del(X,[X|L],L).
        del(X,[Y|L],[Y|L1]):-del(X,L,L1).
GOAL
        L=[пн, вт, ср, чт, пт, сб, вс],write("L=",L),nl,
        write("X="),readln(X),
        del(X,L,L1),write("L1=",L1),!;
        write("Элемент отсутствует в списке"),nl.
```

Результат выполнения программы:

1-й случай:

```
        L=["пн","вт","ср","чт","пт","сб","вс"]
        X=ср
        L1=["пн","вт","чт","пт","сб","вс"]
```

2-й случай:

```
        L=["пн","вт","ср","чт","пт","сб","вс"]
        X=пр
        Элемент отсутствует в списке
```

Пример 7

Вставить в список имен новый элемент, значение которого вводится с клавиатуры. Вывести все возможные варианты вставок.

Решение:

```
DOMAINS
    list = symbol*
PREDICATES
    del(symbol,list,list)
    ins(symbol,list,list)
CLAUSES
    del(X,[X|L],L).
    del(X,[Y|L],[Y|L1]):-del(X,L,L1).
    ins(X,L1,L):-del(X,L,L1).
GOAL
    L=[olga, oksana, toma, dima],write("L=",L),nl,
    write("X="),readln(X),
    ins(X,L,L1),write("L1=",L1),nl, fail.
```

Результат выполнения программы:

```
    L=["olga","oksana","toma","dima"]
    X=vera
    L1=["vera","olga","oksana","toma","dima"]
    L1=["olga","vera","oksana","toma","dima"]
    L1=["olga","oksana","vera","toma","dima"]
    L1=["olga","oksana","toma","vera","dima"]
```

```
L1=["olga","oksana","toma","dima","vera"]
```

Пример 8

Найти сумму элементов списка целых чисел.

Решение:

```
DOMAINS
        list=integer*
PREDICATES
        sum_list(list, integer)
CLAUSES
        sum_list([],0).
        sum_list([X|L],S):-sum_list(L,S1),S=S1+X.
GOAL
        L=[1,2,3,4,5],sum_list(L,S), write("S=",S).
```

Результат выполнения программы:

```
S=15
```

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Сформировать список [2, 4, 6, 8, 10] и удалить из него введенное число.
2. Сформировать списки [1, 3, 5, 7, 9] и [2, 4, б, 8, 10] и объединить их в один.
3. Сформировать список [3, 6, 9, 12, 15, 18] и вставить в него введенное число.
4. Сформировать список из N натуральных чисел, начиная с 10. Каждое следующее на 5 больше предыдущего.
5. Сформировать список [3, 6, 9, 12, 15] и найти сумму его элементов
6. Сформировать список [6, 5, 4, 3, 2] и найти сумму его элементов
7. Сформировать список [7, 5, 3, 1] и найти произведение его элементов
8. Сформировать список из N последовательных натуральных чисел, начиная с 10. Найти сумму его элементов

10. Тематика рефератов/докладов/эссе, методические рекомендации по выполнению контрольных и курсовых работ, иные материалы

10.1. Задания для самостоятельной работы студентов по дисциплине Основы искусственного интеллекта

Задание (типовое)

Используя предикаты parent (symbol, symbol), man (symbol), woman (symbol), married (symbol,symbol), записать факты, описывающие Вашу семью. Записать 8 правил вывода для любых родственных отношений в Вашей семье (например: мать, отец, сестра, брат, племянница, племянник, тетя, дядя, внучка, внук, бабушка, дедушка, двоюродная сестра, двоюродный брат и т.д.).

При отладке программы изучить и использовать возможности трассировки.

Написать программу без использования внутренней цели (без секции GOAL).

Во всех прочих программах используется внутренняя цель (в секции GOAL)

Методические указания

Как правило, программа на Прологе состоит из двух программных секций: секции предикатов PREDICATES и секции предложений CLAUSES.

В секции PREDICATES описываются собственные предикаты. Описание предикатов заключается в их перечислении с указанием доменов (типов) их аргументов. Стандартные предикаты объявлять не требуется. Формат объявления предиката:

predicate_name (argument_domain_1, argument_domain_2, …, argument_domain_n)

В секции CLAUSES помещаются предложения – факты и правила, составляющие программу. Все предложения для одного предиката должны быть сгруппированы. Каждое предложение заканчивается точкой. Формат записи предложений:

fact(object_1, object_2, …, object_n).

rule(Var_1, Var_2, …,Var_m):–subgoal_1, subgoal_2, …, subgoal_k.

Для отладки программы можно использовать возможности трассировки. Трассировка позволяет в пошаговом режиме проследить процесс нахождения решения.

Для того чтобы включить трассировку, можно первой строкой программы поместить директиву trace. Для пошагового выполнения программы в режиме трассировки используется клавиша F10.
При работе в режиме трассировки вся текущая информация появляется в окне трассировки Trace.
Сообщения в окне трассировки могут быть следующими:
- CALL – вывод имени цели и значений ее параметров;
- FAIL – вывод имени неудачно завершившейся цели;
- REDO – вывод сообщения о том, что произведен поиск с возвратом;
- RETURN – вывод имени удачно завершившейся цели и значений ее параметров (символ * показывает, что существуют другие решения).

Примеры выполнения заданий.

РЕКУРСИЯ

```
PREDICATES
factorial (integer, integer)
CLAUSES
factorial (0, 1):- !.
factorial (N, Factorial_N):- M=N–1, factorial (M, Factorial_M), Factorial_N=Factorial_M*N.
GOAL
write (“Для какого числа Вы хотите найти факториал? ”), readint (Number), factorial (Number, Result), write (Number, “!=”, Result).
```

Пример хвостовой рекурсии

```
PREDICATES
counter (integer)
CLAUSES
counter (N):- N>0, !, write (“N=”, N), nl, New_N=N–1, counter (New_N).
counter (N):- write (“Отрицательное N=”, N).
GOAL
counter (1000).
```

Пример нехвостовой рекурсии

```
PREDICATES
counter (integer)
check (integer)
CLAUSES
counter (N):- check (N), write (“N=”, N), nl, New_N=N–1, counter (New_N).
check (N):- N>0, write (“Положительное ”).
check (_):- write (“Отрицательное ”).
GOAL
counter (1000).
```

Подсчет числа элементов списка или, другими словами, определение длины списка.

```
DOMAINS
intlist=integer*
PREDICATES
list_length (intlist, integer)
CLAUSES
list_length ([ ], 0):- !.
list_length (List, Length):- List=[H | T], list_length (T, Length_T), Length=Length_T+1.
GOAL
listlength ([1, 2], Length), write(“Length=”, Length).
```

Поэлементный вывод дерева, в узлах которого хранятся целые числа, на экран

```
DOMAINS
treetype=tree (integer, treetype, treetype) ; empty
```

```
PREDICATES
printtree (treetype)
CLAUSES
printtree (empty):- !.
printtree (tree (Root, Left, Right)):- write (Root), write (" ~ "), printtree (Left), printtree (Right).
GOAL
printtree (tree (5, tree (2, tree (–1, empty, empty), tree (4, empty, empty)), tree (9, tree(7, empty, empty), empty))).
```

Результат работы программы обхода дерева «сверху вниз»:
5 ~ 2 ~ –1 ~ 4 ~ 9 ~ 7 ~

ВАРИАНТЫ ЗАДАНИЙ ДЛЯ САМОСТОЯТЕЛЬНЫХ РАБОТ
Поиск с возвратом

Задания:

Вариант 1

Написать программу, реализующую телефонный справочник. В справочнике содержится следующая информация о каждом абоненте: имя и телефон. Реализовать вывод всей информации из справочника, поиск телефона по имени, поиск имени по телефону. Для удобства работы реализовать меню с соответствующими пунктами.

Вариант 2

Написать программу, реализующую географический справочник. В справочнике содержится следующая информация: названия стран и площади страны. Реализовать вывод всей информации из справочника, поиск по названию. Реализовать поиск по площади, при этом должна быть возможность ввести некоторое пороговое значение (например, вывести названия всех стран, площадь которых не менее 3 млн. км^2). Для удобства работы реализовать меню с соответствующими пунктами.

Вариант 3

Написать программу, реализующую калькулятор на четыре арифметических действия (без скобок).

Вариант 4

Написать программу, реализующую словарь. В словаре содержится следующая информация: слово и его несколько переводов. Реализовать вывод всего словаря, перевод с русского на английский, с английского на русский. Для удобства работы реализовать меню с соответствующими пунктами.

Вариант 5

Написать программу, реализующую авиасправочник. В справочнике содержится следующая информация о каждом рейсе: номер рейса, пункт назначения, цена билета. Реализовать вывод всей информации из справочника, поиск пункта назначения по номеру рейса. Реализовать поиск по пункту назначения с указанием максимально возможной цены билета (должны быть выведены все рейсы, цена билета на которые ниже указанного значения). Для удобства работы реализовать меню с соответствующими пунктами.

Вариант 6

Написать программу для продажи театральных билетов. Должна быть представлена следующая информация: спектакль, свободные места, цена билета. Реализовать вывод всей информации о билетах, поиск билета по ряду. Реализовать поиск по цене с указанием максимально возможной цены (должна быть выведена информация о билетах, цены на которые ниже указанного значения). Для удобства работы реализовать меню с соответствующими пунктами.

Вариант 7

Написать программу, реализующую автомагазин. Должна быть представлена следующая информация о каждом автомобиле: модель, мощность двигателя, цвет, цена. Реализовать вывод всей информации по автомобилям, поиск по цвету. Реализовать поиск по мощности с указанием минимальной мощности (должна быть выведена информация обо всех моделях, мощность которых выше указанного значения). Для удобства работы реализовать меню с соответствующими пунктами.

Вариант 8

Написать программу, реализующую книжный магазин. Должна быть представлена следующая информация: название книги, количество экземпляров, цена. Реализовать вывод всей информации о книгах, поиск книги по названию. Реализовать поиск по цене с указанием интервала возможной цены (должна быть выведена информация о книгах, цены которых попадают в указанный интервал). Для удобства работы реализовать меню с соответствующими пунктами.

Вариант 9

Написать программу для продажи туристических туров. Должна быть представлена следующая информация: название тура, страна, продолжительность, цена. Реализовать вывод информации обо всех турах, поиск тура по стране. Реализовать поиск по продолжительности с указанием интервала возможной продолжительности (должна быть выведена информация о турах, продолжительность которых попадает в указанный интервал). Для удобства работы реализовать меню с соответствующими пунктами.

Вариант 10

Написать программу для заказа мест в отеле. Должна быть представлена следующая информация: название отеля, класс отеля, свободные места, цена номера. Реализовать вывод информации обо всех свободных номерах, поиск отеля по классу. Реализовать поиск по цене с указанием максимально возможной цены (должна быть выведена информация о номерах, цены на которые ниже указанного значения) Для удобства работы реализовать меню с соответствующими пунктами.

Рекурсия

Задания:

Вариант 1

Вычислить значение n-го члена ряда Фибоначчи: f(0)=0, f(1)=1, f(n)=f(n-1)+f(n-2).

Вариант 2

Вычислить произведение двух целых положительных чисел (используя суммирование).

Вариант 3

Подсчитать, сколько раз встречается некоторое слово в строке. Строка и слово должны вводиться с клавиатуры. Для разделения строки на слова использовать стандартный предикат fronttoken (String, Lexeme, StringRest), позволяющий разделить строку String на первое слово Lexeme и остаток строки StringRest.

Вариант 4

Поменять порядок следования букв в слове на противоположный. Для разделения строки на символы использовать стандартный предикат frontchar (String, Char, StringRest), позволяющий разделять строку String на первый символ Char и остаток строки StringRest.

Вариант 5

Вычислить сумму ряда целых нечетных чисел от 1 до n.

Вариант 6

Поменять порядок следования слов в предложении на противоположный. Для разделения строки на слова использовать стандартный предикат fronttoken (String, Lexeme, StringRest), позволяющий разделить строку String на первое слово Lexeme и остаток строки StringRest..

Вариант 7

Вычислить сумму ряда целых четных чисел от 2 до n.

Вариант 8

Организовать ввод целых положительных чисел и их суммирование до тех пор, пока сумма не превысит некоторого порогового значения. Введенные отрицательные целые числа суммироваться не должны.

Вариант 9

Организовать ввод букв и их соединение в строку до тех пор, не будет введен символ #. Для присоединения символа к строке использовать стандартный предикат frontchar (String, Char, StringRest), позволяющий присоединять символ Char к строке StringRest и получать строку String.

Вариант 10

Подсчитать, сколько раз встречается некоторая буква в строке. Строка и буква должны вводиться с клавиатуры. Для разделения строки на символы использовать стандартный предикат frontchar (String, Char, StringRest), позволяющий разделять строку String на первый символ Char и остаток строки StringRest.

Рекурсивные структуры данных (списки)

Задание:

Вариант 1

Написать программу для получения значения n-го элемента списка. Например: в списке [three, one, two] второй элемент равен one.

Вариант 2

Написать программу для удаления из списка всех элементов, равных 0. Например: список [1, 0, 2, 0, 0, 3] преобразуется в список [1, 2, 3].

Вариант 3

Написать программу для циклического сдвига списка вправо на заданное число элементов. Например: список [6, 5, 4, 3, 2, 1], циклически сдвинутый вправо на 2 элемента, преобразуется в список [2, 1, 6, 5, 4, 3].

Вариант 4

Написать программу для удаления из списка 2-ого, 4-ого и т.д. элементов. Например: список [6, 5, 4, 3, 2, 1] преобразуется в список [6, 4, 2].

Вариант 5

Написать программу для замены в списке всех элементы, равные 0, на -1. Например: список [1, 0, 0] преобразуется в список [1, -1, -1].

Вариант 6

Написать программу для перевода списка арабских чисел (от 1 до 10) в список римских чисел. Например: список [1, 2, 3] преобразуется в список [“I”, “II”, “III”].

Вариант 7

Написать программу для подсчета количества определенных элементов в списке. Например: в списке [1, 2, 1, 3, 1] три единицы.

Вариант 8

Написать программу для подсчета количества элементов списка без какого-либо указываемого элемента. Например: в списке [1, 2, 1, 3, 1] два элемента без учета единиц.

Вариант 9

Написать программу для подсчета количества элементов списка, значения которых лежат в определенном диапазоне. Например: в списке [10, 20, 10, 30, 15] два элемента, значения которых больше 10 и меньше 30.

Вариант 10

Написать программу для реверса списка. Например: список [1, 2, 3] преобразуется в список [3, 2, 1].

Рекурсивные структуры данных (деревья)

Задание:

Вариант 1

Написать программу для проверки упорядоченности бинарного дерева.

Вариант 2

Вывести бинарное дерево на экран в виде дерева.

Вариант 3

Написать программу для вычисления глубины бинарного дерева (глубина пустого дерева равна 0, глубина одноузлового дерева равна 1).

Вариант 4

Написать программу для подсчета количества листьевых вершин дерева, значения которых лежат в определенном диапазоне.

Вариант 5

Написать программу для преобразования дерева в список.

Вариант 6

Написать программу для нахождения среднего арифметического отрицательных узлов дерева.

Вариант 7

Написать программу для подсчета количества вершин бинарного дерева, значения которых не равны 0.

Вариант 8

Написать программу для нахождения среднего арифметического положительных узлов дерева.

Вариант 9

Написать программу для подсчета количества вершин бинарного дерева, значения которых равны 0.

Вариант 10
Написать программу для нахождения среднего арифметического листьевых вершин бинарного дерева.

Презентации по дисциплине Основы искусственного интеллекта:

1. Интеллектуальные системы
2. Модели представления знаний
3. Экспертные системы
4. Язык логического программирования ПРОЛОГ