Основы программирования на языке Пролог

Курс лекций. Учебное пособие

П.А. Шрайнер

Рекомендовано для студентов высших учебных заведений, обучающихся по специальностям в области информационных технологий

Серия «Основы информационных технологий»

Интернет-Университет Информационных Технологий
www.intuit.ru
Москва, 2005

ББК 32.973.26-018.1я73
УДК 004.438(075.8)
Ш85

Ш85 Основы программирования на языке Пролог : курс лекций : учеб. пособие для студентов вузов, обучающихся по специальностям в обл. информ. технологий / П. А. Шрайнер. - М. : Интернет - Ун-т Информ. Технологий, 2005. - 176 с. - (Серия «Основы информационных технологий» / Интернет-Ун-т информ. технологий). - ISBN 5-9556-0034-5.

Курс посвящен изучению основ языка логического программирования Пролог. Общие принципы программирования на Прологе изучаются всюду, где это возможно, без привязки к конкретной реализаци.

Рекомендовано для студентов высших учебных заведений, обучающихся по специальностям в области информационных технологий.

Библиогр. 15

Издание осуществлено при финансовой и технической поддержке компаний:
Издательство «Открытые Системы», «РМ Телеком» и Kraftway Computers.

ISBN 5-9556-0034-5

О проекте

Интернет-Университет Информационных Технологий – это первое в России высшее учебное заведение, которое предоставляет возможность получить дополнительное образование во Всемирной сети. Web-сайт университета находится по адресу www.intuit.ru.

Мы рады, что вы решили расширить свои знания в области компьютерных технологий. Современный мир – это мир компьютеров и информации. Компьютерная индустрия – самый быстрорастущий сектор экономики, и ее рост будет продолжаться еще долгое время. Во времена жесткой конкуренции от уровня развития информационных технологий, достижений научной мысли и перспективных инженерных решений зависит успех не только отдельных людей и компаний, но и целых стран. Вы выбрали самое подходящее время для изучения компьютерных дисциплин. Профессионалы в области информационных технологий сейчас востребованы везде: в науке, экономике, образовании, медицине и других областях, в государственных и частных компаниях, в России и за рубежом. Анализ данных, прогнозы, организация связи, создание программного обеспечения, построение моделей процессов – вот далеко не полный список областей применения знаний для компьютерных специалистов.

Обучение в университете ведется по собственным учебным планам, разработанным ведущими российскими специалистами на основе международных образовательных стандартов Computer Curricula 2001 Computer Science. Изучать учебные курсы можно самостоятельно по учебникам или на сайте Интернет-Университета, задания выполняются только на сайте. Для обучения необходимо зарегистрироваться на сайте университета. Удостоверение об окончании учебного курса или специальности выдается при условии выполнения всех заданий к лекциям и успешной сдачи итогового экзамена.

Книга, которую вы держите в руках, очередная в многотомной серии «Основы информационных технологий», выпускаемой Интернет-Университетом Информационных Технологий. В этой серии будут выпущены учебники по всем базовым областям знаний, связанным с компьютерными дисциплинами.

Добро пожаловать в Интернет-Университет Информационных Технологий!

Анатолий Шкред
anatoli@shkred.ru

Об авторе

Шрайнер Павел Александрович

Кандидат физико-математических наук, доцент Сибирского Университета Потребительской Кооперации, сотрудник Института математики им. С.Л. Соболева СО РАН.

Закончил механико-математический факультет НГУ, имеет более 30 научных публикаций, принимал участие в международных европейских и российских конференциях и школах.

Автором курса совместно с археологами из НГПУ была разработана на Прологе экспертная система, предназначенная для анализа керамических комплексов археологических памятников. На основе результатов работы этой системы было сделано несколько докладов на конференциях, защищена кандидатская диссертация по археологии.

Исследовательский проект автора курса, посвященный автоматическому распознаванию различных свойств неклассических логик был поддержан в 2002-2003 годах грантом Минобразования РФ, а в 2003-2004 годах — грантом РФФИ. Содержание проекта: разработка и реализация на Прологе алгоритмов автоматического распознавания различных свойств неклассических логик.

Оглавление

Внимание!

На сайте Интернет-университета информационных технологий Вы можете пройти тестирование по каждой лекции и курсу в целом.

Добро пожаловать на наш сайт:

www.intuit.ru

Лекция 1. Введение в язык логического программирования Пролог

История возникновения и развития Пролога. Японский проект ЭВМ пятого поколения. Императивные и декларативные языки программирования. Области использования Пролога. Пролог — язык начального обучения программированию. Преимущества и недостатки языка Пролог.

Ключевые слова: императивные языки, декларативные языки, язык Пролог.

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 (Institute for New Generation Computer Technology Research Center). 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 архитектуры компьютеров фон Неймана. 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 архитектура фон Неймана 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. Думается, что и 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 ЛОГическое ПРОграммирование (PROgrammation en LOGique во французском варианте и PROgramming in LOGic — в английском).

Пролог основывается на таком разделе математической логики, как исчисление предикатов. Точнее, его базис составляет процедура доказательства теорем методом резолюции для хорновских дизъюнктов. Этой теме будет посвящена следующая лекция.

В истории возникновения и развития языка Пролог можно выделить следующие этапы.

В 1965 году в работе «A machine oriented logic based on the resolution principle», опубликованной в 12 номере журнала «Journal of the ACM», Дж. Робинсон представил метод автоматического поиска доказательства теорем в исчислении предикатов первого порядка, получивший название «принцип резолюции». Эту работу можно прочитать в переводе: Робинсон Дж. Машинно-ориентированная логика, основанная на принципе резолюции // Кибернетический сборник. — Вып. 7 (1970). На самом деле, идея данного метода была предложена Эрбраном в 1931 году, когда еще не было компьютеров (Herbrand, «Une methode de demonstration», These, Paris, 1931). Робинсон модифицировал этот метод так, что он стал пригоден для автоматического, компьютерного использования, и, кроме того, разработал эффективный алгоритм унификации, составляющий базис его метода.

В 1973 году «группа искусственного интеллекта» во главе с Аланом Колмероэ создала в Марсельском университете программу, предназначенную для доказательства теорем. Эта программа использовалась при построении систем обработки текстов на естественном языке. Программа доказательства теорем получила название Prolog (от Programmation en Logique). Она и послужила прообразом Пролога. Ходят легенды, что автором этого названия была жена Алана Колмероэ. Программа была написана на Фортране и работала довольно медленно.

Большое значение для развития логического программирования имела работа Роберта Ковальского «Логика предикатов как язык программирования» (Kowalski R. Predicate Logic as Programming Language. IFIP Congress, 1974), в которой он показал, что для того чтобы добиться эффективности, нужно ограничиться использованием множества хорновских дизъюнктов. Кстати, известно, что Ковальский и Колмероэ работали вместе в течение одного лета.

В 1976 году. Ковальский вместе с его коллегой Маартеном ван Эмденом предложил два подхода к прочтению текстов логических программ: процедурный и декларативный. Об этих подходах речь пойдет в третьей лекции.

В 1977 году в Эдинбурге Уоррен и Перейра создали очень эффективный компилятор языка Пролог для ЭВМ DEC-10, который послужил прототипом для многих последующих реализаций Пролога. Что интересно, компилятор был написан на самом Прологе. Эта реализация Пролога, известная как «эдинбургская версия», фактически стала первым и единственным стандартом языка. Алгоритм, использованный при его реализации, послужил прототипом для многих последующих реализаций языка. Как правило, если современная Пролог-система и не поддерживает эдинбургский Пролог, то в ее состав входит подсистема, переводящая прологовскую программу в «эдинбургский» вид. Имеется, конечно, стандарт ISO/IEC 13211– 1:1995, но его поддерживают далеко не все Пролог-системы.

В 1980 году Кларк и Маккейб в Великобритании разработали версию Пролога для персональных ЭВМ.

В 1981 году стартовал вышеупомянутый проект Института по разработке методов создания компьютеров нового поколения.

На 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).

Стерлинг и Шапиро в книге «Искусство программирования на языке Пролог» пишут: «Зрелость языка означает, что он больше не является доопределяемой и уточняемой научной концепцией, а становится реальным объектом со всеми присущими ему пороками и добродетелями. Настало время признать, что хотя Пролог и не достиг высоких целей логического программирования, но, тем не менее, является мощным, продуктивным и практически пригодным формализмом программирования».

Традиционно под программой понимают последовательность операторов (команд, выполняемых компьютером). Этот стиль программирования принято называть *императивным*. 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 модели вычислений компьютера фон Нейман. 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.

При программировании на Прологе усилия программиста должны быть направлены на описание логической модели фрагмента предметной

области решаемой задачи в терминах объектов предметной области, их свойств и отношений между собой, а не деталей программной реализации. 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, в духе модного сейчас объектно-ориентированного программирования.

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. Проверим эти утверждения на собственном опыте при изучении данного курса.

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.

Мне приходилось обучать Прологу школьников и студентов. Однажды занятие спецкурса по Прологу посетила одна преподавательница информатики. К тому времени у нее был довольно большой опыт программирования (и преподавания) на императивных языках. После окончания

занятия она долго не могла прийти в себя. Реакция у нее была примерно следующая: «Я не понимаю, как такое может быть! Как программа в несколько строк на Прологе может делать то, на что в программе на Паскале понадобится несколько страниц текста?» Несмотря на то, что человеком она была далеко не глупым, она так и не смогла преодолеть «императивную зашоренность» и начать программировать в декларативном стиле. В отличие от нее, студенты и школьники, даже уже имеющие опыт программирования на императивных языках, но еще не «закостеневшие» в этом направлении, довольно легко воспринимали декларативный подход и без труда начинали программировать на Прологе. Так, один из школьников (семиклассник) после пары месяцев изучения Пролога написал на нем приставку к системе бронирования авиабилетов «Габриель», которая использовалась в новосибирском «Аэрофлоте».

Как язык программирования Пролог очень хорошо, на мой взгляд, подходит для начального обучения программированию, так как он ориентирован на человеческое мышление – в отличие от императивных языков, ориентированных на компьютер. Теми, кто только начинает изучать программирование, Пролог осваивается легко. Практически полное отсутствие синтаксических конструкций, таких как ветвления, циклы и т.д. также влияет на скорость освоения языка. Кроме того, программирование на Прологе, как мне кажется, упорядочивает мышление и позволяет человеку, изучающему этот язык программирования, лучше разобраться в своей мыслительной деятельности. Очень часто для того, чтобы запрограммировать решение задачи, программисту (или эксперту в некоторой предметной области) вначале требуется понять, как он сам решает эту задачу, провести некую формализацию, перевести свои знания из неявных в явные.

Однако, в отличие от некоторых приверженцев Пролога, я не склонен думать, что Пролог — лучший язык всех времен и народов, универсальный «решатель» любой задачи. На наш взгляд, 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. Поэтому совсем даже неплохо, если человек владеет не одним инструментом решения задач, а может воспользоваться наиболее подходящим из имеющихся в его распоряжении. 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.

Изучать Пролог без привязки к конкретной его версии, мне кажется, не совсем целесообразно. Как уже было сказано выше, версий Пролога очень много, и нужно выбрать какую-нибудь одну из них, чтобы привязать к этой версии разбираемые примеры. Мы остановимся на наиболее известной у нас в стране и довольно эффективной версии Пролога — Турбо Прологе. 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.

Сейчас вышла шестая версия системы Visual Prolog, которая, однако, довольно далеко шагнула в сторону не только от эдинбургской версии Пролога, но даже и от своей пятой версии, которая практически без проблем принимала программы на Турбо Прологе.

Исходя из соображений малого объема, доступности, малой ресурсоемкости, традиций, а также отсутствия всего лишнего, в том числе графической оболочки, остановимся на Турбо Прологе 2.0, хотя, наверное, это выбор довольно спорный. Но наша цель на данный момент — сосредоточиться на самом языке. Всюду, где это возможно, мы будем изучать Пролог, желательно как можно ближе к «чистому» Прологу. Попытаемся разобраться и с теоретическими основами этого языка. Но, тем не менее, я считаю, что программирование стоит осваивать, имея доступ к компьютеру с установленной на нем конкретной версией изучаемого языка. Все разбираемые в лекциях примеры будут работоспособны во второй версии Турбо Пролога и выше. В частности, они должны компилироваться в Visual Prolog версий 4–5.2. Как правило, их можно без особых проблем перенести и в другие версии Пролога. При этом, возможно, потребуется легкая модификация. Например, замена конструкции «*:* — »на «*is*» и т.д.

Самое существенное отличие Турбо Пролога от эдинбургской версии (так называемого «классического» Пролога) — наличие в нем строгой типизации данных для повышения скорости трансляции и выполнения программ. В начале программы на Турбо Прологе обычно располагаются разделы описаний доменов (типов данных) и предикатов. В Турбо Прологе отсутствует возможность рассматривать правила как данные, т. е. добавлять и удалять их во время работы, сопоставлять имя предиката с переменной. Изменяемой частью программы является внутренняя база данных (их может быть несколько). Во время выполнения программы в нее можно добавлять и из нее можно удалять факты.

В отличие от «классического» Пролога в Турбо Прологе нельзя определять операции. Турбо Пролог является компилируемым, а не интерпретируемым языком. К достоинствам Турбо Пролога относится возможность присоединять к программе на этом языке процедуры, написанные на Паскале, Си, Фортране или Ассемблере.

Лекция 2. Логические основы Пролога

Хорновские дизъюнкты. Принцип резолюций. Алгоритм унификации. Процедура доказательства теорем методом резолюций для хорновских дизъюнктов. Особенности работы с негативными знаниями в Прологе.

Ключевые слова: метод резолюций, унификация, хорновские дизъюнкты.

Эта лекция будет посвящена теоретическим основам языка Пролог. В принципе, вполне можно писать хорошие программы на языке Пролог, не вдаваясь в глубины математической логики. И в этом смысле можно считать эту главу необязательной, факультативной. Однако тем, кому интересно узнать, «как она вертится», мы попробуем объяснить, как устроен Пролог, на чем он основывается.

Давайте начнем с самого начала или почти с самого начала, раз уж мы договорились, что никаких предварительных навыков от слушателей не требуется. Нам придется попытаться разобраться с понятиями логики первого порядка, которая лежит в основе Пролога; они обычно изучаются в курсе математической логики. Конечно, для того чтобы изучить даже самые начала математической логики, одной лекции недостаточно. Поэтому мы попытаемся пробежаться только по тому кусочку, который имеет отношение к языку Пролог. Часть используемых нами понятий все-таки останется «за кадром».

Говорят, что задана некая формальная система F, если определены:

1. *алфавит* системы — счетное множество символов;
2. *формулы* системы — некоторое подмножество всех слов, которые можно образовать из символов, входящих в алфавит (обычно задается процедура, позволяющая составлять формулы из символов алфавита системы);
3. *аксиомы* системы — выделенное множество формул системы;
4. *правила* вывода системы — конечное множество отношений между формулами системы.

Зададим логику первого порядка (или логику предикатов), на которой основывается Пролог. Язык логики предикатов — один из формальных языков, наиболее приближенных к человеческому языку.

Алфавит логики первого порядка составляют следующие символы:

1. переменные (будем обозначать их последними буквами английского алфавита **u, v, x, y, z**);
2. константы (будем обозначать их первыми буквами английского алфавита **a, b, c, d**);

3. функциональные символы (используем для их обозначения ближние буквы **f** и **g**);
4. предикатные символы (обозначим их дальними буквами **p, q** и **r**);
5. пропозициональные константы истина и ложь (**true** и **false**);
6. логические связки ¬ (отрицание), ∧ (конъюнкция), ∨ (дизъюнкция), → (импликация);
7. кванторы: ∃ (существования), ∀ (всеобщности);
8. вспомогательные символы **(**, **)**, **,**.

Всякий предикатный и функциональный символ имеет определенное число аргументов. Если предикатный (функциональный) символ имеет *n* аргументов, он называется *n-местным* предикатным (функциональным) символом.

Термом будем называть выражение, образованное из переменных и констант, возможно, с применением функций, а точнее:

1. всякая переменная или константа есть терм;
2. если $t_1, \ldots, t_n$ — термы, а f — n-местный функциональный символ, то $f(t_1, \ldots, t_n)$ — терм;
3. других термов нет.

По сути дела, все объекты в программе на Прологе представляются именно в виде термов.

Если терм не содержит переменных, то он называется *основным* или *константным термом*.

Атомная или *элементарная формула* получается путем применения предиката к термам, точнее, это выражение $p(t_1, \ldots, t_n)$, где p — n-местный предикатный символ, а $t_1, \ldots, t_n$ — термы.

Формулы логики первого порядка получаются следующим образом:

1. всякая атомная формула есть формула;
2. если A и B — формулы, а x —переменная, то выражения ¬A (читается «не A» или «отрицание A»), A∧B (читается «A и B»), A∨B (читается «A или B»), A→B (читается «A влечет B»), ∃xA (читается «для некоторого x» или «существует x») и ∀xA (читается «для любого x» или «для всякого x»)— формулы;
3. других формул нет.

В случае если формула имеет вид ∀xA или ∃xA, ее подформула A называется *областью действия квантора* ∀x или ∃x соответственно. Если вхождение переменной x в формулу находится в области действия квантора ∀x или ∃x, то оно называется *связанным* вхождением. В противном случае вхождение переменной в формулу называется *свободным*.

Чтобы не увеличивать чрезмерно объем лекции, мы не будем рассматривать полный список аксиом и правил вывода логики первого порядка. Те из них, которые пригодятся нам в дальнейшем, будут приведены в соответствующих местах.

Литералом будем называть атомную формулу или отрицание атомной формулы. Атом называется *положительным литералом*, а его отрицание — *отрицательным литералом*.

Дизъюнкт — это дизъюнкция конечного числа литералов. Если дизъюнкт не содержит литералов, его называют *пустым дизъюнктом* и обозначают посредством символа □.

Давайте посмотрим, как можно привести любую формулу к множеству дизъюнктов, с которым работает метод резолюций. Для этого нам понадобятся некоторые определения нормальных форм.

Говорят, что формула находится в *конъюнктивной нормальной форме*, если это конъюнкция конечного числа дизъюнктов. Имеет место теорема о том, что для любой бескванторной формулы существует формула, логически эквивалентная исходной и находящаяся в конъюнктивной нормальной форме.

Формула находится в *предваренной* (или *пренексной*) *нормальной форме*, если она представлена в виде $Q_1x_1,...,Q_nx_nA$, где Q_i — это квантор ∀ или ∃, а формула A не содержит кванторов. Выражение $Q_1x_1,...,Q_nx_n$ называют *префиксом*, а формулу A — *матрицей*.

Формула находится в *сколемовской нормальной форме*, если она находится в предваренной нормальной форме и не содержит кванторов существования.

Алгоритм приведения произвольной формулы исчисления предикатов к множеству дизъюнктов

Первый шаг. Приводим исходную формулу к предваренной нормальной форме. Для этого:

1. пользуясь эквивалентностью $A \to B \equiv \neg A \vee B$ исключим импликацию;
2. перенесем все отрицания внутрь формулы, чтобы они стояли только перед атомными формулами, используя следующие эквивалентности:

   $\neg(A \vee B) \equiv \neg A \wedge \neg B$

   $\neg(A \wedge B) \equiv \neg A \vee \neg B$

   $\neg(\exists x A) \equiv \forall x \neg A$

   $\neg(\forall x A) \equiv \exists x \neg A$

   $\neg\neg A \equiv A$
3. переименовываем связанные переменные так, чтобы ни одна переменная не входила в нашу формулу одновременно связанно и свободно.
4. выносим кванторы в начало формулы, используя эквивалентности:

$QxA(x) \vee B \equiv Qx(A(x) \vee B)$, если B не содержит переменной x, а $Q \in \{\forall, \exists\}$

$QxA(x) \wedge B \equiv Qx(A(x) \wedge B)$, если B не содержит переменной x, а $Q \in \{\forall, \exists\}$

$\forall xA(x) \wedge \forall xB(x) \equiv \forall x(A(x) \wedge B(x))$

$\exists xA(x) \vee \exists xB(x) \equiv \exists x(A(x) \wedge B(x))$

$Q_1xA(x) \vee Q_2xB \equiv Q_1xQ_2y(A(x) \vee B(y))$, где $Q \in \{\forall, \exists\}$

$Q_1xA(x) \wedge Q_2xB \equiv Q_1xQ_2y(A(x) \wedge B(y))$, где $Q \in \{\forall, \exists\}$

Второй шаг. Проведем сколемизацию, т.е. элиминируем в формуле кванторы существования. Для этого для каждого квантора существования выполним следующий алгоритм.

Если устраняемый квантор существования — самый левый квантор в префиксе формулы, заменим все вхождения в формулу переменной, связанной этим квантором, на новую константу и вычеркнем квантор из префикса формулы.

Если левее этого квантора существования имеются кванторы всеобщности, заменим все вхождения в формулу переменной, связанной этим квантором, на новый функциональный символ от переменных, которые связаны левее стоящими кванторами всеобщности, и вычеркнем квантор из префикса формулы.

Проведя этот процесс для всех кванторов существования, получим формулу, находящуюся в сколемовской нормальной форме. Алгоритм устранения кванторов существования придумал Сколем в 1927 году.

Имеет место теорема о том, что формула и ее сколемизация эквивалентны в смысле выполнимости.

Третий шаг. Элиминируем кванторы всеобщности. Полученная формула будет бескванторной и эквивалентной исходной в смысле выполнимости.

Четвертый шаг. Приведем формулу к конъюнктивной нормальной форме, для чего воспользуемся эквивалентностями, выражающими дистрибутивность:

$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$

$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$

Пятый шаг. Элиминируем конъюнкции, представляя формулу в виде множества дизъюнктов.

Получаем множество дизъюнктов, эквивалентное исходной формуле в том смысле, который дает нам следующая теорема.

Теорема. Формула является тождественно ложной тогда и только тогда, когда множество дизъюнктов, полученных из нее, является невыполнимым.

Напомним, что множество формул называется *невыполнимым*, если не существует такого означивания переменных, чтобы все формулы из этого множества были бы истинными.

Пример. Превратим формулу $\forall x(P(x) \rightarrow \exists y(P(y) \vee \neg Q(x,y)))$ в эквивалентное ей множество дизъюнктов.

Первый шаг. Приведем исходную формулу к предваренной нормальной форме. Элиминировав импликацию, получим формулу $\forall x(\neg P(x) \vee \exists y(P(y) \vee \neg Q(x,y)))$. Вынесем переменную y за скобки: $\forall x \exists y(\neg P(x) \vee (P(y) \vee \neg Q(x,y)))$. Это можно сделать, потому что формула $\neg P(x)$ не зависит от переменной y. Если бы она зависела, то можно было бы переименовать связанную переменную y.

Второй шаг. Проведем сколемизацию полученной формулы. Левее квантора существования стоит квантор всеобщности, значит, нужно заменить все вхождения переменной y новым унарным функциональным символом, зависящим от x. Получим формулу, находящуюся в сколемовской нормальной форме: $\forall x(\neg P(x) \vee (P(f(x) \vee \neg Q(x,f(x)))))$.

Третий шаг. Элиминируем квантор всеобщности: $\neg P(x) \vee (P(f(x)) \vee \neg Q(x,f(x)))$.

В четвертом и пятом шагах необходимости нет, поскольку формула уже представляет собой дизъюнкт: $\neg P(x) \vee P(f(x)) \vee \neg Q(x,f(x))$.

Следующая техника, лежащая в основе Пролога, с которой мы попробуем разобраться, — это *унификация*. Унификация позволяет отождествлять формулы логики первого порядка путем замены свободных переменных на термы.

Подстановка — это множество вида $\{x_1/t_1, \ldots, x_n/t_n\}$, где для всех i, x_i — переменная, а t_i — терм, причем $x_i \neq t_i$ (отображение переменных в термы). При этом все переменные, входящие в подстановку, различны (для любого $i \neq j$ $x_i \neq x_j$).

Символом **ε** будем обозначать пустую подстановку.

Подстановка, в которой все термы основные, называется *основной подстановкой*.

Простое выражение — это терм или атомная формула.

Если A — простое выражение, а θ — подстановка, то $A\theta$ получается путем одновременной замены всех вхождений каждой переменной из θ в A соответствующим термом. $A\theta$ называется *частным случаем (примером)* выражения A. Содержательно подстановка заменяет каждое вхождение переменной x_i на терм t_i.

Пусть θ и η — подстановки, $\theta=\{x_1/t_1, \ldots, x_n/t_n\}$, $\eta=\{y_1/s_1, \ldots, y_n/s_n\}$. Композиция $\theta\eta$ получается из множества $\{x_1/t_1\eta, \ldots, x_n/t_n(, y_1/s_1, \ldots, y_n/s_n\}$ удалением пар $x_i/t_i\eta$, где $x_i \neq t_i\eta$ и пар y_i/s_i, где y_i совпадает с одним из x_j.

Пример. Пусть $\theta=\{x/f(y), y/z\}$, $\eta=\{x/a, y/b, z/y\}$. Построим $\theta\eta$.

Для этого возьмем множество {x/f(b),y/y,x/a,y/b,z/y} и выбросим из него пары y/y (потому что заменяемая переменная совпадает с термом), ,x/a,y/b (потому что заменяемая переменная из подстановки η совпадает с заменяемой переменной из подстановки θ). Получим ответ: θη={x/f(b),z/y}.

Подстановка θ называется *более общей*, чем подстановка η, если существует такая подстановка γ, что η=θγ.

Подстановка θ называется *унификатором* простых выражений A и B, если Aθ=Bθ. Про A и B в такой ситуации говорят, что они *унифицируемы*. Унификация используется в Прологе для композиции и декомпозиции структур данных.

Пример. A=p(f(x),z) и B=p(y,a) унифицируемы. Можно взять в качестве их унификатора подстановку {y/f(x),z/a} или подстановку {y/f(a),x/a,z/a}.

Вообще говоря, две формулы могут иметь бесконечно много унификаторов. Унификатор q называют *наиболее общим* (или *простейшим*) *унификатором* простых выражений A и B, если он является более общей подстановкой, чем все другие унификаторы простых выражений A и B.

Пример. В рассмотренном выше примере наиболее общим унификатором является подстановка {y/f(a),z/a}.

Пусть S — конечное множество простых выражений. Определим *множество* d(S) *разногласий (рассогласований)*. Зафиксируем самую левую позицию, на которой не во всех выражениях из S стоит один и тот же символ. Занесем в d(S) подвыражения выражений из S, начинающиеся с этой позиции.

Пример. Пусть S={p(f(x),h(y),a),p(f(x),z,a),p(f(x),h(y),b)}. Множество *рассогласований* для S d(S)={h(y),z}.

Алгоритм унификации

Дадим алгоритм поиска наиболее общего унификатора для конечного множества простых выражений S. В том случае, если это множество не унифицируемо, алгоритм должен обнаруживать эту ситуацию.

Шаг 1. Полагаем k=0, $\delta_0=\varepsilon$.

Шаг 2. Если $S\delta_k$ — одноэлементное множество, останавливаем алгоритм; δ_k — наиболее общий унификатор для S. В противном случае строим множество рассогласований $d(S\delta_k)$ и переходим к третьему шагу.

Шаг 3. Если в $d(S\delta_k)$ существуют переменная x и терм t такие, что x не входит в t, то полагаем что $\delta_{k+1}=\delta_k\{x/t\}$. Увеличиваем на единицу k, переходим ко второму шагу. Иначе останавливаем алгоритм, множество S не унифицируемо.

Обратите внимание, что алгоритм унификации заканчивает свою работу за конечное число шагов для любого конечного множества простых выражений, потому что на каждом проходе мы уменьшаем количество переменных. Так как множество простых выражений было конечным, то и множество различных переменных в нем конечно, и, значит, через число шагов, не превышающее количества различных переменных, алгоритм завершится.

Утверждение о том, что для любого унифицируемого конечного множества простых выражений S алгоритм унификации закончит свою работу и выдаст наиболее общий унификатор для S, называется *теоремой унификации*.

Теперь можно перейти к рассмотрению *метода резолюций*.

В чем вообще заключается задача? Мы хотим построить алгоритм, который позволял бы нам автоматически давать ответ на вопрос, может ли быть выведено некоторое заключение из множества имеющихся посылок. Известно, что в общем случае даже для логики первого порядка такой алгоритм невозможен. Как правило, формальные системы, для которых можно построить подобный разрешающий алгоритм, обладают небольшой выразительной силой. К ним, например, относится логика высказываний и логика одноместных предикатов.

Однако Робинсон решил, что правила вывода, используемые компьютером при автоматическом выводе, не обязательно должны совпадать с правилами вывода, используемыми при «человеческом» выводе. В частности, он предложил вместо правила вывода «modus ponens», которое утверждает, что из A и A→B выводится B, использовать его обобщение, *правило резолюции*, которое сложнее понимается человеком, но эффективно реализуется на компьютере. Давайте попробуем разобраться с этим правилом.

Правило резолюции для логики высказываний можно сформулировать следующим образом.

Если для двух дизъюнктов существует атомная формула, которая в один дизъюнкт входит положительно, а в другой отрицательно, то, вычеркнув соответственно из одного дизъюнкта положительное вхождение атомной формулы, а из другого — отрицательное, и объединив эти дизъюнкты, мы получим дизъюнкт, называемый *резольвентой*. Исходные дизъюнкты в таком случае называются *родительскими* или резольвируемыми, а вычеркнутые формулы — *контрарными литералами*. Другими словами, резольвента — это дизъюнкт, полученный из объединения родительских дизъюнктов вычеркиванием контрарных литералов.

Графически это правило можно изобразить так:

$$\frac{A \vee B,\ P \vee \neg P}{A \vee B}$$

Здесь A∨P и B∨¬P — родительские дизъюнкты, P и ¬P — контрарные литералы, A∨B — резольвента.

Если родительские дизъюнкты состояли только из контрарных литералов, то резольвентой будет пустой дизъюнкт.

Пример. Правило вывода «modus ponens» получается из правила резолюции, если взять в качестве родительских дизъюнктов C_1=A, C_2=¬A∨B(≡A→B). Контрарными литералами в применении этого правила будут A и ¬A, резольвентой — формула B.

Сформулируем правило резолюции для логики первого порядка.

Пусть имеется два дизъюнкта C_1 и C_2, у которых нет общих переменных, L_1 — литерал, входящий в дизъюнкт C_1, L_2 — литерал, входящий в дизъюнкт C_2. Если литералы имеют наибольший общий унификатор θ, то дизъюнкт $(C_1\theta - L_1\theta) \cup (C_2\theta - L_2\theta)$ называется *резольвентой* дизъюнктов C_1 и C_2. Литералы L_1 и L_2 называются контрарными литералами. То же правило записывается в графическом виде как

$$\frac{A \vee P_2,\ B \vee \neg P_2}{(A \vee B)\theta}$$

Здесь P_1 и P_2 — контрарные литералы, $(A \vee B)\theta$ — резольвента, полученная из дизъюнкта (A∨B) применением унификатора θ, $A \vee P_1$ и $B \vee P_2$ — родительские дизъюнкты, а θ — наибольший общий унификатор P_1 и P_2.

Метод резолюций является обобщением метода «доказательства от противного». Вместо того чтобы пытаться вывести некоторую формулу-гипотезу из имеющегося непротиворечивого множества аксиом, мы добавляем отрицание нашей формулы к множеству аксиом и пытаемся вывести из него противоречие. Если нам удается это сделать, мы приходим к выводу (пользуясь законом исключенного третьего), что исходная формула была выводима из множества аксиом. Опишем более подробно.

Добавим отрицание исходной формулы к множеству посылок, преобразуем каждую из этих формул во множество дизъюнктов, объединим получившиеся множества дизъюнктов и попытаемся вывести из этого множества дизъюнктов противоречие (пустой дизъюнкт □). Для этого будем выбирать из нашего множества дизъюнкты, содержащие унифицируемые контрарные литералы, вычислять их резольвенту по правилу резолюции, добавлять ее к исследуемому множеству дизъюнктов. Этот процесс будем продолжать до тех пор, пока не выведем пустой дизъюнкт.

Возможны, вообще говоря, три случая:

1. Этот процесс никогда не завершается.
2. Среди текущего множества дизъюнктов не окажется таких, к которым можно применить правило резолюции. Это означает, что множество дизъюнктов выполнимо, и, значит, исходная формула не выводима.

3. На очередном шаге получена пустая резольвента. Это означает, что множество дизъюнктов невыполнимо и, следовательно, начальная формула выводима.

Имеет место теорема, утверждающая, что описанный выше процесс обязательно завершится за конечное число шагов, если множество дизъюнктов было невыполнимым.

С другой стороны, мы опираемся на результат, что формула выводима из некоторого множества формул тогда и только тогда, когда описанное множество дизъюнктов невыполнимо. А также на то, что множество дизъюнктов невыполнимо тогда и только тогда, когда из него применением правила резолюции можно вывести пустой дизъюнкт.

В сущности, метод резолюций несовершенен и приводит к «комбинаторному взрыву». Однако некоторые его разновидности (или стратегии) довольно эффективны. Одной из самых удачных стратегий является *линейная* или *SLD-резолюция* для *хорновских дизъюнктов* (Linear resolution with Selection function for Definition clauses), то есть дизъюнктов, содержащих не более одного положительного литерала. Их называют *предложениями* или *клозами*.

Если дизъюнкт состоит только из одного положительного литерала, он называется *фактом*. Дизъюнкт, состоящий только из отрицательных литералов, называется *вопросом* (или *целью* или *запросом*). Если дизъюнкт содержит и позитивный, и негативный литералы, он называется правилом. Правило вывода выглядит примерно следующим образом $\neg A_1 \vee \neg A2 \ldots \neg A_n \vee B$. Это эквивалентно формуле $A_1 \wedge A_2 \ldots \wedge A_n \rightarrow B$, которая на Прологе записывается в виде

```
B:-A1,A2,...,An.
```

Логической программой называется конечное непустое множество хорновских дизъюнктов (фактов и правил).

При выполнении программы к множеству фактов и правил добавляется отрицание вопроса, после чего используется *линейная резолюция*. Ее специфика в том, что правило резолюции применяется не к произвольным дизъюнктам из программы. Берется самый левый литерал цели (подцель) и первый унифицируемый с ним дизъюнкт. К ним применяется правило резолюции. Полученная резольвента добавляется в программу в качестве нового вопроса. И так до тех пор, пока не будет получен пустой дизъюнкт, что будет означать успех, или до тех пор, пока очередную подцель будет невозможно унифицировать ни с одним дизъюнктом программы, что будет означать неудачу.

В последнем случае включается так называемый бэктрекинг — механизм возврата, который осуществляет откат программы к той точке, в которой выбирался унифицирующийся с последней подцелью дизъюнкт.

Для этого точка, где выбирался один из возможных унифицируемых с подцелью дизъюнктов, запоминается в специальном стеке, для последующего возврата к ней и выбора альтернативы в случае неудачи. При откате все переменные, которые были означены в результате унификации после этой точки, опять становятся свободными.

В итоге выполнение программы может завершиться неудачей, если одну из подцелей не удалось унифицировать ни с одним дизъюнктом программы, и может завершиться успешно, если был выведен пустой дизъюнкт, а может и просто зациклиться.

Лекция 3. Основные понятия Пролога

Предложения: факты и правила. Цели внутренние и внешние. Отношения (предикаты). Переменные свободные и связанные. Анонимная переменная. Отсечение. «Зеленые» и «красные» отсечения. Семантические модели Пролога: декларативная и процедурная.

Ключевые слова: предложения, факты, правила, цели, вопросы, отсечение, декларативная модель, процедурная модель.

Данная лекция посвящена базовым понятиям языка Пролог. В этой и следующей лекциях мы будем изучать основы написания программ на Прологе.

Начнем с того, что познакомимся с так называемой *нормальной формой Бэкуса–Наура (БНФ),* разработанной в 1960 Джоном Бэкусом и Питером Науром и используемой для формального описания синтаксиса языков программирования. Впервые БНФ была применена Питером Науром при записи синтаксиса языка Алгол-60.

При описании синтаксиса конструкций используются следующие обозначения:

Символ `::=` читается как «по определению» («это», «есть»). Слева от разделителя располагается объясняемое понятие, справа — конструкция, разъясняющая его. Например,

```
<Имя> ::= <Идентификатор>
```

В угловые скобки заключается часть выражения, которая используется для обозначения синтаксической конструкции языка, в частности объясняемое понятие. В приведенном выше примере это `<Имя>` и `<Идентификатор>`.

Символ `|` означает в нотации БНФ «или», он применяется для разделения различных альтернативных растолкований определяемого понятия.

Пример. Десятичную цифру можно определить следующим образом:

```
<цифра> ::= 0|1|2|3|4|5|6|7|8|9
```

Часть синтаксической конструкции, заключенная в квадратные скобки, является необязательной (может присутствовать или отсутствовать);

Пример. Запись

```
<Целое число> ::= [-]<Положительное целое число>
```

означает, что целое число можно определить через положительное целое число, перед которым может стоять знак минус.

Символ * обозначает, что часть синтаксической конструкции может повторяться произвольное число раз (ноль и более). Заметим, что иногда вместо символа * используют фигурные скобки ({,}).

Пример. Определить положительное целое число в нотации БНФ можно следующим образом:

```
<Положительное целое число> ::= <цифра>[<цифра>]*.
```

То есть положительное целое число состоит из одной или нескольких цифр.

Программа на языке Пролог (ее иногда называют *базой знаний*) состоит из *предложений* (или *утверждений*), каждое предложение заканчивается точкой.

Предложения бывают двух видов: *факты, правила*.

Предложение имеет вид

```
A:-
   B1,... , Bn.
```

A называется *заголовком* или *головой* предложения, а B1,..., Bn — *телом*.

В принципе об этом уже говорилось в предыдущей лекции. Но там мы рассматривали эти понятия в основном с теоретической точки зрения, заходя со стороны математической логики, а сейчас наш подход будет больше практическим, со стороны программирования.

Факт констатирует, что между объектами выполнено некоторое отношение. Он состоит только из заголовка. Можно считать, что факт — это предложение, у которого тело пустое.

Например, известный нам факт, что Наташа является мамой Даши, может быть записан в виде:

```
мама(Наташа, Даша).
```

Факт представляет собой безусловно истинное утверждение.

Напомню, что в математической логике, с которой мы познакомились в предыдущей лекции, отношения принято называть *предикатами*.

Если воспользоваться нормальной формой Бэкуса–Науэра, то предикат можно определить следующим образом:

```
<Предикат>::=<Имя>     <Имя>(<аргумент>[,<аргумент>]*),
```

т.е. предикат состоит либо только из имени, либо из имени и следующей за ним последовательности аргументов, заключенной в скобки.

Аргументом или параметром предиката может быть константа, переменная или составной объект. Число аргументов предиката называется его *арностью* или *местностью*. Про переменные мы поговорим чуть-чуть позже, а подробное рассмотрение констант отложим до пятой лекции. Пока отметим, что константа получает свое значение в разделе описания констант, а переменная означивается в процессе работы программы.

В Турбо Прологе имя предиката должно состоять из последовательности латинских букв, цифр, знаков подчеркивания и начинаться с буквы или знака подчеркивания. В других версиях Пролога имя предиката может содержать символы не только из английского алфавита, но и из национального – например, из русского.

Соответственно, приведенный выше пример факта можно записать в Турбо Прологе, например, так:

```
mother("Наташа", "Даша").
```

Некоторые предикаты уже известны системе, они называются *стандартными* или *встроенными*.

В Турбо Прологе предложения с одним и тем же предикатом в заголовке должны идти одно за другим. Такая совокупность предложений называется *процедурой*.

В приведенном выше примере про то, что Наташа является мамой Даши, `мама` — это имя двухаргументного предиката, у которого строковая константа "`Наташа`" является первым аргументом, а строковая константа "`Даша`" — вторым.

Правило — это предложение, истинность которого зависит от истинности одного или нескольких предложений. Обычно правило содержит несколько хвостовых целей, которые должны быть истинными для того, чтобы правило было истинным.

В нотации БНФ правило будет иметь вид:

```
<Правило>::=<предикат>:-<предикат>[,<предикат>]*.
```

Пример. Известно, что бабушка человека — это мама его мамы или мама его папы.

Соответствующие правила будут иметь вид:

```
бабушка(X,Y):-
            мама(X,Z),мама(Z,Y).
бабушка(X,Y):-
```

```
мама(X,Z),папа(Z,Y).
```

Символ « :-» означает «если», и вместо него можно писать `if`.

Символ «,» — это логическая связка «и» или конъюнкция, вместо него можно писать `and`.

Первое правило сообщает, что `X` является бабушкой `Y`, если существует такой `Z`, что `X` является мамой `Z`, а `Z` — мамой `Y`. Второе правило сообщает, что `X` является бабушкой `Y`, если существует такой `Z`, что `X` является мамой `Z`, а `Z` — папой `Y`.

В данном примере `X`, `Y` и `Z` — это переменные.

Имя переменной в Турбо Прологе может состоять из букв латинского алфавита, цифр, знаков подчеркивания и должно начинаться с прописной буквы или знака подчеркивания. При этом переменные в теле правила неявно связаны квантором всеобщности. Переменная в Прологе, в отличие от алгоритмических языков программирования, обозначает объект, а не некоторую область памяти. Пролог не поддерживает механизм деструктивного присваивания, позволяющий изменять значение инициализированной переменной, как императивные языки.

Переменные могут быть *свободными* или *связанными*.

Свободная переменная — это переменная, которая еще не получила значения. Она не равняется ни нулю, ни пробелу; у нее вообще нет никакого значения. Такие переменные еще называют *неконкретизированными*.

Переменная, которая получила какое-то значение и оказалась связанной с определенным объектом, называется *связанной*. Если переменная была конкретизирована каким-то значением и ей сопоставлен некоторый объект, то эта переменная уже не может быть изменена.

Областью действия переменной в Прологе является одно предложение. В разных предложениях может использоваться одно имя переменной для обозначения разных объектов. Исключением из правила определения области действия является *анонимная переменная*, которая обозначается символом подчеркивания «_». Анонимная переменная применяется в случае, когда значение переменной не важно. Каждая анонимная переменная — это отдельный объект.

Третьим специфическим видом предложений Пролога можно считать *вопросы*.

Вопрос состоит только из тела и может быть выражен с помощью БНФ в виде:

<Вопрос>::=<Предикат>[,<Предикат>]*

Вопросы используют для выяснения выполнимости некоторого отношения между описанными в программе объектами. Система рассмат-

ривает вопрос как цель, к которой надо стремиться. Ответ на вопрос может оказаться положительным или отрицательным, в зависимости от того, может ли быть достигнута соответствующая цель.

Программа на Прологе может содержать вопрос в программе (так называемая *внутренняя цель*). Если программа содержит внутреннюю цель, то после запуска программы на выполнение система проверяет достижимость заданной цели.

Если внутренней цели в программе нет, то после запуска программы система выдает приглашение вводить вопросы в диалоговом режиме (*внешняя цель*). Программа, компилируемая в исполняемый файл, обязательно должна иметь внутреннюю цель.

Если цель достигнута, система отвечает, что у нее есть информация, позволяющая сделать вывод об истинности вопроса (*«Yes»*). При этом если в вопросе содержатся переменные, то система либо выдает их значения, приводящие к решению, если решение существует, либо сообщает, что решений нет (*«No solution»*). Если достичь цели не удалось, система ответит, что у нее нет положительного ответа (*«No»*).

Следует заметить, что ответ *«No»* на вопрос не всегда означает, что отношение, о котором был задан вопрос, не выполняется. Система может дать такой ответ и в том случае, когда у нее просто нет информации, позволяющей положительно ответить на вопрос.

Можно сказать, что утверждение — это правило, а факт или вопрос — это его частный случай.

Рассмотрим несколько примеров. Пусть в программе заданы следующие отношения:

```
мама("Наташа","Даша").
мама("Даша","Маша").
```

Можно спросить у системы, является ли `Наташа` мамой `Даши`. Этот вопрос можно ввести в виде:

```
мама("Наташа","Даша").
```

Найдя соответствующий факт в программе, система ответит *«Yes»* (то есть *«Да»*). Если мы спросим:

```
мама("Наташа","Маша").
```

то получим ответ *«No»* (то есть *«Нет»*). Можно также попросить вывести имя мамы `Даши`:

```
мама(X,Даша).
```

Система сопоставит вопрос с первым фактом, конкретизирует переменную X значением “Наташа” и выдаст ответ:

```
X=Наташа
1 Solution
```

Вопрос об имени дочери Наташи записывается в виде:

```
мама(Наташа,X).
```

Соответствующим ответом будет:

```
X=Даша
1 Solution
```

Можно попросить систему найти имена всех известных ей мам и дочек, задав вопрос:

```
мама(X,Y).
```

Система последовательно будет пытаться согласовывать вопрос с имеющимися в программе предложениями от первого до последнего. В случае успешной унификации соответствующих термов переменная X будет означена именем матери, а переменная Y — именем ее дочери.

В итоге получим ответ:

```
X=Наташа Y=Даша
X=Даша Y=Маша
2 solutions
```

Если надо получить только имена всех мам, можно воспользоваться анонимной переменной и записать вопрос:

```
мама(X,_).
```

Получим ответ:

```
X=Наташа
X=Даша
2 solutions
```

И, наконец, если надо получить ответ на вопрос: есть ли информация о людях, находящихся в отношении «мама — дочка», то его можно сформулировать в виде:

```
мама(_,_),
```

В данном случае нам не важны конкретные имена, а интересует, есть ли в нашей базе знаний хотя бы один соответствующий факт. Ответом в данном случае будет просто *«Yes»*. Система сообщит о том, что у нее есть информация об объектах, связанных отношением «мама».

Введем в нашу программу правило, определяющее отношение «бабушка — внучка», в терминах «быть мамой»:

```
бабушка(X,Y):-
            мама(X,Z),
            мама(Z,Y).
```

По сути дела здесь записано, что один человек является бабушкой другого, если это он является мамой его мамы. Конечно, для полноты картины не помешает записать еще и второе правило, которое говорит, что бабушка — это мама папы, если в программу добавить факты еще и про пап.

Заметим, что в нашей программе нет ни одного факта, связанного с отношением `бабушка`. Тем ни менее, система оказывается способна найти ответы на вопросы о бабушках, пользуясь введенными фактами и правилом. Например, если нас интересует, чьей бабушкой является `Наташа`, то мы можем записать этот вопрос следующим образом:

```
бабушка("Наташа",X).
```

Для того чтобы найти ответ на вопрос, система просмотрит нашу базу сверху вниз, пытаясь найти предложение, в заголовке которого стоит предикат `бабушка`. Найдя такое предложение (это предложение `бабушка(X,Y):-мама(X,Z),мама(Z,Y)`), система конкретизирует переменную из заголовка предложения `X` именем `"Наташа"`, переменную `Y` с переменной `X` из вопроса, после чего попытается достигнуть цели: `мама("Наташа",Z)` и `мама(Z,Y)`. Для этого она просматривает базу знаний в поиске предложения, заголовок которого можно сопоставить с предикатом `мама("Наташа",Z)`.

Это можно сделать, конкретизировав переменную `Z` именем `"Даша"`. Затем система ищет предложение, в заголовке которого стоит предикат мама с первым аргументом `"Даша"` и каким-то именем в качестве второго аргумента. Подходящим предложением оказывается факт `ма-`

ма("Даша","Маша"). Система установила, что обе подцели мама("Наташа",Z) и мама(Z,Y) достижимы при Z="Даша",Y="Маша". Она выдает ответ:

```
X=Маша
```

Напомним, что наша переменная X из вопроса была связана с переменной Y из правила. После этого, если есть такая возможность, система пытается найти другие решения, удовлетворяющие вопросу. Однако в данном случае других решений нет.

Вообще говоря, цель может быть *согласована*, если она сопоставляется с заголовком какого-либо предложения. Если сопоставление происходит с фактом, то цель согласуется немедленно. Если же сопоставление происходит с заголовком правила, то цель согласуется только тогда, когда будет согласована каждая подцель в теле этого правила, после вызова ее в качестве цели. Подцели вызываются слева направо. Поиск подходящего для сопоставления предложения ведется с самого начала базы. Если подцель не допускает сопоставления, то система совершает возврат для попытки повторного согласования подцели. При попытке повторного согласования система возобновляет просмотр базы с предложения, непосредственно следующего за тем, которое обеспечивало согласование цели ранее.

В программе на Прологе важен порядок предложений внутри процедуры, а также порядок хвостовых целей в теле предложений. От порядка предложений зависит порядок поиска решений и порядок, в котором будут находиться ответы на вопросы. Порядок целей влияет на количество проверок, выполняемых программой при решении.

Пример. Давайте создадим предикат, который будет находить максимум из двух чисел. У предиката будет три аргумента. Первые два аргумента — входные для исходных чисел, в третий выходной аргумент будет помещен максимум из первых двух аргументов.

Предикат будет довольно простым. Мы запишем, что в случае, если первое число больше второго, максимальным будет первое число, в случае, если первое число меньше, максимумом будет второе число. Надо также не забыть про ситуацию, когда числа равны, в этом случае максимумом будет любое из них.

Решение можно записать в следующем виде:

```
max(X,Y,X):-
          X>Y. /* если первое число больше второго,
                  то первое число — максимум */
max(X,Y,Y):-
```

```
            X<Y. /* если первое число меньше второго,
                    то второе число — максимум */
max(X,Y,Y):
            X=Y. /* если первое число равно второму,
                    возьмем в качестве максимума
                    второе число */
```

Последнее предложение можно объединить со вторым или третьим в одно предложение. Тогда процедура будет состоять не из трех предложений, а всего из двух:

```
max(X,Y,X):-
            X>Y. /* если первое число больше второго,
                    то первое число — максимум */
max(X,Y,Y):-
            X<=Y. /* если первое число меньше второго или
                     равно второму, возьмем в качестве мак-
                     симума второе число */
```

Однако полученная процедура еще далека от совершенства. С одной стороны, в случае, когда первое проверяемое условие `(X>Y)` не выполнено, будет проверяться второе условие `(X<=Y)`, хотя понятно, что если не выполнено `X>Y`, значит `X<=Y`. С другой стороны, в случае, если первое условие имело место и первое число оказалось больше второго, Пролог-система свяжет третий аргумент предиката max с первым аргументом, после чего попытается сопоставить второе предложение. Хотя нам очевидно, что после того, как максимум определен, не нужно больше ничего делать. Других вариантов в данной ситуации просто не может быть. И, значит, проверка второго условия избыточна.

В данной ситуации нам пригодится встроенный предикат, который по-английски называется *cut*, по-русски — *отсечение*, а в программе на Прологе он обозначается восклицательным знаком «!». Этот предикат предназначен для ограничения пространства поиска с целью повышения эффективности работы программ. Он всегда завершается успешно. После того, как до него дошла очередь, он устанавливает «забор», который не дает «откатиться назад», чтобы выбрать альтернативные решения для уже «сработавших» подцелей. То есть для тех, которые расположены левее отсечения. На цели, расположенные правее, отсечение не влияет. Кроме того, отсечение отбрасывает все предложения процедуры, расположенные после предложения, в котором находится отсечение.

С использованием отсечения наше решение будет еще короче:

```
max2(X,Y,X):-
        X>Y,!. /*  если первое число больше второго,
                   то первое число – максимум */
max2(_,Y,Y). /* в противном случае максимумом будет
                второе число */
```

В случае, если сработает отсечение, а это возможно, только если окажется истинным условие `X>Y`, Пролог-система не будет рассматривать альтернативное второе предложение. Второе предложение «сработает» только в случае, если условие оказалось ложным. В этой ситуации в третий аргумент попадет то же значение, которое находилось во втором аргументе. Обратите внимание, что в этом случае нам уже не важно, чему равнялся первый аргумент, и его можно заменить анонимной переменной.

Все случаи применения отсечения принято разделять на «зеленые» и «красные». *Зелеными* называются те из них, при отбрасывании которых программа продолжает выдавать те же решения, что и при наличии отсечения. Если же при устранении отсечений программа начинает выдавать неправильные решения, то такие отсечения называются *красными*.

Пример «красного» отсечения имеется в реализации предиката *max2* (если убрать отсечение, предикат будет выдавать в качестве максимума второе число, даже если оно меньше первого). Пример «зеленого» отсечения можно получить, если в запись предиката *max* добавить отсечения (при их наличии предикат будет выдавать те же решения, что и без них).

В принципе, с помощью отсечения в Прологе можно смоделировать такую конструкцию императивных языков, как ветвление.

Процедура

```
S:-
   <условие>,!,P.
S :-
   P2.
```

будет соответствовать оператору `if <условие> then P else P2`, то есть если условие имеет место, то выполнить `P`, иначе выполнить `P2`. Например, в случае с максимумом, можно расшифровать нашу процедуру как «если `X>Y`, то `M=X`, иначе `M=Y`».

Пример. Теперь напишем предикат, который будет находить максимум не из двух чисел, а из трех. У него будет уже четыре параметра. Первые три — входные для сравниваемых чисел, а четвертый — выходной параметр для их максимума.

Подходов к решению этой задачи может быть несколько.

Первое, что приходит в голову, это решить задачу по аналогии с нахождением максимума из двух чисел. Вариант без отсечения будет выглядеть так:

```
max3a(X,Y,Z,X):-
              X>=Y,X>=Z.
           /* если первое число больше или равно второму
              и третьего, то первое число — максимум */
max3a(X,Y,Z,Y):-
              Y>=X,Y>=Z.
           /* если второе число больше или равно первому
              и третьему, то второе число является
              максимумом */
max3a(X,Y,Z,Z):-
              Z>=X,Z>=Y.
           /* если третье число больше или равно первому
              и второму, то максимум — это третье число */
```

Недостаток этой программы, кроме ее длины, еще и в том, что если какие-то из исходных чисел окажутся равными, мы получим несколько одинаковых решений. Например, если все три числа совпадают, то каждое из трех правил будет истинным и, соответственно, мы получим три одинаковых, хотя и правильных ответа.

Применение отсечения позволит существенно сократить решение:

```
max3b(X,Y,Z,X):-
              X>Y,X>Z,!.
           /* если первое число больше второго и третьего,
              то первое число — максимум */
max3b(_,Y,Z,Y):-
              Y>=Z,!.
           /* иначе, если второе число больше третьего,
              то второе число является максимумом */
max3b(_,_,Z,Z). /* иначе максимум — это третье число */
```

Число сравнений значительно сократилось за счет того, что отсечение в первом правиле гарантирует нам, что на второе правило мы попадем только в том случае, если первое число не больше второго и третьего. В этой ситуации максимум следует искать среди второго и третьего чисел. Если ни первое, ни второе число не оказались больше третьего, значит, в качестве максимума можно взять как раз третье число, уже ничего не проверяя. Обратите внимание на то, что во втором правиле нам было не важно, чему

равно первое число, а в третьем предложении участвовало только третье число. Не участвующие параметры заменены анонимными переменными.

И, наконец, самое короткое решение можно получить, если воспользоваться уже имеющимся предикатом `max2`. Решение будет состоять всего из одного предложения.

```
max3(X,Y,Z,M):-
        max2(X,Y,XY), /* XY — максимум из X и Y */
        max2(XY,Z,M). /* M — максимум из XY и Z */
```

Мы записали, что для того, чтобы найти максимум из трех чисел, нужно найти максимум из первых двух чисел, после чего сравнить его с третьим числом.

Семантические модели Пролога

В Прологе обычно применяются две семантические модели: **декларативная** и **процедурная**. Семантические модели предназначены для объяснения смысла программы.

В *декларативной* модели рассматриваются отношения, определенные в программе. Для этой модели порядок следования предложений в программе и условий в правиле не важен.

Процедурная модель рассматривает правила как последовательность шагов, которые необходимо успешно выполнить для того, чтобы соблюдалось отношение, приведенное в заголовке правила.

Множество предложений, имеющих в заголовке предикат с одним и тем же именем и одинаковым количеством аргументов, трактуются как процедура. Для процедурной модели важен порядок, в котором записаны предложения и условия в предложениях.

При написании программы на Прологе кажется логичным в первую очередь рассматривать декларативную семантику, однако и о процедурной не стоит забывать, особенно в том случае, когда программа не работает или работает не совсем так, как предполагалось.

Следует заметить, что в некоторых случаях использование отсечения может привести к изменению декларативного смысла.

Лекция 4. Рекурсия

Рекурсия. Достоинства и недостатки рекурсии. Хвостовая рекурсия. Организация циклов на основе рекурсии. Вычисление факториала.

Ключевые слова: рекурсия, хвостовая рекурсия, правая рекурсия, левая рекурсия.

В отличие от традиционных языков программирования, в которых основным средством организации повторяющихся действий являются циклы, в Прологе для этого используются процедура поиска с возвратом (откат) и рекурсия. Откат дает возможность получить много решений в одном вопросе к программе, а рекурсия позволяет использовать в процессе определения предиката его самого. При изучении рекурсии частенько вспоминается случай с бароном Мюнхгаузеном, который сам себя за волосы вытаскивал из болота. Про откат мы подробнее поговорим в шестой лекции, а в этой займемся изучением рекурсии. Заметим, что рекурсию используют не только в Прологе, но и в обычных императивных языках программирования. Но для Пролога, в отличие от императивных языков, рекурсия является основным приемом программирования. Более того, Пролог позволяет определять рекурсивные структуры данных. Работе с ними будет посвящено несколько лекций нашего курса.

Начнем изучение рекурсии в Прологе с классического примера. Предположим, что в базе знаний есть набор фактов, описывающий родственные связи людей через отношение «быть родителем». Предикат *родитель* имеет два аргумента. В качестве его первого аргумента указывается имя родителя, в качестве второго — имя ребенка. Мы хотим создать отношение «быть предком», используя предикат *родитель*.

Для того чтобы один человек был предком другого человека, нужно, чтобы он либо был его родителем, либо являлся родителем другого его предка.

Запишем эту идею:

```
предок(Предок,Потомок):-
                    родитель(Предок,Потомок).
                          /* предком является родитель */
предок(Предок,Потомок):-
                    родитель(Предок,Человек),
                    предок(Человек,Потомок).
                    /* предком является родитель предка */
```

Отношение `предок` является *транзитивным замыканием* отношения `родитель`, то есть это наименьшее отношение, включающее отношение

родитель и обладающее свойством транзитивности. Напомним, что отношение называется *транзитивным*, если для любых пар *(A,B)* и *(B,C)*, находящихся в этом отношении, пара *(A,C)* также находится в этом отношении. Очевидно, что отношение предок содержит отношение родитель. Это следует из первого предложения, в котором записано, что всякий родитель является предком. Второе предложение дает транзитивность.

По аналогии с математической индукцией, на которую рекурсия немного похожа, любая рекурсивная процедура должна включать в себя базис и шаг рекурсии.

Базис рекурсии — это предложение, определяющее некую начальную ситуацию или ситуацию в момент прекращения. Как правило, в этом предложении записывается некий простейший случай, при котором ответ получается сразу даже без использования рекурсии. Так, в приведенной выше процедуре, описывающей предикат предок, базисом рекурсии является первое правило, в котором определено, что ближайшими предками человека являются его родители. Это предложение часто содержит условие, при выполнении которого происходит выход из рекурсии или отсечение.

Шаг рекурсии — это правило, в теле которого обязательно содержится, в качестве подцели, вызов определяемого предиката. Если мы хотим избежать зацикливания, определяемый предикат должен вызываться не от тех же параметров, которые указаны в заголовке правила. Параметры должны изменяться на каждом шаге так, чтобы в итоге либо сработал базис рекурсии, либо условие выхода из рекурсии, размещенное в самом правиле. В общем виде правило, реализующее шаг рекурсии, будет выглядеть так:

```
<имя определяемого предиката>:-
          [<подцели>],
          [<условие выхода из рекурсии>],
          [<подцели>],
          <имя определяемого предиката>,
          [<подцели>].
```

В некоторых ситуациях предложений, реализующих базис рекурсии, и предложений, описывающих шаг рекурсии, может быть несколько. Как правило, это бывает в сложных случаях, например, когда выполняемые в процессе реализации шага рекурсии действия зависят от выполнения или невыполнения какого-либо условия. Такие задачи встретятся нам в последующих лекциях, когда речь пойдет об обработке рекурсивных структур данных. В этой же лекции мы будем иметь дело в основном с простыми случаями рекурсии, когда рекурсивная процедура имеет один базис и один шаг рекурсии.

Пример. Создадим предикат, который будет вычислять по натуральному числу его факториал. Эта задача допускает рекурсивное решение на многих языках программирования, а также имеет рекурсивное математическое описание:

```
1!=1 /* факториал единицы равен единице */
N!=(N-1)!*N  /* для того, чтобы вычислить факториал
                некоторого числа, нужно вычислить
                факториал числа на единицу меньшего
                и умножить его на исходное число */
```

Попробуем записать реализацию предиката, эквивалентную математическому определению предиката:

```
fact(1,1). /* факториал единицы равен единице */
fact(N,F):-
          N1=N-1,
          fact(N1,F1), /* F1 равен факториалу числа
                          на единицу меньшего исходного
                          числа */
          F=F1*N. /* факториал исходного числа равен
                     произведению F1 на само число */
```

К сожалению, при попытке вычислить факториал произвольного натурального числа с помощью описанного выше предиката `fact` произойдет переполнение стека *(«Stack overflow»)*. Попробуем разобраться, в чем причина. Рассмотрим, например, что будет происходить, если мы попытаемся вычислить факториал трех.

Соответствующий вопрос можно записать следующим образом:

```
fact(3,X).
```

Пролог-система попытается унифицировать цель с заголовком первого предложения `(fact(1,1))`. Ей это не удастся, поскольку число три не равно единице. При унификации цели с заголовком второго предложения `(fact(N,F))` переменная `N` конкретизируется числом три, а переменная `X` связывается с переменной `F`. После этого происходит попытка выполнить подцели, расположенные в теле правила слева направо. Сначала переменная `N1` означивается числом, на единицу меньшим, чем значение переменной `N`, то есть двойкой. Срабатывание следующей подцели `(fact(N1,F1))` приводит к рекурсивному вызову предиката, вычисляющего факториал, со значением переменной `N1`, равным двум.

Так же, как и в случае, когда первый аргумент был равен трем, унификации с головой первого предложения не происходит (единица не равна двум). Сопоставление с головой второго правила происходит успешно. Дальше все происходит почти так же, как для значения переменной `N`, равного трем. Вычисляется новое значение `N1`, равное двум без единицы, то есть единице. Пролог снова пытается вычислить подцель `fact(N1,F1)` (правда, со значением переменной `N1`, равным единице).

На этот раз происходит сопоставление цели `(fact(1,F1))` с заголовком первого предложения; при этом переменная `F1` конкретизируется единицей. Пролог-системе наконец-то удалось вычислить вторую подцель второго правила, и она переходит к вычислению третьей подцели `(F=F1*N)`. Переменная `N` была равна двум, переменная `F1` — единице, их произведение двум и, значит, переменная `F` конкретизируется двойкой.

Начинается обратный ход рекурсии. После того, как был вычислен факториал двойки, Пролог-система готова вычислить факториал тройки. Для этого нужно умножить факториал двух на три. Переменная `F` будет конкретизирована числом шесть. Мы получили ответ на вопрос о факториале трех.

Однако вычисления на этом не заканчиваются. Пролог-система обнаруживает, что цель `fact(1,F1)` может быть сопоставлена не только с заголовком первого предложения, но и с заголовком правила `(fact(N,F))`. Переменная `N` конкретизируется единицей, а переменная `F1` связывается с переменной `F`. После этого переменная `N1` означивается числом, на единицу меньшим, чем значение переменной `N`, то есть нулем. Пролог-система пытается вычислить цель `fact(0,F1)`. С заголовком первого предложения `(fact(1,1))` сопоставить эту цель не удается, поскольку ноль не равен единице. Зато с заголовком второго предложения `(fact(N,F))` цель успешно унифицируется. Переменная `N1` становится равна минус единице. После этого делается попытка вычислить цель `fact(-1,F1)`... . Потом `fact(-2,F1)`, `fact(-3,F1)`, `fact(-4,F1)`, `fact(-5,F1)`... .

Этот процесс будет продолжаться до тех пор, пока не будет исчерпана часть оперативной памяти, отведенная под стек. После этого выполнение программы остановится с сообщением о том, что стек переполнен.

Почему так получилось? Что мы сделали неправильно? Причина в том, что в исходном определении факториала, которое мы использовали, предполагалось, что правило работает только для натуральных чисел, то есть для положительных целых чисел. У нас же в программе произошел выход в отрицательную часть целых чисел, что не было предусмотрено формулой, на которой была основана наша процедура.

Как можно исправить ошибку? У нас есть два варианта корректировки процедуры.

Можно проверить, что число, для которого применяется правило, больше единицы. Для единицы останется факт, утверждающий, что факториалом единицы будет единица. Выглядеть этот вариант будет следующим образом:

```
fact(1,1). /* факториал единицы равен единице */
fact(N,F):-
          N>1, /* убедимся, что число больше единицы */
          N1=N-1,
          fact(N1,F1), /* F1 равен факториалу числа,
                          на единицу меньшего исходного
                          числа */
          F=F1*N. /* факториал исходного числа равен
                     произведению F1 на само число */
```

В этом случае, хотя и произойдет повторное согласование цели `fact(1,F1)` с заголовком правила, и переменная `N` будет конкретизирована единицей, а переменная `F` связана с переменной `F1`, первая подцель правила `(N>1)` будет ложной. На этом процесс оборвется. Попытки вычислять факториал на неположительных числах не произойдет, процедура будет работать именно так, как нам хотелось.

Второй вариант решения проблемы — добавить в первое предложение процедуры отсечение. Напомним, что вызов отсечения приводит к тому, что предложения процедуры, расположенные ниже той, из которой оно было вызвано, не рассматриваются. И, соответственно, после того, как какая-то цель будет согласована с заголовком первого предложения, сработает отсечение, и попытка унифицировать цель с заголовком второго предложения не будет предпринята. Процедура в этом случае будет выглядеть так:

```
fact(1,1):-!. /* условие останова рекурсии */
fact(N,F):-
          N1=N-1,
          fact(N1,F1), /* F1 равен факториалу числа,
                          на единицу меньшего исходного
                          числа */
          F=F1*N. /* факториал исходного числа равен
                     произведению F1 на само число */
```

Конечно, с одной стороны, метод рекурсии имеет свои преимущества перед методом итерации, который используется в императивных языках программирования намного чаще. Рекурсивные алгоритмы, как

правило, намного проще с логической точки зрения, чем итерационные. Некоторые алгоритмы удобно записывать именно рекурсивно.

С другой стороны, рекурсия имеет большой недостаток: ей, вообще говоря, может не хватать для работы стека. При каждом рекурсивном вызове предиката в специальном стековом фрейме запоминаются все промежуточные переменные, которые могут понадобиться. Максимальный размер стека при работе под управлением операционной системы MS DOS – всего 64 Кб. Этого достаточно для размещения около трех-четырех тысяч стековых фреймов (в зависимости от количества и размера промежуточных переменных). При больших входных значениях стека может не хватить.

Есть, правда, один вариант рекурсии, который использует практически столько же оперативной памяти, сколько итерация в императивных языках программирования. Это так называемая *хвостовая* или *правая рекурсия*. Для ее осуществления рекурсивный вызов определяемого предиката должен быть последней подцелью в теле рекурсивного правила и к моменту рекурсивного вызова не должно остаться точек возврата (непроверенных альтернатив). То есть у подцелей, расположенных левее рекурсивного вызова определяемого предиката, не должно оставаться каких-то непроверенных вариантов и у процедуры не должно быть предложений, расположенных ниже рекурсивного правила. Турбо Пролог, на который мы ориентируемся в нашем курсе, распознает хвостовую рекурсию и устраняет связанные с ней дополнительные расходы. Этот процесс называется *оптимизацией хвостовой рекурсии* или *оптимизацией последнего вызова*.

Пример. Попробуем реализовать вычисление факториала с использованием хвостовой рекурсии. Для этого понадобится добавить два дополнительных параметра, которые будут использоваться нами для хранения промежуточных результатов. Третий параметр нужен для хранения текущего натурального числа, для которого вычисляется факториал, четвертый параметр — для факториала числа, хранящегося в третьем параметре.

Запускать вычисление факториала мы будем при первом параметре, равном числу, для которого нужно вычислить факториал. Третий и четвертый аргументы будут равны единице. Во второй аргумент по завершении рекурсивных вычислений должен быть помещен факториал числа, находящегося в первом параметре. На каждом шаге будем увеличивать третий аргумент на единицу, а второй аргумент умножать на новое значение третьего аргумента. Рекурсию нужно будет остановить, когда третий аргумент сравняется с первым, при этом в четвертом аргументе будет накоплен искомый факториал, который можно поместить в качестве ответа во второй аргумент.

Вся процедура будет выглядеть следующим образом:

```
fact2(N,F,N,F):-!. /* останавливаем рекурсию, когда третий
                      аргумент равен первому*/
fact2(N,F,N1,F1):-
             N2=N1+1, /* N2 — следующее натуральное число
                         после числа N1 */
             F2=F1*N2, /* F2 — факториал N2 */
             fact2(N,F,N2,F2).
                /* рекурсивный вызов с новым натуральным
                   числом N2 и соответствующим ему
                   посчитанным факториалом F2 */
```

Остановить рекурсию можно, воспользовавшись отсечением в базисе рекурсии, как это было сделано выше, или добавив в начало второго предложения сравнение `N1` с `N`.

Если мы решим, что вызывать предикат с четырьмя аргументами неудобно, можно ввести дополнительный двухаргументный предикат, который будет запускать исходный предикат:

```
factM(N,F):-
        fact2(N,F,1,1). /* вызываем предикат с уже
                           заданными начальными
                           значениями */
```

Пример. В предыдущей лекции мы записали аналог императивного ветвления, воспользовавшись отсечением. Теперь напишем, используя рекурсию и отсечение, реализацию цикла с предусловием. Обычно этот цикл выглядит примерно так: `while <условие> do P`. Это соответствует текстовому описанию «пока имеет место `<условие>`, выполнять `P`». На Прологе подобную конструкцию можно записать следующим образом:

```
w:-
   <условие>,p,w.
w:-!.
```

Пример. Еще одна классическая задача, имеющая рекурсивное решение, связна с вычислением так называемых *чисел Фибоначчи*. Числа Фибоначчи можно определить так: первое и второе числа равны единице, а каждое последующее число является суммой двух предыдущих. Соответственно, третье число Фибоначчи будет равно двум, четвертое равно трем (сумма второго числа (один) и третьего числа (два)), пятое — пяти (сумма третьего и четвертого чисел, то есть двух и трех), шестое — восьми (сумма четвертого и пятого, трех и пяти) и т.д.

Базисов рекурсии в данном случае два. Первый будет утверждать, что первое число Фибоначчи равно единице. Второй базис — аналогичное утверждение про второе число Фибоначчи. Шаг рекурсии также будет необычным, поскольку будет опираться при вычислении следующего числа Фибоначчи не только на предшествующее ему число, но и на предшествующее предыдущему числу. В нем будет сформулировано, что для вычисления числа Фибоначчи с номером *N* сначала нужно вычислить и сложить числа Фибоначчи с номерами *N–1* и *N–2*.

Записать эти рассуждения можно так:

```
fib(1,1):-!. /* первое число Фибоначчи равно единице */
fib(2,1):-!. /* второе число Фибоначчи равно единице */
fib(N,F) :-
        N1=N-1, fib(N1,F1), /* F1 это N-1-е число
                               Фибоначчи */
        N2=N-2, fib(N2,F2), /* F2 это N-2-е число
                               Фибоначчи */
        F=F1+F2. /* N-е число Фибоначчи равно сумме
                    N-1-го и N-2-го чисел Фибоначчи */
```

Обратите внимание на отсечение в первых двух предложениях. Оно служит для остановки рекурсии, чтобы при прямом ходе рекурсии не произошло выхода из области натуральных чисел (номеров чисел Фибоначчи) в область отрицательных чисел, как это происходило у нас в первой версии предиката, вычисляющего факториал.

Вместо этих двух отсечений от зацикливания можно избавиться путем добавления в начало правила, реализующего шаг рекурсии, проверки значения, находящегося в первом параметре предиката `(N>2)`. Это условие в явном виде указывает, что рекурсивное правило применяется для вычисления чисел Фибоначчи, начиная с третьего.

Но надо сказать, что хотя наше решение получилось ясным и прозрачным, довольно точно соответствующим определению чисел Фибоначчи, оно, тем не менее, весьма неэффективное. При вычислении `N-1`-го числа Фибоначчи `F1` вычисляются все предыдущие числа Фибоначчи, в частности, `N-2`-е число Фибоначчи `F2`. После этого заново начинает вычисляться `N-2`-е число Фибоначчи, которое уже было вычислено. Мало того, опять вычисляются все предыдущие числа Фибоначчи. Получается, что для вычисления числа Фибоначчи используется количество рекурсивных вызовов предиката `fib`, равное искомому числу Фиббоначи.

Давайте попробуем повысить эффективность вычисления чисел Фибоначчи. Будем искать сразу два числа Фибоначчи: то, которое нам нужно найти, и следующее за ним. Соответственно, предикат будет иметь

третий дополнительный аргумент, в который и будет помещено следующее число. Базис рекурсии из двух предложений сожмется в одно, утверждающее, что первые два числа Фибоначчи равны единице.

Вот как будет выглядеть этот предикат:

```
fib_fast(1,1,1):-!. /* первые два числа Фибоначчи равны
                       единице */
fib_fast(N,FN,FN1):-
                N1=N-1,fib_fast(N1,FN_1,FN),
                                /* FN_1 это N-1-е число
                                   Фибоначчи, FN это
                                   N-е число Фибоначчи */
FN1=FN+FN_1. /* FN1 это N+1-е число Фибоначчи */
```

Несмотря на то, что предикат `fib_fast` находит, в отличие от предиката `fib`, не одно число Фибоначчи, а сразу два, он использует намного меньше стекового пространства и работает во много раз быстрее. Для вычисления числа Фибоначчи с номером `N` (а заодно и `N+1`-го числа Фибоначчи) необходимо всего лишь `N` рекурсивных вызовов предиката `fib_fast`.

Если нам не нужно следующее число Фибоначчи, можно сделать последним аргументом анонимную переменную или добавить описанный ниже двухаргументный предикат:

```
fib_fast(N,FN):-
     fib_fast(N,FN,_).
```

Обратите внимание, что если во втором правиле процедуры, описывающей предикат `предок`, с которого мы начали знакомство с рекурсией, изменить порядок подцелей, с декларативной точки зрения смысл останется прежним:

```
предок2(Предок,Потомок):-
        родитель(Предок,Потомок). /* предком является
                                     родитель */
предок2(Предок,Потомок):-
        предок2(Человек,Потомок), /* предком является
                                     родитель предка */
родитель(Предок,Человек).
```

Однако работать модифицированная процедура будет совсем не так. В случае, если вызвать предикат `предок2`, указав в качестве аргументов

имена людей, один из которых является предком другого, он успешно подтвердит, что первый человек — предок второго. Во всех остальных ситуациях (один из аргументов свободен; человек, имя которого указано в качестве первого аргумента, не является предком человека, чье имя указано в качестве второго аргумента) все произойдет не так, как следовало бы. После того, как этот предикат выдаст те же ответы, что и исходный предикат `предок`, он зациклится и в итоге выдаст сообщение о том, что стек переполнен.

С оригинальным предикатом `предок` этого не происходило, потому что в его втором правиле первой подцелью стоял вызов подцели `родитель`. В результате выполнения этой подцели переменная `Человек` получала в качестве значения имя какого-то человека. Поэтому в момент вызова второй подцели `предок(Человек,Потомок)` переменная `Человек` была уже связанной. В новой же версии предиката второе правило начинается с вызова подцели `предок2(Человек,Потомок)`, причем переменная `Человек` в ней свободна. После того, как будут исчерпаны все альтернативы для означивания свободных переменных через первое предложение процедуры, подцель будет унифицироваться с заголовком второго предложения. Свободная переменная `Человек` будет связана с переменной `Предок`, а переменная `Потомок` подцели — с переменной `Потомок` заголовка правила. При попытке выполнить первую подцель правила все повторится. Этот процесс будет продолжаться до тех пор, пока не будет исчерпано все свободное пространство стека.

Такой вид рекурсии, когда тело правила начинается с рекурсивного вызова определяемого предиката, называется *левосторонней рекурсией*. С левосторонней рекурсией очень часто возникают описанные проблемы. Поэтому нужно стараться, если возможно, избегать использования левосторонней рекурсии, в отличие от правосторонней или хвостовой рекурсии.

Лекция 5. Основы Турбо Пролога

Структура программы на Турбо-Прологе. Домены: стандартные, списковые, составные. Альтернативные домены. Программы: «Родственники», факториал, возведение в степень, числа Фиббоначи.

Ключевые слова: домены, составные домены, константы, директивы компилятора, предложения, цели внутренние, цели внешние.

В этой лекции мы изучим специфику среды Турбо Пролог, а также отличия этой разновидности языка Пролог от других версий. Во-первых, Турбо Пролог — это компилируемый язык, а не интерпретируемый, как некоторые другие версии Пролога. Во-вторых, в Турбо Прологе принята строгая типизация данных (для повышения скорости трансляции и выполнения программ). В начале программы на Турбо Прологе обычно располагаются разделы описаний объектов программы. В-третьих, в Турбо Прологе отсутствует возможность рассматривать правила как данные, т.е. добавлять и удалять их во время работы. В процессе выполнения программы в нее можно добавлять и из нее можно удалять только факты. В-пятых, в нем нельзя определять операции. Многое из вышеперечисленного можно считать недостатками, однако в целом все это приводит к тому, что Турбо Пролог отличается высокой скоростью компиляции и выполнения.

Структура программы на Турбо Прологе

Программа на Турбо Прологе состоит из следующих семи разделов:

- директивы компилятора;
- **CONSTANTS** — раздел описания констант;
- **DOMAINS** — раздел описания доменов;
- **DATABASE** — раздел описания предикатов внутренней базы данных;
- **PREDICATES** — раздел описания предикатов;
- **CLAUSES** — раздел описания предложений;
- **GOAL** — раздел описания внутренней цели.

В программе не обязательно должны быть все эти разделы. Так, например, она может состоять из одного описания цели:

```
GOAL
write("hello"),readchar(_).
```

Эта программа, вполне в императивном духе, выведет сообщение (с помощью стандартного предиката `write`) и будет ожидать нажатия пользователем любой клавиши (стандартный предикат `readchar` читает символ).

Однако, как правило, программа содержит, по меньшей мере, разделы *PREDICATES* и *CLAUSES*.

Если программа запускается в среде разработки Турбо Пролога, то раздел *GOAL* необязателен. При написании же программы, не зависящей от среды разработки, в ней необходимо указать внутреннюю цель.

В программе может быть несколько разделов описаний *DOMAINS*, *PREDICATES*, *DATABASE* и *CLAUSES*. Однако разделов GOAL не может быть в программе более одного.

Порядок разделов может быть произвольным, но при этом константы, домены и предикаты должны быть определены до их использования. Однако в разделе *DOMAINS* можно ссылаться на домены, которые будут объявлены позже.

Рассмотрим разделы немного подробнее.

Директивы компилятора

В самом начале программы можно расположить одну или несколько директив компилятора, которые дают компилятору дополнительные инструкции по обработке программы.

Давайте для примера рассмотрим несколько наиболее широко используемых директив компилятора.

Директива *trace* применяется при отладке программы для трассирования. Этот процесс немного похож на пошаговое выполнение императивной программы с отслеживанием значений переменных. Трассировка позволяет пользователю наблюдать за ходом выполнения программы. Если после ключевого слова *trace* указаны имена предикатов через запятую, то трассировка идет только по этим предикатам. В противном случае — по всем предикатам программы. После завершения отладки трассировку нужно выключить, чтобы компилятор мог осуществить оптимизацию хвостовой рекурсии, о которой рассказывалось в предыдущей лекции.

Во время исполнения программы при включенной трассировке в специальном окне трассировки будет отображаться следующая информация:

- после слова *«CALL»* будет указано имя выполняемого предиката (текущая подцель) и его параметры;
- после слова *«FAIL»* будет выводиться имя текущей подцели, которая не была достигнута;
- после слова *«RETURN»* будет выводиться результат вычисления текущей подцели, в случае успеха. При этом если у подцели есть еще альтернативы, к которым возможен возврат, то перед именем предиката высвечивается звездочка («*»);
- слово *«REDO»* перед именем предиката указывает на то, что произошел возврат и происходит вычисление альтернативного решения.

Переход от подцели к подцели вызывается нажатием функциональной клавиши *F10*. При этом в окне редактирования выполняющуюся подцель указывает курсор, она также отображается в окне трассировки с параметрами и дополнительной информацией.

Директива *nowarnings* используется для подавления предупреждения системы о том, что какая-то переменная встречается в предложении только один раз. Эту директиву стоит использовать только в хорошо отлаженных программах. Как правило, для подавления такого предупреждения (*«WARNING: The variable is only used once»*) достаточно заменить переменную, которая встретилась только один раз, на анонимную переменную.

С помощью директивы *include* при компиляции в исходный текст можно вставить содержимое некоторого файла.

Заметим, что многие директивы компилятора могут быть не только расположены в тексте программы, но и установлены в меню среды разработки Турбо Пролога (*Options→Compiler Directives)*. Значение директивы компилятора, указанное в тексте программы, имеет более высокий приоритет, чем значение, установленное в меню.

Раздел описания констант

Раздел, озаглавленный зарезервированным словом *CONSTANTS,* предназначен для описания констант. Объявление константы имеет вид:

```
<имя константы>=<значение>
```

Имя константы должно быть *идентификатором,* то есть оно может состоять из английских букв, цифр и знака подчеркивания, причем не может начинаться с цифры.

Каждое определение константы должно размещаться в отдельной строке.

Например:

```
CONSTANTS
pi=3.14
bgi_path="c:\\prolog\\bgi"
```

В разделе описания констант можно использовать в качестве первого символа имени константы прописные символы, потому что в этом разделе прописные и строчные символы не различаются. Однако при использовании констант в разделе описания предложений нужно задействовать в качестве первого символа имени константы только строчные символы, чтобы Пролог-система не восприняла константу как переменную.

Разделов описания констант может быть несколько, но каждая константа должна быть определена до ее первого использования.

Раздел описания доменов

Раздел описания доменов является аналогом раздела описания типов в обычных императивных языках программирования и начинается с ключевого слова *DOMAINS*.

В Турбо Прологе имеются стандартные домены, которые не нужно указывать в разделе описания доменов. Основные стандартные домены — это:

integer — целое число (из промежутка —32768...32767);

real — действительное число (лежащее между ±1e–307...±1e308);

char — символ, заключенный в одиночные апострофы;

string — последовательность символов, заключенная в двойные кавычки;

symbol — символическая константа (начинающаяся со строчной буквы последовательность букв латинского алфавита, цифр и знаков подчеркивания или последовательность любых символов, заключенная в кавычки). Этот домен соответствует понятию *атома*, с которым мы познакомились во второй лекции;

file — файл (подробному изучению файлов будет посвящена лекция 12).

В разделе описания доменов объявляются любые нестандартные домены, используемые в качестве аргументов предикатов.

Объявление домена имеет следующий вид:

```
<имя домена>=<определение домена>
```

или

```
file=<имя файлового домена1>;...;<имя файлового доменаN>
```

Удобно использовать описание доменов для сокращения имен стандартных доменов. Например, чтобы не писать каждый раз *integer*, можно написать следующее:

```
DOMAINS
i=integer
```

и далее использовать вместо ключевого слова `integer` односимвольное обозначение `i`.

Из доменов можно конструировать *составные* или *структурные* домены *(структуры)*. Структура описывается следующим образом:

```
<имя структуры>=<имя функтора>(<имя домена первой
компоненты>,...,<имя домена последней компоненты>)
[;<имя функтора>(...)]*
```

Каждая компонента структуры в свою очередь может быть структурой. Например, структура, описывающая точку на плоскости и имеющая две компоненты (координаты точки)

```
point = p(integer, integer)
```

может входить в качестве компоненты в более сложную структуру, описывающую треугольник:

```
triangle = tr(point, point, point)
```

В описание структуры могут входить альтернативы, разделенные символом *«;»* или ключевым словом *«or»*.

Так, структуру, описывающую точку и на плоскости, и в пространстве, можно задать следующим образом:

```
point = p(integer, integer);p(integer, integer, integer).
```

Описание файлового домена имеет вид:

```
file = <символическое имя файла 1>;...;
<символическое имя файла N>
```

Для представления данных в Турбо Прологе, в отличие от стандартных алгоритмических языков программирования, используются не массивы, а списки. Подробнее списки будут рассмотрены в следующей лекции, а пока заметим, что списковый домен задается следующим образом:

```
<имя спискового домена>=<имя домена элементов списка>*
```

Например, список целых чисел описывается так:

```
list_of_integer=integer*
```

Раздел описания предикатов внутренней базы данных

Работа с внутренними (динамическими) базами данных в Прологе будет рассмотрена в лекции 13. Начинается раздел описания предикатов внутренней базы данных с зарезервированного слова DATABASE и описываются в нем те предикаты, которые можно в процессе выполнения программы добавлять во внутреннюю базу данных или удалять оттуда. Описываются предикаты базы данных аналогично предикатам в разделе описания предикатов PREDICATES, который мы сейчас рассмотрим.

Раздел описания предикатов

В разделе, озаглавленном зарезервированным словом *PREDICATES,* содержатся описания определяемых пользователем предикатов. В традиционных языках программирования подобными разделами являются разделы описания заголовков процедур и функций. Описание n-местного предиката имеет следующий вид:

```
<имя предиката>(<имя домена первого аргумента>,...,
<имя домена n-го аргумента>).
```

Домены аргументов должны быть либо стандартными, либо объявленными в разделе описания доменов. Обратите внимание на то, что имя предиката в Турбо Прологе должно быть *идентификатором*, т.е. оно должно состоять только из английских букв, цифр и символа подчеркивания, причем не может начинаться с цифры.

Например, предикат, описывающий отношение «мама», которым мы пользовались в третьей лекции, может быть описан следующим образом:

```
PREDICATES
mother(string,string)
```

Это описание означает, что у предиката два аргумента, причем оба строкового типа.

Один предикат может иметь несколько описаний. Это используется, когда нам нужно, чтобы предикат работал с аргументами различной природы. Например, в следующей лекции мы запишем предикат, который будет проверять принадлежность элемента списку. Описать его можно, например, так:

```
PREDICATES
member(integer,integer*)
```

```
member(real,real*)
member(char,char*)
member(string,string*)
```

Такие описания означают, что у предиката два аргумента. При этом возможны четыре варианта использования этого предиката. Первый аргумент может быть целым или вещественным числом, символом или строкой, второй аргумент, соответственно, списком целых или вещественных чисел, или списком, элементами которого являются символы, или списком, состоящим из строк. При этом процедура, реализующая этот предикат в разделе описания предложений, будет единственной.

Кроме того, при описании предиката можно указать, будет он детерминированным или недетерминированным. Детерминированный предикат возвращает только одно решение, а недетерминированный предикат при помощи поиска с возвратом может давать много решений. Детерминированные предикаты менее требовательны к оперативной памяти и выполняются быстрее. Отсечение, с которым мы познакомились в третьей лекции, позволяет превращать недетерминированные предикаты в детерминированные.

Для того чтобы указать, что предикат является детерминированным (недетерминированным), нужно перед его именем поместить зарезервированное слово *determ (nondeterm)*. Если ни *determ*, ни *nondeterm* при описании предиката не использовались, то, по умолчанию, предикат считается детерминированным.

В Турбо Прологе имеется директива компилятора *check_determ*, которая принудительно включает проверку предикатов на детерминированность.

В Турбо Прологе есть так называемые *стандартные (встроенные) предикаты*, которые не нужно описывать в разделе описания предикатов PREDICATES. Наиболее употребляемые из них мы рассмотрим чуть позже. Все встроенные предикаты являются детерминированными.

Раздел описания предложений

Этот раздел можно считать основным разделом программы, потому что именно в нем содержатся факты и правила, реализующие пользовательские предикаты. Все предикаты, которые применяются в этом разделе и не являются стандартными предикатами, должны быть описаны в разделе описания предикатов или в разделе описания предикатов базы данных. Начинается этот раздел со служебного слова CLAUSES.

Напомним, что предложения, у которых в заголовке указан один и тот же предикат, должны идти друг за другом. Такой набор предложений называется *процедурой*. Предложения довольно подробно обсуждались в

третьей лекции, поэтому сейчас мы не будем подробно останавливаться на этом. Заметим только, что программу на Прологе принято оформлять по следующим правилам:

- между процедурами пропускается пустая строка;
- тело правила записывается со следующей строки, после строки, в которой был заголовок, с отступом;
- каждую подцель записывают на отдельной строке, одну под другой.

Эти правила не являются обязательными, но они делают программу более «читабельной».

Раздел описания внутренней цели

С зарезервированного слова GOAL начинается раздел описания *внутренней цели* программы. Если этот раздел отсутствует, то после запуска программы Пролог-система выдает приглашение вводить вопросы в диалоговом режиме (*внешняя цель*). При выполнении внешней цели Пролог-система ищет все решения, выводя все возможные значения для переменных, участвующих в вопросе. Если же выполняется внутренняя цель, то осуществляется поиск только первого решения, а для получения всех решений нужно предпринимать дополнительные действия.

Программа, компилируемая в исполняемый файл, который можно запускать независимо от среды разработки, обязательно должна иметь внутреннюю цель. Внешнюю цель обычно применяют на этапе отладки программы.

Внешние и внутренние цели уже обсуждались в третьей лекции и еще будут рассматриваться в следующей лекции, поэтому сейчас мы не будем останавливаться на этом вопросе более подробно.

Пример. Запишем полностью реализацию на Турбо Прологе той базы знаний про мам и бабушек, которую мы рассматривали в качестве примера в третьей лекции. Нажимаем комбинацию клавиш Alt+E (от Editor), попадаем в редактор. Набираем код, приведенный ниже.

```
DOMAINS /* раздел описания доменов */
s=string /* вводим синоним для строкового типа данных */
PREDICATES /* раздел описания предикатов */
mother(s,s) /* предикат мама будет иметь два аргумента
               строкового типа */
grandmother(s,s) /* то же имеет место и для предиката
                    бабушка */
CLAUSES /* раздел описания предложений */
mother("Наташа","Даша"). /* «Наташа» и «Даша» связаны
                            отношением мама */
```

```
mother("Даша","Маша"). /* «Даша» и «Маша» также
                           принадлежат отношению мама */
grandmother(X,Y):- /* X является бабушкой Y, если
                      найдется такой Z, что */
     mother(X,Z), /* X является мамой Z, а */
     mother(Z,Y). /* Z является мамой Y */
```

Для запуска программы нажимаем *Alt+R* (Run). Так как раздела описания внутренней цели в нашей программе не было, Пролог-система выведет приглашение на ввод внешней цели (*«GOAL:»*). Вводим вопросы, наблюдаем результаты. Повтор предыдущей цели *F8*.

Теперь давайте рассмотрим некоторые наиболее употребляемые стандартные предикаты. Начнем с предикатов, предназначенных для вывода и ввода.

Предикаты ввода-вывода

Турбо Пролог имеет отдельные предикаты для чтения с клавиатуры или из файла данных целого, вещественного, символьного и строкового типа. Работе с файлами будет посвящена лекция 12, а сейчас мы рассмотрим чтение из стандартного устройства ввода информации (клавиатуры) и, соответственно, запись на стандартное устройство вывода информации (монитор).

Предикат *readln* считывает строку с текущего устройства ввода и связывает ее со своим единственным выходным параметром.

Предикат *readint* читает с текущего устройства целое число и связывает его со своим единственным выходным параметром.

Предикат *readreal* отличается от предиката *readint* тем, что он считывает не целое, а вещественное число.

Для чтения символа с текущего устройства ввода используется предикат *readchar*. Есть еще предикат *inkey*, который так же, как и *readchar*, читает символ со стандартного устройства ввода. Разница между ними в том, что предикат *readchar* приостанавливает работу программы до тех пор, пока не будет введен символ, а предикат *inkey* не прерывает выполнение программы. Если нужно просто проверить, нажата ли клавиша, можно воспользоваться предикатом *keypressed*, не имеющим аргументов.

Предикат *readterm* предназначен для чтения сложных термов. У него два параметра: первый входной указывает имя домена, второй параметр конкретизируется термом домена, записанного в первом параметре. Если считанная этим предикатом строка не соответствует домену, указанному в его первом параметре, предикат выдаст сообщение об ошибке.

Для записи данных в текущее устройство записи служит предикат *write*. Он может иметь произвольное количество параметров. Кроме того, в Турбо Прологе есть еще и предикат *writef*, который служит для форматного вывода данных.

Для осуществления перехода на следующую строку (возврат каретки и перевод строки) применяется предикат *nl*, не имеющий параметров.

Описанная ниже группа предикатов служит для преобразования типов.

Предикат *upper_lower* имеет два аргумента и три варианта использования. Если в качестве первого аргумента указана строка (или символ), а второй аргумент свободен, то второй аргумент будет означен строкой (символом), полученной из первого аргумента преобразованием к нижнему регистру. Если в исходной строке были прописные английские буквы, то они будут заменены строчными. Если же, наоборот, первый аргумент свободен, а второй аргумент — это строка (или символ), то первый аргумент получит значение, равное строке (символу), полученной из второго аргумента преобразованием к верхнему регистру. Если в строке, находящейся во втором аргументе, были строчные английские буквы, то они будут заменены прописными. И, наконец, имеется третий вариант использования. Если и первый, и второй аргументы связаны, то предикат будет истинным только в том случае, если во втором аргументе находится строка (символ), которая получается из строки, находящейся в первом аргументе, путем замены всех прописных английских букв на строчные. В противном случае предикат будет ложным.

Также имеют два параметра и три варианта использования предикаты *str_int, str_real*. Первый преобразует строку в целое число и наоборот. Второй служит для превращения строки в вещественное число или вещественного числа в строку.

Предикат *str_char* имеет те же параметры использования и применяется для преобразования односимвольной строки в один символ и наоборот.

Немного по-другому работает предикат *char_int*. Он позволяет переходить от символа к его ASCII-коду и обратно.

Хотя Пролог — не самый лучший инструмент для выполнения большого объема вычислений, в нем имеются стандартные средства для реализации обычных вычислений. При этом можно использовать четыре арифметических операции (сложение (+), вычитание (–), умножение (*) и деление (/)), а также целочисленное деление (*div*) и взятие остатка от деления одного целого числа на другое (*mod*). Для сравнения чисел можно воспользоваться операциями равно (=), не равно (<>), больше (>), больше или равно (>=), меньше (<), меньше или равно (<=).

Кроме того, можно использовать обычные математические функции, такие как: логарифмы натуральный (*ln*) и десятичный (*log*), квадрат-

ный корень (*sqrt*), модуль (*abs*), экспонента (*exp*). Тригонометрические функции: синус (*sin*), косинус (*cos*), тангенс (*tan*), арктангенс (*arctan*). Величины углов указываются в радианах.

Функция *trunc* отбрасывает дробную часть своего параметра, а функция *round* округляет вещественное число до ближайшего целого.

Для вычисления псевдослучайных чисел имеется два варианта предиката *random*. Первый вариант имеет один выходной параметр, в который помещается сгенерированное вещественное число, лежащее в промежутке межу нулем и единицей. Второй вариант этого предиката — двухаргументный. В качестве первого входного аргумента указывается целое число. Второй аргумент означивается целым числом, лежащим между нулем и первым аргументом.

Нуль-местный предикат *true* всегда истинен, а нуль-местный предикат *fail* – всегда ложен. Предикат *fail* часто используется для организации поиска с возвратом. Причем размещение какой-либо подцели в теле правила после предиката fail бессмысленно, поскольку в связи с тем, что этот предикат всегда терпит неудачу, цель никогда не будет достигнута.

Одноместный предикат *free* истинен, если его аргументом является свободная переменная, и ложен в противном случае. Предикат *bound*, наоборот, истинен, если его аргумент — это связанная переменная, и ложен, если его аргумент свободен.

В Турбо Прологе любой текст, находящийся между символами `/*` и `*/`, рассматривается как комментарий. Кроме того, любой текст между символом `%` и концом строки также воспринимается как комментарий. Комментарий отличается от остального текста тем, что он игнорируется компилятором Турбо Пролога. Соответственно, комментарии пишутся не для компилятора, а для человека; для того, чтобы сделать программу более легкой для понимания.

Лекция 6. Управление выполнением программы на Прологе

Метод поиска в глубину. Откат после неудачи. Отсечение и откат. Метод поиска, определяемый пользователем.

Ключевые слова: бэктрекинг, откат, поиск в глубину, поиск с возвратом, откат после неудачи.

Эта лекция посвящена способам организации управления программой при программировании на Прологе. Конечно, какую-то часть способов организации в явном или неявном виде мы уже рассмотрели в предыдущих лекциях. Здесь будет сформулировано явно то, что до сих пор использовалось в неявном виде. Мы разберем, в каких случаях применяются эти методы, как они работают и как ими пользоваться. Также мы рассмотрим некоторые новые методы, которых до сих пор не касались.

Начнем с того, что еще раз обсудим *бэктрекинг* (или *откат*, или *поиск с возвратом*, или *механизм поиска в глубину*). Откат уже упоминался в предыдущих лекциях. Суть этого механизма такова: в том месте программы, где возможен выбор нескольких вариантов, Пролог сохраняет в специальный стек точку возврата для последующего возвращения в эту позицию. Точка возврата содержит информацию, необходимую для возобновления процедуры при откате. Выбирается один из возможных вариантов, после чего продолжается выполнение программы.

Во всех точках программы, где существуют альтернативы, в стек заносятся указатели. Если впоследствии окажется, что выбранный вариант не приводит к успеху, то осуществляется откат к последней из имеющихся в стеке точек программы, где был выбран один из альтернативных вариантов. Выбирается очередной вариант, программа продолжает свою работу. Если все варианты в точке уже были использованы, то регистрируется неудачное завершение и осуществляется переход на предыдущую точку возврата, если такая есть. При откате все связанные переменные, которые были означены после этой точки, опять освобождаются.

При объяснении сущности бэктрекинга часто приводят пример с поиском пути в лабиринте. Один из возможных способов найти выход из лабиринта — это на каждой развилке поворачивать в одну и ту же сторону, до тех пор, пока не попадешь в тупик. После попадания в тупик нужно вернуться до ближайшей развилки. На ней нужно выбрать другое направление. После этого нужно опять на всех развилках выбирать поворот в том же направлении, что и в самом начале. Продолжаем этот алгоритм до тех пор, пока не выберемся из лабиринта.

Пример. Рассмотрим работу механизма отката на примере слегка модифицированной программы, описывающей родственные отношения. Разделы описания доменов и предикатов оставим без изменения, а порядок фактов в разделе предложений изменим, для того чтобы он лучше соответствовал нашим целям. Получим следующую программу:

```
DOMAINS /* раздел описания доменов */
s=string /* вводим синоним для строкового типа данных */
PREDICATES /* раздел описания предикатов */
mother(s,s) /* предикат мама будет иметь два аргумента
               строкового типа */
grandmother(s,s) /* то же имеет место и для предиката
                    бабушка */
CLAUSES /* раздел описания предложений */
mother("Даша","Маша"). /* «Даша» и «Маша» связаны
                          отношением мама */
mother("Наташа","Даша"). /*  «Наташа» является мамой
                             «Даши» */
mother("Наташа","Глаша"). /* «Наташа» и «Глаша» связаны
                             отношением мама */
mother("Даша","Саша"). /* «Даша» является мамой «Саши» */
grandmother(X,Y):- /* X является бабушкой Y,
                      если найдется такой Z, что */
                   mother(X,Z), /* X является мамой Z, а */
                   mother(Z,Y). /* Z является мамой Y */
```

В качестве внешней цели, после запуска программы зададим вопрос об именах всех бабушек и внучек `(grandmother(B,V))`.

Чтобы проследить процесс работы Пролог-системы при поиске ответа на наш вопрос, можно включить трассировку, записав в начале программы директиву компилятору *trace*. В окне редактирования курсор указывает подцель, которая выполняется на данном шаге. В окне трассировки отображается дополнительная информация. Напомним, что переход от подцели к подцели осуществляется с помощью функциональной клавиши *F10*.

Для выполнения цели `grandmother(B,V)` должны быть удовлетворены две подцели: `mother(B,Z)` и `mother(Z,V)`. Первая подцель унифицируется с первым предложением, описывающим отношение «быть мамой». При этом переменная `B` конкретизируется именем `"Даша"`, а переменная `Z` — `"Маша"`. В окне трассировки в этот момент результат вычисления текущей подцели `(mother("Даша","Маша"))`, выводящийся после слова *RETURN*, сопровождается звездочкой (*), которая показывает, что у подцели есть альтернативные решения. Это то самое место, указатель на

которое Пролог заносит в стек точек возврата для возможного последующего возвращения.

Затем делается попытка удовлетворить вторую подцель mother(Z,V), причем переменная Z означена именем "Маша". Попытка унифицировать эту подцель с одним из фактов, имеющих отношение к предикату mother, оказывается неудачной. Это происходит потому, что в нашей базе знаний нет никакой информации о детях Маши. О неуспехе говорит слово *FAIL* в окне трассировки. Происходит откат до места, сохраненного в стеке точек возврата. При этом переменные B и Z, означенные к моменту отката, вновь становятся свободными.

Выбирается вторая альтернатива. Переменная B при этом становится равной имени "Наташа", а переменная Z получает значение "Даша". Звездочка в окне трассировки, как и при первом проходе этой точки, показывает нам, что исчерпаны еще не все из имеющихся альтернатив, удовлетворяющих нашей подцели.

Делается попытка найти решение для второй подцели mother(Z,V) (при Z = "Даша"). Первое же предложение в процедуре, реализующей предикат mother, унифицируется с текущей подцелью, переменная V получает значение "Маша". Очередная звездочка в окне трассировки отмечает, что указатель на это место помещен в стек точек возврата, для того чтобы вернуться сюда, и что возможны другие означивания для переменной V, приводящие текущую подцель к успеху.

Получаем, что ответ на наш вопрос возможен при следующих значениях переменных: B=Наташа, V=Маша. Этот ответ отображается в окне диалога, после чего осуществляется откат к последнему месту, записанному в стек точек возврата. При этом освобождается переменная V, которая уже была означена именем "Маша". Подцель mother(Даша,V) унифицируется с заголовком последнего предложения процедуры, определяющей предикат mother. Переменная V означивается именем "Саша". В диалоговом окне выводится второй возможный ответ на заданный нами в качестве внешней цели вопрос: B=Наташа, V=Саша.

Альтернативных решений для подцели mother(Даша,V) больше нет. Соответственно, в окне трассировки отсутствует звездочка, а в стеке точек возврата нет больше указателя на то место, куда можно было возвращаться для того, чтобы выбирать новые значения для второй подцели правила, определяющего отношение grandmother.

Однако в стеке точек возврата еще остается указатель на тот участок программы, где находились означивания переменных для первой подцели в теле правила, определяющего отношение «быть бабушкой». Пролог-система осуществляет откат, попутно освобождая переменные.

Первая подцель сопоставляется с третьим фактом mother("Наташа","Глаша"). В окне трассировки видим уже знакомый символ звездочки,

который свидетельствует о том, что испробованы еще не все возможные варианты для текущей подцели `mother(B,Z)`. Делаются последовательные попытки сопоставить подцель `mother("Глаша",V)` с одним из фактов, имеющихся в базе знаний. Однако эти попытки заканчиваются неудачей, поскольку наша программа не содержит информации о детях Глаши. В окне трассировки отображается слово FAIL, информирующее нас об этой неудаче.

Процесс выполнения программы в очередной, последний раз откатывается к тому месту, где можно выбрать решение для первой подцели. Подцель унифицируется с последним предложением в процедуре, описывающей знания, касающиеся мам. Переменная `B` конкретизируется именем `"Даша"`, а переменная `Z` — `"Саша"`. Других вариантов для сопоставления первой подцели не остается. Стек точек возврата пуст. В окне трассировки нет индикатора, сообщающего об альтернативных решениях, к которым возможно возвращение. Пролог-система пытается сопоставить с чем-нибудь вторую подцель `mother("Саша",V)`, однако ей не удается этого сделать. Ни один из фактов не содержит в качестве первого аргумента имя `"Саша"`. Очередная неудача в попытке найти внуков для Даши.

Программа завершается. В диалоговом окне — два найденных в процессе работы решения:

```
B=Наташа, V=Маша
B= Наташа, V=Саша
2 Solutions
```

Теперь посмотрим, какие имеются у программиста возможности по управлению откатом.

Рассмотрим модификацию механизма поиска в глубину, которая позволяет получать дополнительные решения и называется *метод отката после неудачи*. Этот метод используется в ситуации, когда нужно получить не один ответ, а все возможные в данной ситуации ответы. Например, если вопрос является внутренней целью, то Турбо Пролог останавливает поиск после первого же успешного вычисления цели. При этом выявляется только первое решение.

Пример. Давайте зададим тот же вопрос, что и в предыдущем примере, но уже не как внешнюю цель, а укажем ее в разделе описания внутренней цели:

```
GOAL
grandmother(B,V)
```

Если запустить эту программу, она завершит свою работу, так ничего и не выдав в окне диалога. Дело в том, что при наличии в программе вну-

тренней цели Турбо Пролог не отображает в диалоговом окне значения переменных, тогда как при выполнении внешней цели Турбо Пролог выводит в окне значения всех содержащихся в вопросе переменных. Это первое существенное отличие внешней и внутренней цели.

Однако мы можем сами организовать отображение результатов вычисления внутренней цели, добавив к ней в качестве подцелей вывод значений переменных B и V на экран, используя встроенный предикат write. Раздел описания внутренней цели при этом может выглядеть, например, так:

```
GOAL
grandmother(B,V),write("Имя бабушки – ",B), write(",
имя внучки – ",V),nl
```

В этом случае мы увидим на экране имена бабушки и внучки, но в окне диалога отобразится не два решения, а всего одно:

```
Имя бабушки – Наташа, имя внучки – Маша
```

Дело в том, что при наличии в программе внутренней цели Турбо Пролог находит только одно возможное означивание переменных, а не все возможные, как в случае внешней цели. Это второе отличие внешней и внутренней цели. Если нам необходимо получить все решения, нужно организовать это специально, например, с помощью метода отката после неудачи. Обычно внешние цели используются на этапе отладки новых предикатов. Если же нам нужна программа, которая может запускаться вне среды разработки, в ней обязательно должна быть внутренняя цель.

В методе отката после неудачи обычно используется всегда ложный предикат `fail`, о котором говорилось в прошлой лекции. Хотя, по большому счету, вместо этого предиката можно воспользоваться каким-нибудь заведомо ложным выражением. Например, *1=2* или чем-то в этом роде.

Если добавить к нашей внутренней цели предикат `fail`, то получим в окне диалога оба решения, которые мы могли наблюдать, когда задавали этот вопрос, используя внешнюю цель:

```
Имя бабушки – Наташа, имя внучки – Маша
Имя бабушки – Наташа, имя внучки – Саша
```

Пример. Теперь давайте напишем предикат, который будет выводить на экран с помощью стандартного предиката `write` имена всех дочек.

```
show_names:-
          mother(_,Name), /* означивает переменную Name
                             именем дочки */
          write(" ", Name), nl,
                          /* выводит значение переменной
                             Name на экран */
          fail. /* вызывает откат на место, сохраненное
                   в стеке точек возврата */
```

Допишем этот предикат к нашей программе про мам и бабушек, не забыв добавить его описание в раздел PREDICATES.

В качестве внутренней цели укажем вывод сообщения "Имена дочек:" (с помощью встроенного предиката write), переведем курсор на следующую строку стандартным предикатом nl. В качестве третьей подцели запишем предикат show_names, выводящий имена всех дочек:

```
GOAL
write("Имена дочек:"),nl,
show_names.
```

Как будет работать эта программа? Сначала будет выведена строка "Имена дочек:", и произойдет переход на следующую строку. После этого выполняется подцель show_names.

Во время попытки вычисления этой подцели механизм унификации означивает переменную именем дочери, указанным в качестве второго аргумента в первом предложении процедуры, описывающей предикат mother. Переменная Name получает значение "Маша". При этом в окне трассировки можно видеть звездочку, которая говорит о том, что в стек точек возврата помещен указатель на место, в которое возможен откат для получения других решений подцели mother(_,Name). Имя "Маша" выводится на экран встроенным предикатом write.

После этого предикат fail вызывает неуспешное завершение правила, затем осуществляется откат в точку, помещенную в стек последней. Процесс повторяется до тех пор, пока не будут исчерпаны все варианты достижения подцели mother(_,Name). В окне диалога будут выведены имена всех дочек в том порядке, в котором они упоминались в нашей программе:

```
Имена дочек:
Маша
Даша
Глаша
Саша
```

Пример. Давайте изменим наш предикат, чтобы он выводил имена не всех дочек, а только дочек одной мамы. Для этого нужно добавить предикату аргумент, в качестве которого будет указываться имя мамы. Кроме того, в первой подцели требуется заменить анонимную переменную некоторой обычной переменной, которую затем нужно сравнить с именем мамы, указанной в качестве аргумента предиката:

```
show_names2(Mother):-
          mother(M,Name), /* означивает переменную Name
                             именем дочки мамы Mother */
          M=Mother, /* проверяет совпадение имен мам M
                       и Mother */
          write(" ", Name), nl, /*  выводит значение
                                    переменной Name
                                    на экран */
          fail. /* вызывает откат к месту, сохраненному
                   в стеке точек возврата */
```

Вместо первых двух подцелей `mother(M,Name)`, `M=Mother` можно записать альтернативный вариант: `mother(Mother,Name)`. Результат будет тем же, но процесс вычисления будет отличаться.

Выведем с помощью модифицированного предиката имена дочек Даши. Для этого изменим внутреннюю цель следующим образом:

```
GOAL
write("Имена дочек Даши:"),nl,
show_names2("Даша").
```

Отличие в работе этого предиката от предиката, который выводил имена всех дочек, заключаются в следующем. После того, как переменная `M` будет означена именем очередной мамы, будет проверяться совпадение ее значения с именем `"Даша"`. В случае совпадения будет выведено имя дочери Даши и осуществится откат. Если же имя окажется отличным от "Даша", откат произойдет немедленно. Выполнение программы в этом случае не доберется до вывода имени дочери на экран. Предикат `fail` не потребуется, так как подцель `M=Mother` будет неуспешной сама по себе. Тем не менее, наличие стандартного предиката `fail` в качестве последней подцели правила необходимо для того, чтобы вызвать откат, если подцель `M=Mother` окажется успешной.

По завершении работы этой программы можно будет увидеть в диалоговом окне результаты:

```
Имена дочек Даши:
Маша
Саша
```

Рассмотрим теперь так называемый *метод отсечения и отката*. В основе этого метода лежит использование комбинации предикатов `fail` (для имитации неудачи и искусственной организации отката) и «`!`» (*отсечение* или *cut*), который позволяет прервать этот процесс в случае выполнения какого-то условия. Об отсечении мы уже говорили в третьей лекции. Разберем этот метод на примере.

Пример. Модифицируем еще раз предикат, выводящий имена всех дочек, так чтобы он выводил их не все, а только до определенного имени. У предиката будет один входной аргумент, где указывается имя дочери, на котором должен оборваться вывод на экран имен дочек. В тело правила нужно добавить подцель, в которой текущее имя дочки будет сравниваться с именем дочки, указанным в качестве аргумента предиката. После этой подцели

```
show_names3(Daughter):-
mother(_,Name), /* означивает переменную Name именем
                   дочки */
write(" ", Name), nl, /* выводит значение переменной
                         Name */
Name=Daughter, /*  проверяет совпадение имен дочек Name
                   и Daughter. В случае несовпадения
                   вызывает откат на место, указатель
                   на которое хранится в стеке точек
                   возврата. В случае совпадения, за счет
                   наличия отсечения, завершает поиск
                   и вывод имен дочек */
write("Искомый человек найден!"),!.
```

Этот предикат будет конкретизировать переменную `Name` именем чьей-то дочери, выводить на экран значение этой переменной. После этого будет сравниваться значение переменной `Name` со значением переменной `Daughter`. В случае несовпадения эта подцель терпит неудачу, вызывая откат на первую подцель правила. В случае совпадения имен подцель успешна, управление получает предикат отсечение. Он всегда истинен и запрещает откат для подцелей, расположенных левее. На этом работа предиката `show_names3` завершается. То есть откаты в этом правиле будут продолжаться до тех пор, пока текущее значение переменной Name не совпадет со значением переменной Daughter.

Еще один способ организации повторяющихся действий — так называемый *метод повтора, определяемый пользователем*. Он также использует откат, однако, в отличие от метода отката после неудачи, в котором откат осуществляется только после специально созданной неудачи, в этом методе откат возможен всегда за счет использования специального предиката, обычно кодируемого в виде следующего предложения:

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

Этот предикат всегда успешен. После выполнения первого предложения процедуры в стек точек возврата запоминается указатель, поскольку имеется еще один альтернативный вариант для него. В теле второго предложения опять вызывается предикат `repeat`.

Предикат `repeat` не является встроенным предикатом, а имя `"repeat"` — зарезервированным словом. Соответственно, вместо этого имени можно использовать какое-нибудь другое.

С помощью этого предиката можно организовывать циклы, подобные циклам в императивных языках программирования. Нужно не забыть добавить в правило, организующее цикл, кроме предиката `repeat`, условие завершения цикла и отсечение, чтобы не получилось зацикливания.

Пример. Создадим предикат, который будет дублировать символ, введенный пользователем с клавиатуры. Завершиться этот процесс должен, когда пользователь введет некий ключевой символ, о котором мы заранее договоримся, что его появление означает окончание процесса дублирования символов. Давайте, для определенности, возьмем в качестве такого символа точку (.).

```
double_char:-
        repeat,
        readchar(C), /* читаем символ с клавиатуры
                        в переменную C */
        write(C,C), nl,/* выводим на экран значение
                          переменной C */
        C='.',!, /* сравниваем значение переменной C
                    с символом '.'*/
        nl,write("Была введена точка. Закончили.").
```

Первой подцелью нашего правила записан вызов предиката `repeat`. Он обеспечивает нам повторное выполнение следующих за ним подцелей. Можно эксперимента ради закомментировать предикат `repeat`, дабы

убедиться, что в этом случае цикла не получится. Последующие подцели выполнятся всего один раз.

Далее, используя стандартный предикат `readchar`, осуществляем чтение символа с клавиатуры в переменную C. Посредством встроенного предиката `write` выводим два экземпляра введенного пользователем символа на экран; стандартным предикатом `nl` переводим курсор на следующую строку. Затем значение переменной `C` сравнивается с символом точка (`'.'`). Это условие, которое, с одной стороны, обеспечивает откат на первую подцель (предикат `repeat`), а с другой обеспечивает выход из цикла. Если поступивший с клавиатуры символ отличен от точки, подцель `C='.'` терпит неуспех. Пролог-система осуществляет откат на последнее место, указатель на которое записан в стеке точек возврата. Из подцелей, расположенных левее, альтернативы имелись только у первой подцели (предиката `repeat`). Все повторяется еще раз. Пользователь вводит символ, он дважды отображается на экране, сравнивается с точкой. Процесс повторяется до тех пор, пока введенный символ не окажется точкой. В этом случае подцель `C='.'` окажется успешной, управление перейдет на подцели, расположенные правее. Предикат `nl` сдвинет курсор на начало следующей строки, стандартный предикат `write` выводит на экран сообщение: `"Была введена точка. Закончили."` — и процесс дублирования символов на этом завершается.

Напишем внутреннюю цель, которая проинформирует пользователя о правилах работы нашей программы, после чего запустит предикат `double_char`:

```
GOAL
write("Вводите символы, которые нужно повторить (точка —
завершение)"),nl,
double_char.
```

Лекция 7. Списки

Списки. Рекурсивное определение списка. Операции над списками.

Ключевые слова: список, элемент списка, голова списка, хвост списка, количество элементов списка, последний элемент списка, конкатенация списков, соседние элементы списка, обращение списка.

В императивных языках, как правило, основной структурой данных являются массивы. В Прологе так же, как и в Лиспе, основным составным типом данных является список. В этой лекции мы займемся изучением именно списков.

Дадим сначала неформальное определение списка.

Будем называть *списком* упорядоченную последовательность элементов произвольной длины.

Список задается перечислением элементов списка через запятую в квадратных скобках, так, как показано в приведенных ниже примерах:

- `[monday, tuesday, wednesday, thursday, friday, saturday, sunday]` — список, элементами которого являются английские названия дней недели;
- `[“понедельник“, “вторник“, “среда“, “четверг“, “пятница“, “суббота“, “воскресенье“]` — список, элементами которого являются русские названия дней недели;
- `[1, 2, 3, 4, 5, 6, 7]` — список, элементами которого являются номера дней недели;
- `[‘п’, ‘в’, ‘с’, ‘ч’, ‘п’, ‘с’, ‘в’]` — список, элементами которого являются первые символы русских названий дней недели;
- `[]` — *пустой* список, т.е. список, не содержащий элементов (в языке функционального программирования Лисп он обозначается nil).

Элементы списка могут быть любыми, в том числе и составными объектами. В частности, элементы списка сами могут быть списками.

В разделе описания доменов списки описываются следующим образом:

```
DOMAINS
<имя спискового домена>=<имя домена элементов списка>*
```

Звездочка после имени домена указывает на то, что мы описываем список, состоящий из объектов соответствующего типа.

Например:

```
listI = integer*  /* список, элементы которого –
                     целые числа */
listR = real*   /* список вещественных чисел */
listC = char*   /* список символов */
lists = string* /* список, состоящий из строк */
listL = listI*  /* список, элементами которого являются
                   списки целых чисел */
```

Последнему примеру будут соответствовать списки вида:

```
[[1,3,7],[],[5,2,94],[-5,13]]
```

В классическом Прологе элементы списка могут принадлежать разным доменам, например:

```
[monday, 1, "понедельник"]
```

В Турбо Прологе, в связи со строгой типизацией, все элементы списка должны принадлежать одному домену. Однако можно разместить в одном списке объекты разной природы, используя домен с соответствующими альтернативами.

Например, следующее описание:

```
DOMAINS
element = i(integer); c(char); s(string)
listE = element*
```

позволит иметь дело со списками вида

```
[i(-15), s("Мама"),c('A'),s("мыла"),c('+'),s("раму"),
i(48),c('!')]
```

Дадим рекурсивное определение списка.

Список — это структура данных, определяемая следующим образом:

1. пустой список (`[ ]`) является списком;
2. структура вида `[H|T]` является списком, если `H` — первый элемент списка (или несколько первых элементов списка, перечисленных через запятую), а `T` — список, состоящий из оставшихся элементов исходного списка.

Принято называть `H` *головой* списка, а `T` — *хвостом* списка. Заметим, что выбор переменных для обозначения головы и хвоста не случаен. По-английски голова — Head, а хвост — Tail.

Фактически операция «|» позволяет разделить список на хвост и голову (в Лиспе есть подобные операции car и cdr) или, наоборот, приписать объект (объекты) к началу списка (cons в Лиспе).

Данное определение позволяет организовывать рекурсивную обработку списков, разделяя непустой список на голову и хвост. Хвост, в свою очередь, также является списком, содержащим меньшее количество элементов, чем исходный список. Если хвост не пуст, его также можно разбить на голову и хвост. И так до тех пор, пока мы не доберемся до пустого списка, у которого нет головы.

Например, в списке `[1, 2, 3]` элемент `1` является головой, а список `[2, 3]` — хвостом, т.е. `[1, 2, 3] = [1|[2, 3]]`.

Заметим, что хвост этого списка `[2, 3]`, в свою очередь, может быть представлен в виде головы `2` и хвоста `[3]`, а список `[3]` можно рассматривать в виде головы `3` и хвоста `[]`. Пустой список далее не разделяется.

В итоге получаем, что список `[1, 2, 3]` эквивалентен списку `[1|[2, 3]]`, который, в свою очередь, эквивалентен списку `[1|[2|[3]]]`. Последний сопоставим со списком `[1|[2|[3|[ ]]]]`.

В этом же списке можно выделить два первых элемента и хвост из третьего элемента `[1,2|[3]]`. И, наконец, возможен вариант разбиения на голову из трех первых элементов и пустой хвост: `[1, 2, 3|[]]`.

Чтобы организовать обработку списка в соответствии с приведенным выше рекурсивным определением, нам достаточно задать предложение (правило или факт, определяющее, что нужно делать с пустым списком), которое будет базисом рекурсии, а также рекурсивное правило, устанавливающее порядок перехода от обработки всего непустого списка к обработке его хвоста. Иногда базис рекурсии записывается не для пустого, а для одно- или двухэлементного списка.

В качестве резюме к нашим рассуждениям запишем еще раз определение списка в нотации Бэкуса–Науэра:

```
Список ::= [ ]|[Элемент <,Элемент>*]|[Голова|Хвост]
Голова ::= Элемент <,Элемент>*
Хвост ::= Список
```

Словесно это можно записать так: список или пустой, или представим в виде перечисления элементов, записанных через запятую, или состоит из головы и хвоста, который, в свою очередь, также является списком.

Рассмотрим обработку списков.

Пример. Создадим предикат, позволяющий вычислить длину списка, т.е. количество элементов в списке.

Для решения этой задачи воспользуемся очевидным фактом, что в пустом списке элементов нет, а количество элементов непустого списка,

представленного в виде объединения первого элемента и хвоста, равно количеству элементов хвоста, увеличенному на единицу. Запишем эту идею:

```
length([], 0). /* в пустом списке элементов нет */
length([_|T], L) :-
            length(T, L_T), /* L_T — количество
                               элементов в хвосте */
            L = L_T + 1. /* L — количество элементов
                            исходного списка */
```

Обратите внимание, что при переходе от всего списка к его хвосту нам неважно, чему равен первый элемент списка, поэтому мы используем анонимную переменную.

Разберем на примере, как это будет работать. Пусть нас интересует количество элементов в списке `[1,2,3]`. Запишем соответствующий вопрос Пролог-системе:

```
length([1,2,3],X).
```

Система попытается вначале сопоставить нашу цель с первым предложением `length([], 0)`, однако ей это не удается сделать, потому что первый аргумент цели является непустым списком. Система переходит ко второму предложению процедуры. Сопоставление с заголовком правила проходит успешно, переменная `X` связывается с переменной `L`, список `[1,2,3]` будет сопоставлен со списком `[_|T]`, переменная `T` будет конкретизирована значением `[2,3]`. Теперь система переходит к попытке достижения подцели `length(T,L_T)`. Как и в предыдущем случае, первое предложение с подцелью не сопоставляется, так как список `T` не пустой. При сопоставлении заголовка правила с подцелью хвост `T` конкретизируется одноэлементным списком `[3]`. На следующем шаге рекурсии переменная `T` означена пустым списком (хвост одноэлементного списка). И, значит, наша подцель выглядит следующим образом: `length([], L_T)`. Эта цель сопоставляется с фактом, переменная `L_T` становится равной нулю. Раскручивается обратный ход рекурсии: переменная `L_T` увеличивается на единицу, результат попадает в переменную `L`. Получаем, что длина списка `[3]` равна единице. На следующем обратном шаге происходит еще одно добавление единицы, после чего длина списка `[2,3]` конкретизируется двойкой. И, наконец, на последнем возвратном шаге получаем означивание переменной `L` числом 3 (количеством элементов в списке `[1,2,3]`).

Пример. Создадим предикат, позволяющий проверить принадлежность элемента списку. Предикат будет иметь два аргумента: первый — искомое значение, второй — список, в котором производится поиск.

Построим данный предикат, опираясь на тот факт, что объект принадлежит списку, если он либо является первым элементом списка, либо элементом хвоста. Это может быть записано в виде двух предложений:

```
member(X,[X|_]). /* X – первый элемент списка */
member(X,[_|T]) :-
                member(X,T). /* X принадлежит хвосту T*/
```

Заметим, что в первом случае (когда первый элемент списка совпадает с исходным элементом), нам неважно, какой у списка хвост, и можно в качестве хвоста указать анонимную переменную. Аналогично, во втором случае, если `X` принадлежит хвосту, нам не важно, какой элемент первый.

Отметим, что описанный предикат можно использовать двояко: во-первых, конечно, для того, для чего мы его и создавали, т.е. для проверки, имеется ли в списке конкретное значение. Мы можем, например, поинтересоваться, принадлежит ли двойка списку `[1, 2, 3]`:

```
member(2, [1, 2, 3]).
```

Получим, естественно, ответ: *«Yes»*.

Подобным образом можно спросить, является ли число 4 элементом списка `[1, 2, 3]`:

```
member(4, [1, 2, 3]).
```

Ответом, конечно, будет *«No»*.

Второй способ использования данного предиката — это получение по списку его элементов. Для этого нужно в качестве первого аргумента предиката указать свободную переменную. Например:

```
member(X, [1, 2, 3]).
```

В качестве результата получим список всех элементов списка:

```
X=1
X=2
X=3
```

Третий способ позволит получить по элементу варианты списков, которые могут его содержать. Теперь свободную переменную запишем вторым аргументом предиката, а первым — конкретное значение. Например,

```
member(1, X).
```

Вначале Пролог-система выдаст предупреждение о том, что переменная `X` не связана в первом предложении *(«708 WARNING: The variable is not bound in this clause. (F10=ok, Esc=abort)»*).

У нас есть два способа отреагировать на это предупреждение: нажать кнопку *Esc*, чтобы отказаться от генерации списков, содержащих единицу в качестве элемента; нажать *F10* для того, чтобы продолжить выполнение цели. Во втором случае Пролог-система начнет выдавать варианты списков, содержащих единицу:

```
X=[1|_] /* единица – первый элемент списка */
X=[_,1|_] /* единица – второй элемент списка */
X=[_,_,1|_] /* единица – третий элемент списка */
```

и т.д.

Этот процесс будет продолжаться до тех пор, пока не будет нажата комбинация клавиш *Ctrl+Break*.

Если данный предикат планируется использовать только первым способом, то можно ускорить его работу, устранив поиск элемента в хвосте списка, если он уже найден в качестве первого элемента списка. Это можно сделать двумя способами.

Первый способ. Добавим в правило проверку на несовпадение первого элемента списка с искомым элементом, чтобы поиск элемента в хвосте списка производился только тогда, когда первый элемент списка не является искомым. Модифицированный предикат будет выглядеть следующим образом:

```
member2(X,[X|_]).
member2(X,[Y|T]):-
                    X<>Y, member2(X,T).
```

Заметим, что эту модификацию предиката `member` нельзя использовать для получения всех элементов списка. Если подставить в качестве первого аргумента несвязанную переменную, то при попытке согласования подцели правила неозначенная переменная `X` будет сравниваться с неозначенной переменной `Y`. Получим сообщение об ошибке «*Free variable in expression*».

Второй способ. Добавим в факт отсечение, чтобы в ситуации, когда искомый элемент оказался первым элементом списка, не производился лишний поиск в хвосте исходного списка. Получим:

```
member3(X,[X|_]):-!.
member3(X,[_|T]):-
                    member3(X,T).
```

Заметим, что хотя эта модификация предиката `member` более эффективна, чем исходная, за счет того, что она не выполняет поиск в хвосте после того, как искомый элемент найден, ее можно использовать только для того, чтобы проверить, имеется ли в списке конкретное значение. Если мы попытаемся применить ее для получения всех элементов списка, подставив в качестве первого аргумента несвязанную переменную, то результатом будет только первый элемент списка. Отсечение не позволит нам получить оставшиеся элементы.

Пример. Создадим предикат, позволяющий соединить два списка в один. Первые два аргумента предиката будут представлять соединяемые списки, а третий — результат соединения.

В качестве основы для решения этой задачи возьмем рекурсию по первому списку. Базисом рекурсии будет факт, устанавливающий, что если присоединить к списку пустой список, в результате получим исходный список. Шаг рекурсии позволит создать правило, определяющее, что для того, чтобы приписать элементы списка, состоящего из головы и хвоста, ко второму списку, нужно соединить хвост и второй список, а затем к результату приписать спереди первый элемент первого списка. Запишем решение:

```
conc([ ], L, L). /* при присоединении пустого списка
                    к списку L получим список L */
conc([H|T], L, [H|T1]) :-
conc(T,L,T1). /* соединяем хвост и список L, получаем
                 хвост результата */
```

Заметим, что этот предикат также можно применять для решения нескольких задач.

Во-первых, для соединения списков. Например, если задать вопрос

```
conc([1, 2, 3], [4, 5], X)
```

то получим в результате

```
X= [1, 2, 3, 4, 5]
```

Во-вторых, для того, чтобы проверить, получится ли при объединении двух списков третий. Например, на вопрос:

```
conc([1, 2, 3], [4, 5], [1, 2, 5]).
```

ответом будет, конечно, `No`.

В-третьих, можно использовать этот предикат для разбиения списка на подсписки. Например, если задать следующий вопрос:

```
conc([1, 2], Y, [1, 2, 3]).
```

то ответом будет `Y=[3]`.

Аналогично, на вопрос

```
conc(X, [3], [1, 2, 3]).
```

получим ответ `X=[1, 2]`.

И, наконец, можно спросить

```
conc(X, Y, [1, 2, 3]).
```

Получим четыре решения:

```
X=[], Y=[1, 2, 3]
X=[1], Y=[2, 3]
X=[1, 2], Y=[3]
X=[1, 2, 3], Y=[]
```

В-четвертых, можно использовать этот предикат для поиска элементов, находящихся левее и правее заданного элемента. Например, если нас интересует, какие элементы находятся левее и, соответственно, правее числа 2, можно задать следующий вопрос:

```
conc(L, [2|R], [1, 2, 3, 2, 4]).
```

Получим два решения:

```
L=[1], R=[3, 2, 4].
L=[1, 2, 3], R=[4]
```

В-пятых, на основе нашего предиката `conc` можно создать предикат, находящий последний элемент списка:

```
last(L,X):-
          conc(_,[X],L).
```

Справедливости ради стоит заметить, что этот предикат можно реализовать и «напрямую», без использования предиката conc:

```
last2([X],X). /* последний элемент одноэлементного
                 списка — этот элемент */
last2([_|L],X):-
        last2(L,X). /* последний элемент списка совпадает
                       с последним элементом хвоста */
```

В-шестых, можно определить, используя conc, предикат, позволяющий проверить принадлежность элемента списку. При этом воспользуемся тем, что если элемент принадлежит списку, то список может быть разбит на два подсписка так, что искомый элемент является головой второго подсписка:

```
member4(X,L):-
             conc(_,[X|_],L).
```

В-седьмых, используя предикат, позволяющий объединить списки, можно создать предикат, проверяющий по двум значениям и списку, являются ли эти значения соседними элементами списка. Предикат будет иметь три параметра: первые два — значения, третий — список.

Идея решения заключается в следующем. Если два элемента оказались соседними в списке, значит, этот список можно разложить на два подсписка, причем голова второго подсписка содержит два наших элемента в нужном порядке. Выглядеть это будет следующим образом:

```
neighbors(X,Y,L):-
     conc(_,[X,Y|_],L). /* список L получается путем
                           объединения некоторого списка
                           со списком, голову которого
                           составляют элементы X и Y */
```

Обратите внимание, что этот предикат проверяет только наличие нужных значений в указанном порядке. Если нам неважен порядок, в котором два данных значения встречаются в некотором списке, то следует записать модификацию описанного выше предиката, которая будет проверять оба варианта размещения искомых элементов. Для этого достаточно, чтобы список раскладывался на два подсписка, причем голова второго подсписка содержала два наших элемента либо в прямом, либо в обратном порядке. Соответствующий программный код будет следующим:

```
neighbors2(X,Y,L):-
          conc(_,[X,Y|_],L);
          conc(_,[Y,X|_],L). /* список L получается
                                путем объединения некоторого
                                списка со списком, голову
                                которого составляют элементы X
                                и Y или элементы Y и X */
```

Есть подозрение, что многообразие использований предиката `conc` приведенными выше примерами не исчерпывается.

Пример. Разработаем предикат, позволяющий «обратить» список (записать его элементы в обратном порядке). Предикат будет иметь два аргумента: первый — исходный список, второй — список, получающийся в результате записи элементов первого аргумента в обратном порядке.

Для решения этой задачи воспользуемся рекурсией. Базис: если записать элементы пустого списка (которых нет) в обратном порядке — опять получим пустой список. Шаг рекурсии: для того чтобы получить «перевернутый» список, можно «перевернуть» его хвост и «приклеить» к нему первый элемент исходного списка. Запишем эти размышления.

```
reverse([ ],[ ]). /* обращение пустого списка дает пустой
                     список*/
reverse([X|T],Z):-
             reverse(T,S), conc(S,[X],Z).
                  /* обращаем хвост и приписываем к нему
                     справа первый элемент исходного
                     списка*/
```

Обратите внимание, что вторым аргументом в предикате `conc` должен стоять именно одноэлементный список `[X]`, а не элемент `X`. Это связано с тем, что аргументами предиката `conc` должны быть списки.

Можно написать данный предикат без использования предиката `conc`. Правда, тогда нам придется добавить дополнительный аргумент, в котором мы будем «накапливать» результат. Мы будем «отщипывать» от исходного списка по элементу и дописывать его к вспомогательному списку. Когда исходный список будет исчерпан, мы передадим «накопленный» список в третий аргумент в качестве ответа. До этого момента третий аргумент передается от шага к шагу неконкретизированным. Реализация будет выглядеть следующим образом:

```
rev([H|T],L1,L2):-
            rev(T,[H|L1],L2). /* голову первого
                                 аргумента дописываем ко
                                 второму аргументу*/
rev([ ],L,L). /* если исходный список закончился,
                 то второй аргумент - передаем в  третий
                 аргумент в качестве результата*/
```

Для того чтобы использовать этот предикат обычным «двухаргументным» образом, добавим еще один предикат, который будет запускать наш «основной» предикат `rev`, имеющий «лишний» аргумент, используемый для накопления элементов обращенного списка. В начале работы второй аргумент должен быть пустым списком:

```
reverse2(L1,L2):-
            rev (L1,[ ],L2).
```

Пример. Создадим предикат, который позволит проверить, является ли список палиндромом. Палиндромом называется список, который совпадает со своим обращением. Соответственно, у данного предиката будет всего один аргумент (список, который проверяем на «палиндромность»).

Первое, что приходит в голову: воспользоваться только что написанным предикатом `reverse` (или `reverse2`). Перевернуть список и проверить, совпадает ли результат с исходным списком. Выглядеть этот предикат будет следующим образом:

```
palindrom(L):-
            reverse (L,L).
```

Можно решить эту задачу «напрямую», без использования предиката `reverse`.

Пример. Напишем предикат, позволяющий получать элемент списка по его номеру так же, как по номеру можно получать элемент массива в императивных языках программирования. Предикат будет трехаргументный: первый аргумент — исходный список, второй аргумент — номер элемента и третий — элемент списка, указанного в качестве первого аргумента предиката, имеющий номер, указанный в качестве второго аргумента.

Решение проведем рекурсией по номеру элемента. В качестве базиса возьмем очевидный факт, что первым элементом списка является его голова. Шаг рекурсии позволит нам сделать предположение, что `N`-й элемент списка является `(N-1)`-м элементом хвоста. Данному определению будет соответствовать следующее предложение:

```
n_element([X|_],1,X).
n_element([_|L],N,Y):-
                N1=N-1,
                n_element(L,N1,Y).
```

Пример. В большинстве практических задач не обойтись без предиката, удаляющего все вхождения заданного значения из списка. Предикат будет зависеть от трех параметров. Первый параметр будет соответствовать удаляемому списку, второй — исходному значению, а третий — результату удаления из первого параметра всех вхождений второго параметра. Создадим его.

Без рекурсии не обойдется и на этот раз. Если первый элемент окажется удаляемым, то нужно перейти к удалению заданного значения из хвоста списка. Результатом в данном случае должен стать список, полученный путем удаления всех вхождений искомого значения из хвоста первоначального списка. Это даст нам базис рекурсии. Шаг рекурсии будет основан на том, что если первый элемент списка не совпадает с тем, который нужно удалять, то он должен остаться первым элементом результата, и нужно переходить к удалению заданного значения из хвоста исходного списка. Полученный в результате этих удалений список должен войти в ответ в качестве хвоста:

```
delete_all(_,[],[]).
delete_all(X,[X|L],L1):-
                  delete_all (X,L,L1).
delete_all (X,[Y|L],[Y|L1]):-
                  X<>Y,
                  delete_all (X,L,L1).
```

Если нам нужно удалить не все вхождения определенного значения в список, а только первое, то следует немного изменить вышеописанную процедуру. Это можно сделать несколькими способами. Рассмотрим один из них.

Заменим в первом правиле рекурсивный вызов предиката отсечением. В этом случае, пока первый элемент списка не окажется удаляемым, мы будем переходить к рассмотрению хвоста:

```
delete_one(_,[],[]).
delete_one(X,[X|L],L):-!.
delete_one(X,[Y|L],[Y|L1]):-
                  delete_one(X,L,L1).
```

В заключение лекции рассмотрим предикат `findall`, предназначенный для нахождения всех решений некоторой цели. У него три параметра: имя переменной, предикат и список, в который будут помещены найденные решения.

Пример. Посмотрим, как с помощью предиката `findall` можно решать задачи, подобные тем, которые мы решали в предыдущей лекции.

Найдем имена всех дочек: `findall(N,mother(_,N),L)`. В список `L` попадут имена всех дочек.

Найдем имена всех дочек Даши: `findall(N,mother("Даша",N),L)`. В список `L` попадут имена всех дочек Даши.

Лекция 8. Сортировка списков

Рассматривается нахождение суммы элементов списка, среднего и минимального значений; алгоритмы сортировки списков: пузырьковый, выбором, вставкой, слиянием, быстрая сортировка.

Ключевые слова: сумма элементов списка, среднее арифметическое элементов списка, минимальный элемент списка, пузырьковая сортировка, сортировка выбором, сортировка вставкой, быстрая сортировка, сортировка слияниями.

В этой лекции речь пойдет о списках, элементами которых являются числа. Хотя в большинстве задач, которые будут рассматриваться, неважно, к какому домену относятся элементы списка, для определенности будем считать, что это целые числа.

Таким образом, списки, с которыми мы планируем работать, могут быть представлены в разделе описания доменов примерно следующим образом:

```
DOMAINS
listI = integer*
```

Для разминки решим несложный пример.

Пример. Создадим предикат, позволяющий вычислить сумму элементов списка.

Решение будет напоминать подсчет количества элементов списка. Отличаться они будут шагом рекурсии. При подсчете количества элементов нам было неважно, чему равен первый элемент списка, мы просто добавляли единицу к уже подсчитанному количеству элементов хвоста. При вычислении суммы нужно будет учесть значение головы списка.

Так как в пустом списке элементов нет, сумма элементов пустого списка равна нулю. Для вычисления суммы элементов непустого списка нужно к сумме элементов хвоста добавить первый элемент списка. Запишем эту идею:

```
sum([], 0). /* сумма элементов пустого списка
               равна нулю */
sum([H|T], S) :-
          sum(T, S_T), /* S_T – сумма элементов хвоста */
          S = S_T + H. /* S – сумма элементов исходного
                          списка */
```

Попробуйте самостоятельно изменить этот предикат так, чтобы он вычислял не сумму элементов списка, а их произведение.

Еще один вариант данного предиката можно написать, используя накопитель. В нем будем хранить уже насчитанную сумму. Начинаем с пустым накопителем. Переходя от вычисления суммы непустого списка к вычислению суммы элементов его хвоста, будем добавлять первый элемент к уже насчитанной сумме. Когда элементы списка будут исчерпаны (список опустеет), передадим «накопленную» сумму в качестве результата в третий аргумент.

Запишем:

```
sum_list([],S,S). /* список стал пустым, значит
                     в накопителе — сумма элементов
                     списка */
sum_list([H|T],N,S) :-
             N_T=H+N,
                  /* N_T — результат добавления к сумме,
                     находящейся в накопителе, первого
                     элемента списка */
                sum_list(T,N_T,S).
                  /* вызываем предикат от хвоста T и N_T */
```

Если нам нужно вызвать предикат от двух аргументов, а не от трех, то можно добавить вспомогательный предикат:

```
sum2(L,S):-
        sum_list(L,0,S).
```

Последний вариант, в отличие от первого, реализует хвостовую рекурсию.

Разберем еще один простой пример.

Пример. Напишем предикат, вычисляющий среднее арифметическое элементов списка.

Конечно, можно, как всегда, опереться на рекурсию, но проще воспользоваться тем, что у нас уже есть предикаты, которые позволяют вычислить количество и сумму элементов списка. Для нахождения среднего нам достаточно будет сумму элементов списка разделить на их количество. Это можно записать следующим образом:

```
avg(L,A):-
        summa(L,S), /* помещаем в переменную S сумму
                       элементов списка */
```

```
        length(L,K), /* переменная K равна количеству
                        элементов списка */
        A=S/K. /* вычисляем среднее как отношение суммы
                  к количеству */
```

Единственная проблема возникает при попытке найти среднее арифметическое элементов пустого списка. Если мы попытаемся вызвать цель `avg([],A)`, то получим сообщение об ошибке «Division by zero» («Деление на ноль»). Это произойдет, потому что предикат `length([],K)` конкретизирует переменную `K` нулем, а при попытке достижения третьей подцели `A=S/K` и произойдет вышеупомянутая ошибка. Можно посчитать это нормальной реакцией предиката. Раз в пустом списке нет элементов, значит, нет и их среднего арифметического. А можно изменить этот предикат так, чтобы он работал и с пустым списком.

Дабы обойти затруднение с пустым списком, добавим в нашу процедуру, в виде факта, информацию о том, что среднее арифметическое элементов пустого списка равно нулю. Полное решение будет выглядеть следующим образом:

```
avg([],0):-!.
avg(L,A):-
        summa(L,S),
        length(L,K),
        A=S/K.
```

Описывая этот предикат в разделе описания предикатов *PREDICATES*, обратите внимание на то, что второй аргумент будет не целого типа, а вещественного (при делении одного целого числа на другое целое число частное может получиться нецелым).

Пример. Создадим предикат, находящий минимальный элемент списка.

Как обычно, наше решение будет рекурсивным. Но так как для пустого списка понятие минимального элемента не имеет смысла, базис рекурсии мы запишем не для пустого, а для одноэлементного списка. В одноэлементном списке, естественно, минимальным элементом будет тот самый единственный элемент списка («при всем богатстве выбора другой альтернативы нет!»).

Шаг рекурсии: найдем минимум из первого элемента списка и минимального элемента хвоста — это и будет минимальный элемент всего списка.

Оформим эти рассуждения:

```
min_list([X],X). /* единственный элемент одноэлементного
                    списка является минимальным элементом
                    списка */
min_list([H|T],M):-
   min_list(T,M_T), /* M_T – минимальный элемент хвоста */
   min(H,M_T,M). /* M – минимум из M_T и первого элемента
                    исходного списка */
```

Обратите внимание на то, что в правиле, позволяющем осуществить шаг рекурсии, использован предикат `min`, подобный предикату `max`, который был разобран нами в третьей лекции.

Слегка модифицировав предикат `min_list` (подставив в правило вместо предиката `min` предикат `max` и поменяв его название), получим предикат, находящий не минимальный, а максимальный элемент списка.

Перейдем теперь к более интересной задаче, а именно, к сортировке списков. Под сортировкой обычно понимают расстановку элементов в некотором порядке. Для определенности мы будем упорядочивать элементы списков по неубыванию. То есть, если сравнить любые два соседних элемента списка, то следующий должен быть не меньше предыдущего.

Существует множество алгоритмов сортировки. Заметим, что имеется два класса алгоритмов сортировки: сортировка данных, целиком расположенных в основной памяти (*внутренняя сортировка*), и сортировка файлов, хранящихся во внешней памяти (*внешняя сортировка*). Мы займемся исключительно методами внутренней сортировки.

Рассмотрим наиболее известные методы внутренней сортировки и выясним, как можно применить их для сортировки списков в Прологе.

Начнем с наиболее известного *«пузырькового»* способа сортировки. Его еще называют методом *прямого обмена* или методом *простого обмена*.

Пузырьковая сортировка

Идея этого метода заключается в следующем. На каждом шаге сравниваются два соседних элемента списка. Если оказывается, что они стоят неправильно, то есть предыдущий элемент меньше следующего, то они меняются местами. Этот процесс продолжаем до тех пор, пока есть пары соседних элементов, расположенные в неправильном порядке. Это и будет означать, что список отсортирован.

Аналогия с пузырьком вызвана тем, что при каждом проходе минимальные элементы как бы «всплывают» к началу списка.

Реализуем пузырьковую сортировку посредством двух предикатов. Один из них, назовем его `permutation`, будет сравнивать два первых эле-

мента списка и в случае, если первый окажется больше второго, менять их местами. Если же первая пара расположена в правильном порядке, этот предикат будет переходить к рассмотрению хвоста.

Основной предикат bubble будет осуществлять пузырьковую сортировку списка, используя вспомогательный предикат permutation.

```
permutation([X,Y|T],[Y,X|T]):-
                        X>Y,!.
                           /* переставляем первые два
                              элемента, если первый больше
                              второго */
permutation([X|T],[X|T1]):-
                        permutation(T,T1).
                           /*переходим к перестановкам
                             в хвосте*/
bubble(L,L1):-
     permutation(L,LL), /* вызываем предикат,
                           осуществляющий перестановку */
     !,
     bubble(LL,L1). /* пытаемся еще раз отсортировать
                       полученный список */
bubble(L,L). /* если перестановок не было, значит, список
                отсортирован */
```

Но наш пузырьковый метод работает только до тех пор, пока есть хотя бы пара элементов списка, расположенных в неправильном порядке. Как только такие элементы закончились, предикат permutation терпит неудачу, а bubble переходит от правила к факту и возвращает в качестве второго аргумента отсортированный список.

Сортировка вставкой

Теперь рассмотрим сортировку вставкой. Она основана на том, что если хвост списка уже отсортирован, то достаточно поставить первый элемент списка на его место в хвосте, и весь список будет отсортирован. При реализации этой идеи создадим два предиката.

Задача предиката insert — вставить значение (голову исходного списка) в уже отсортированный список (хвост исходного списка), так чтобы он остался упорядоченным. Его первым аргументом будет вставляемое значение, вторым — отсортированный список, третьим — список, полученный вставкой первого аргумента в нужное место второго аргумента так, чтобы не нарушить порядок.

Предикат ins_sort, собственно, и будет организовывать сортировку исходного списка методом вставок. В качестве первого аргумента ему дают произвольный список, который нужно отсортировать. Вторым аргументом он возвращает список, состоящий из элементов исходного списка, стоящих в правильном порядке:

```
ins_sort([ ],[ ]). /* отсортированный пустой список
                      остается пустым списком */
ins_sort([H|T],L):-
                   ins_sort(T,T_Sort),
                        /* T – хвост исходного списка,
                           T_Sort – отсортированный хвост
                           исходного списка */
                   insert(H,T_Sort,L).
                        /* вставляем H (первый элемент
                           исходного списка)в T_Sort,
                           получаем L (список, состоящий
                           из элементов исходного списка,
                           стоящих по неубыванию) */
insert(X,[],[X]). /* при вставке любого значения в пустой
                     список, получаем одноэлементный
                     список */
insert(X,[H|T],[H|T1]):-
                   X>H,!, /* если вставляемое значение
                             больше головы списка, значит,
                             его нужно вставлять в хвост */
                   insert(X,T,T1).
                        /* вставляем X в хвост T,
                            в результате получаем
                            список T1 */
insert(X,T,[X|T]). /* это предложение (за счет отсечения
                      в предыдущем правиле) выполняется,
                      только если вставляемое значение
                      не больше головы списка T, значит,
                      добавляем его первым элементом
                      в список T */
```

Сортировка выбором

Идея алгоритма сортировки выбором очень проста. В списке находим минимальный элемент (используя предикат min_list, который мы придумали в начале этой лекции). Удаляем его из списка (с помощью пре-

диката delete_one, рассмотренного в предыдущей лекции). Оставшийся список сортируем. Приписываем минимальный элемент в качестве головы к отсортированному списку. Так как этот элемент был меньше всех элементов исходного списка, он будет меньше всех элементов отсортированного списка. И, следовательно, если его поместить в голову отсортированного списка, то порядок не нарушится.

Запишем:

```
choice([ ],[ ]). /* отсортированный пустой список
                    остается пустым списком */
choice(L,[X|T]):- /* приписываем X (минимальный элемент
                     списка L) к отсортированному
                     списку T */
                  min_list(L,X), /* X – минимальный элемент
                                    списка L */
                  delete_one(X,L,L1),
                              /* L1 – результат удаления
                                 первого вхождения
                                 элемента X из списка L */
                  choice(L1,T). /* сортируем список L1,
                                   результат обозначаем T */
```

Быстрая сортировка

Автором так называемой «быстрой» сортировки является Хоар. Он назвал ее быстрой потому, что в общем случае эффективность этого алгоритма довольно высока. Идея метода следующая. Выбирается некоторый «барьерный» элемент, относительно которого мы разбиваем исходный список на два подсписка. В один мы помещаем элементы, меньшие барьерного элемента, во второй — большие либо равные. Каждый из этих списков мы сортируем тем же способом, после чего приписываем к списку тех элементов, которые меньше барьерного, вначале сам барьерный элемент, а затем — список элементов не меньших барьерного. В итоге получаем список, состоящий из элементов, стоящих в правильном порядке.

При воплощении в программный код этой идеи нам, как обычно, понадобится пара предикатов.

Вспомогательный предикат partition будет отвечать за разбиение списка на два подсписка. У него будет четыре аргумента. Первые два элемента — входные: первый — исходный список, второй — барьерный элемент. Третий и четвертый элементы — выходные, соответственно, список элементов исходного списка, которые меньше барьерного, и список, состоящий из элементов, которые не меньше барьерного элемента.

Предикат quick_sort будет реализовывать алгоритм быстрой сортировки Хоара. Он будет состоять из двух предложений. Правило будет осуществлять с помощью предиката partition разделение непустого списка на два подсписка, затем сортировать каждый из этих подсписков рекурсивным вызовом себя самого, после чего, используя предикат conc (созданный нами ранее), конкретизирует второй аргумент списком, получаемым объединением отсортированного первого подсписка и списка, сконструированного из барьерного элемента (головы исходного списка) и отсортированного второго подсписка. Запишем это:

```
quick_sort([],[]). /* отсортированный пустой список
                      остается пустым списком */
quick_sort([H|T],O):-
                partition(T,H,L,G),
                        /* делим список T на L (список
                           элементов меньших барьерного
                           элемента H) и G (список
                           элементов не меньших H) */
                quick_sort(L,L_s),
                        /* список L_s – результат
                           упорядочивания элементов
                           списка L */
                quick_sort(G,G_s),
                        /* аналогично, список G_s –
                           результат упорядочивания
                           элементов списка G */
                conc(L_s,[H|G_s],O).
                        /* соединяем список L_s со
                           списком, у которого голова H,
                           а хвост G_s, результат
                           обозначаем O */
partition([],_,[],[]). /*  как бы мы ни делили элементы
                           пустого списка, ничего кроме
                           пустых списков не получим */
partition([X|T],Y,[X|T1],Bs):-
                X<Y,!,
                partition(T,Y,T1,Bs).
                     /* если элемент X меньше барьерного
                        элемента Y,  то мы добавляем его
                        в третий аргумент */
partition([X|T],Y,T1,[X|Bs]):-
```

```
                partition(T,Y,T1,Bs).
                    /* в противном случае дописываем
                       его в четвертый аргумент */
```

Прежде чем перейти к изучению следующего алгоритма сортировки, решим одну вспомогательную задачу.

Пусть у нас есть два упорядоченных списка, и мы хотим объединить их элементы в один список так, чтобы объединенный список также остался отсортированным.

Идея реализации предиката, осуществляющего слияние двух отсортированных списков с сохранением порядка, довольно проста. Будем на каждом шаге сравнивать головы наших упорядоченных списков и ту из них, которая меньше, будем переписывать в результирующий список. И так до тех пор, пока один из списков не закончится. Когда один из списков опустеет, нам останется дописать остатки непустого списка к уже построенному итогу. В результате получим список, состоящий из элементов двух исходных списков, причем элементы его расположены в нужном нам порядке.

Создадим предикат (назовем его `fusion`), реализующий приведенное описание. Так как мы не знаем, какой из списков опустеет раньше, нам необходимо «гнать» рекурсию сразу по обоим базовым спискам. У нас будет два факта — основания рекурсии, которые будут утверждать, что если мы сливаем некий список с пустым списком, то в итоге получим, естественно, тот же самый список. Причем этот факт имеет место и в случае, когда первый список пуст, и в случае, когда пуст второй список.

Шаг рекурсии нам дадут два правила: первое будет утверждать, что если голова первого списка меньше головы второго списка, то именно голову первого списка и нужно дописать в качестве головы в результирующий список, после чего перейти к слиянию хвоста первого списка со вторым. Результат этого слияния будет хвостом итогового списка. Второе правило, напротив, будет дописывать голову второго списка в качестве головы результирующего списка, сливать первый список с хвостом второго списка. Итог этого слияния будет хвостом объединенного списка:

```
fusion(L1,[ ],L1):-!. /* при слиянии списка L1 с пустым
                         списком получаем список L1 */
fusion([ ],L2,L2):-!. /* при слиянии пустого списка
                         со списком L2 получаем список
                         L2 */
fusion([H1|T1],[H2|T2],[H1|T]):-
                    H1<H2,!,
                       /* если голова первого списка  H1
                          меньше головы второго списка H2 */
```

```
                    fusion(T1, [H2|T2],T).
                      /* сливаем хвост первого списка T1
                         со вторым списком [H2|T2] */
fusion(L1, [H2|T2],[H2|T]):-
                    fusion(L1,T2,T).
                         /* сливаем первый список L1
                            с хвостом второго списка T2 */
```

Теперь можно перейти к изучению алгоритма сортировки слияниями.

Сортировка слияниями

Метод слияний — один из самых «древних» алгоритмов сортировки. Его придумал Джон фон Нейман еще в 1945 году. Идея этого метода заключается в следующем. Разобьем список, который нужно упорядочить, на два подсписка. Упорядочим каждый из них этим же методом, после чего сольем упорядоченные подсписки обратно в один общий список.

Для начала создадим предикат, который будет делить исходный список на два. Он будет состоять из двух фактов и правила. Первый факт будет утверждать, что пустой список можно разбить только на два пустых подсписка. Второй факт будет предлагать разбиение одноэлементного списка на тот же одноэлементный список и пустой список. Правило будет работать в случаях, не охваченных фактами, т.е. когда упорядочиваемый список содержит не менее двух элементов. В этой ситуации мы будем отправлять первый элемент списка в первый подсписок, второй элемент — во второй подсписок, и продолжать распределять элементы хвоста исходного списка:

```
splitting([],[],[])./* пустой список можно расщепить
                       только на пустые подсписки */
splitting([H],[H],[]). /*  одноэлементный список разбиваем
                           на одноэлементный список
                           и пустой список */
splitting([H1,H2|T],[H1|T1],[H2|T2]):-
                splitting(T,T1,T2).
                   /* элемент H1 отправляем в первый
                      подсписок, элемент H2 — во второй
                      подсписок, хвост T разбиваем
                      на подсписки T1 и T2 */
```

Теперь можно приступать к записи основного предиката, который, собственно, и будет осуществлять сортировку списка. Он будет состоять

из трех предложений. Первое будет декларировать очевидный факт, что при сортировке пустого списка получается пустой список. Второе утверждает, что одноэлементный список также уже является упорядоченным. В третьем правиле будет содержаться суть метода сортировки слиянием. Вначале список расщепляется на два подсписка с помощью предиката `splitting`, затем каждый из них сортируется рекурсивным вызовом предиката `fusion_sort`, и, наконец, используя предикат `fusion`, сливаем полученные упорядоченные подсписки в один список, который и будет результатом упорядочивания элементов исходного списка.

Запишем изложенные выше соображения.

```
fusion_sort([],[]):-!./* отсортированный пустой список
                       остается пустым списком */
fusion_sort([H],[H]):-!. /* одноэлементный список
                          упорядочен */
fusion_sort(L,L_s):-
                splitting(L,L1,L2),
                          /* расщепляем список L на два
                             подсписка */
                fusion_sort(L1,L1_s),
                          /* L1_s – результат сортировки
                             L1 */
                fusion_sort(L2,L2_s),
                          /* L2_s – результат сортировки
                             L2 */
                fusion(L1_s,L2_s,L_s).
                          /* сливаем L1_s и L2_s
                             в список L_s */
```

Фактически этот алгоритм при прямом проходе дробит список на одноэлементные подсписки, после чего на обратном ходе рекурсии сливает их двухэлементные списки. На следующем этапе сливаются двухэлементные списки и т.д. На последнем шаге два подсписка сливаются в итоговый, отсортированный список.

В качестве завершения темы сортировки разработаем предикат, который будет проверять, является ли список упорядоченным. Это совсем не сложно. Для того чтобы список был упорядоченным, он должен быть либо пустым, либо одноэлементным, либо любые два его соседних элемента должны быть расположены в правильном порядке. Запишем эти рассуждения:

```
sorted([ ]). /* пустой список отсортирован */
sorted([_])./* одноэлементный список упорядочен */
sorted([X,Y|T]):-
                X<=Y,
                sorted([Y|T]).
                     /* список упорядочен, если первые
                        два элемента расположены
                        в правильном порядке и список,
                        образованный вторым элементом
                        и хвостом исходного, упорядочен */
```

Лекция 9. Множества

Реализация множеств в Прологе. Операции над множествами: превращение списка во множество, принадлежность элемента множеству, объединение, пересечение, разность, включение, дополнение.

Ключевые слова: множество, мощность множества, принадлежность элемента множеству, теоретико-множественные операции: объединение, пересечение, разность, принадлежность элемента множеству, включение двух множеств, дополнение множества.

В данной лекции мы попробуем реализовать некоторое приближение математического понятия «множество» в Прологе. Заметим, что в Прологе, в отличие от некоторых императивных языков программирования, нет такой встроенной структуры данных, как множество. И, значит, нам придется реализовывать это понятие, опираясь на имеющиеся стандартные домены. В качестве базового домена используем стандартный списковый домен, с которым мы работали на протяжении двух последних лекций.

Итак, что мы будем понимать под множеством? Просто список, который не содержит повторных вхождений элементов. Другими словами, в нашем множестве любое значение не может встречаться более одного раза.

На самом деле, я не знаю ни одной реализации понятия множества, которая бы достаточно точно соответствовала этому математическому объекту. Наше подобие множества также, по большому счету, лишь отчасти будет приближаться к «настоящему» множеству.

Нам предстоит разработать предикаты, которые реализуют основные теоретико-множественные операции.

Начнем с написания предиката, превращающего произвольный список во множество, в том смысле, в котором мы договорились понимать этот термин. Для этого нужно удалить все повторные вхождения элементов. При этом мы воспользуемся предикатом `delete_all`, который был создан нами ранее, в седьмой лекции. Предикат будет иметь два аргумента: первый — исходный список (возможно, содержащий повторные вхождения элементов), второй — выходной (то, что остается от первого аргумента после удаления повторных вхождений элементов).

Предикат будет реализован посредством рекурсии. Базисом рекурсии является очевидный факт: в пустом списке никакой элемент не встречается более одного раза. По правде говоря, в пустом списке нет ни одного элемента, который встречался бы в нем хотя бы один раз, то есть в нем вообще нет элементов. Шаг рекурсии позволит выполнить правило: чтобы сделать из непустого списка множество (в нашем понимании этого по-

нятия), нужно удалить из хвоста списка все вхождения первого элемента списка, если таковые вдруг обнаружатся. После выполнения этой операции первый элемент гарантированно будет встречаться в списке ровно один раз. Для того чтобы превратить во множество весь список, остается превратить во множество хвост исходного списка. Для этого нужно только рекурсивно применить к хвосту исходного списка наш предикат, удаляющий повторные вхождения элементов. Полученный в результате из хвоста список с приписанным в качестве головы первым элементом и будет требуемым результатом (множеством, т.е. списком, образованным элементами исходного списка и не содержащим повторных вхождений элементов).

Закодируем наши рассуждения:

```
list_set([],[]). /* пустой список является списком
                    в нашем понимании */
list_set ([H|T],[H|T1]) :-
          delete_all(H,T,T2),
                     /* T2 – результат удаления
                        вхождений первого элемента
                        исходного списка H из хвоста T */
          list_set (T2,T1).
                     /* T1 – результат удаления
                        повторных вхождений элементов
                        из списка T2 */
```

Например, если применить этот предикат к списку `[1,2,1,2,3,2,1]`, то результатом будет список `[1,2,3]`.

Заметим, что в предикате, обратном только что записанному предикату `list_set` и переводящем множество в список, нет никакой необходимости по той причине, что наше множество уже является списком.

Теперь займемся реализацией теоретико-множественных операций, таких как принадлежность элемента множеству, объединение, пересечение, разность множеств и т.д.

При реализации этих предикатов можно было бы воспользоваться предикатами, предназначенными для работы с обыкновенными списками, но в результате их применения могут получаться списки, содержащие некоторые элементы несколько раз, даже если исходные списки были множествами, в нашем понимании этого слова.

Можно, конечно, после каждого применения теоретико-множественной операции превращать полученный список обратно во множество применением вышеописанного предиката `list_set`, но это было бы не очень удобно. Вместо этого мы попробуем написать каждую из теоре-

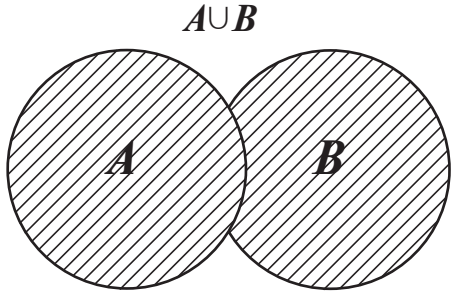
тико-множественных операций так, чтобы в результате ее работы гарантированно получалось множество.

Итак, приступим.

В качестве реализации операции *принадлежности* элемента множеству вполне можно использовать предикат `member3`, который мы разработали в седьмой лекции, когда только начинали знакомиться со списками. Напомним, что факт принадлежности элемента x множеству A в математике принято обозначать следующим образом: `x∈A`.

Для того чтобы найти *мощность* множества, вполне подойдет предикат *length*, рассмотренный нами в седьмой лекции. Напомним, что для конечного множества мощность — это количество элементов во множестве.

Пример. Реализуем операцию объединения двух множеств. На всякий случай напомним, что под *объединением* двух множеств понимают множество, элементы которого принадлежат или первому, или второму множеству. Обозначается объединение множеств `A` и `B` через `A∪B`. В математической записи это выглядит следующим образом: `A∪B={x | x∈A` или `x∈B}`. На рисунке объединение множеств `A` и `B` обозначено штриховкой.



У соответствующего этой операции предиката должно быть три параметра: первые два — множества, которые нужно объединить, третий параметр — результат объединения двух первых аргументов. В третий аргумент должны попасть все элементы, которые входили в первое или второе множество. При этом нам нужно проследить, чтобы ни одно значение не входило в итоговое множество несколько раз. Такое могло бы произойти, если бы мы попытались, например, воспользоваться предикатом `conc` (который мы рассмотрели в седьмой лекции), предназначенным для объединения списков. Если бы какое-то значение встречалось и в первом, и во втором списках, то в результирующий список оно бы попало, по крайней мере, в двойном количестве. Значит, вместо использования предиката `conc` нужно написать новый предикат, применение которого не приведет к ситуации, в которой итоговый список уже не будет множеством за счет того, что некоторые значения будут встречаться в нем более одного раза.

Без рекурсии мы не обойдемся и здесь. Будем вести рекурсию по первому из объединяемых множеств. Базис индукции: объединяем пустое множество с некоторым множеством. Результатом объединения будет второе множество. Шаг рекурсии будет реализован посредством двух правил. Правил получается два, потому что возможны две ситуации: первая — голова первого множества является элементом второго множества, вторая — первый элемент первого множества не входит во второе множество. В первом случае мы не будем добавлять голову первого множества в результирующее множество, она попадет туда из второго множества. Во втором случае ничто не мешает нам добавить первый элемент первого списка. Так как этого значения во втором множестве нет, и в хвосте первого множества оно также не может встречаться (иначе это было бы не множество), то и в результирующем множестве оно также будет встречаться только один раз.

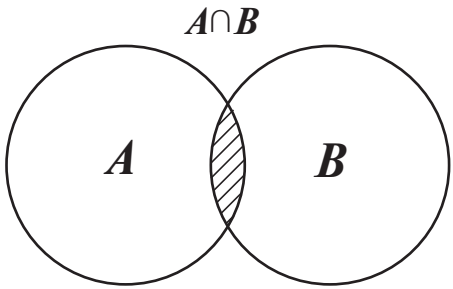
Давайте запишем эти рассуждения:

```
union([ ],S2,S2). /* результатом объединения пустого
                     множества со множеством S2 будет
                     множество S2 */
union([H|T],S2,S):-
                member3(H,S2),
                     /* если голова первого
                        множества H принадлежит второму
                        множеству S2, */
                !,
                union(T,S2,S).
                     /* то результатом S будет объединение
                        хвоста первого множества T
                        и второго множества S2 */
union([H|T],S2,[H|S]):-
                union(T,S2,S).
                     /* в противном случае результатом
                        будет множество, образованное
                        головой первого множества H
                        и хвостом, полученным объединением
                        хвоста первого множества T
                        и второго множества S2 */
```

Если объединить множество `[1,2,3,4]` со множеством `[3,4,5]`, то в результате получится множество `[1,2,3,4,5]`.

Пример. Теперь можно приступить к реализации операции пересечения двух множеств. Напомним, что *пересечение* двух множеств — это мно-

жество, образованное элементами, которые одновременно принадлежат и первому, и второму множествам. Обозначается пересечение множеств A и B через A∩B. В математических обозначениях это выглядит следующим образом: A∩B={x|x∈A *и* x∈B}. На рисунке пересечение множеств A и B обозначено штриховкой.



У предиката, реализующего эту операцию, как и у предиката, осуществляющего объединение двух множеств, есть три параметра: первые два — исходные множества, третий — результат пересечения двух первых аргументов. В итоговом множестве должны оказаться те элементы, которые входят и в первое, и во второе множество одновременно.

Этот предикат, наверное, будет немного проще объединения. Его мы также проведем рекурсией по первому множеству. Базис рекурсии: пересечение пустого множества с любым множеством будет пустым множеством. Шаг рекурсии так же, как и в случае объединения, разбивается на два случая в зависимости от того, принадлежит ли первый элемент первого множества второму. В ситуации, когда голова первого множества является элементом второго множества, пересечение множеств получается приписыванием головы первого множества к пересечению хвоста первого множества со вторым множеством. В случае, когда первый элемент первого множества не встречается во втором множестве, результирующее множество получается пересечением хвоста первого множества со вторым множеством.

Запишем это:

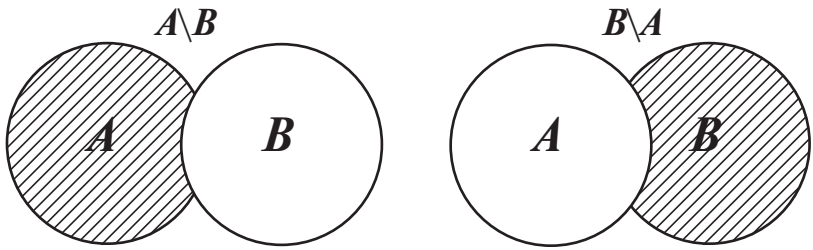
```
intersection([],_,[]). /*  в результате пересечения
                           пустого множества с любым
                           множеством получается пустое
                           множество */
intersection([H|T1],S2,[H|T]):-
                    member3(H,S2),
                       /* если голова первого множества H
                          принадлежит второму множеству S2 */
```

```
                    !,
                    intersection(T1,S2,T).
                       /* то результатом будет множество,
                          образованное головой первого
                          множества H и хвостом, полученным
                          пресечением хвоста первого
                          множества T1 со вторым
                          множеством S2 */
intersection([_|T],S2,S):-
                    intersection(T,S2,S).
                       /* в противном случае результатом
                          будет множество S, полученное
                          объединением хвоста первого
                          множества T со вторым
                          множеством S2 */
```

Если пересечь множество `[1,2,3,4]` со множеством `[3,4,5]`, то в результате получится множество `[3,4]`.

Пример. Следующая операция, которую стоит реализовать, — это разность двух множеств. Напомним, что *разность* двух множеств — это множество, образованное элементами первого множества, не принадлежащими второму множеству. Обозначается разность множеств `A` и `B` через `A-B` или `A\B`. В математических обозначениях это выглядит следующим образом: `A\B={x|x∈A и x∉B}`.

На рисунках разность множеств `A` и `B` (`B` и `A`) обозначена штриховкой.



В этой операции, в отличие от двух предыдущих, важен порядок множеств. Если в объединении или пересечении множеств поменять первый и второй аргументы местами, результат останется прежним. В то время как при `A={1,2,3,4}, B={3,4,5}, A\B={1,2}`, но `B\A={5}`.

У предиката, реализующего разность, как и у объединения и пересечения, будет три аргумента: первый — множество, из которого нужно вычесть, второй — множество, которое нужно отнять, третий — результат

вычитания второго аргумента из первого. В третий параметр должны попасть те элементы первого множества, которые не принадлежат второму множеству.

Рекурсия по первому множеству поможет нам реализовать вычитание. В качестве базиса рекурсии возьмем очевидный факт: при вычитании произвольного множества из пустого множества ничего кроме пустого множества получиться не может, так как в пустом множестве элементов нет. Шаг рекурсии, как и в случае объединения и пересечения, зависит от того, принадлежит ли первый элемент множества, из которого вычитают, множеству, которое вычитают. В случае, когда голова первого множества является элементом второго множества, разность множеств получается путем вычитания второго множества из хвоста первого. Когда первый элемент множества, из которого производится вычитание, не встречается в вычитаемом множестве, ответом будет множество, образованное приписыванием головы первого множества к результату вычитания второго множества из хвоста первого множества.

Запишем эти рассуждения:

```
minus([],_,[]). /* при вычитании любого множества
                   из пустого множества получится пустое
                   множество */
minus([H|T],S2,S):-
                   member3(H,S2),
                      /* если первый элемент  первого
                         множества H принадлежит второму
                         множеству S2*/
                   !,
                   minus(T,S2,S).
                      /* то результатом S будет разность
                         хвоста первого множества T
                         и второго множества S2 */
minus([H|T],S2,[H|S]):-
                         minus(T,S2,S).
                      /* в противном случае, результатом
                         будет множество, образованное
                         первым элементом первого
                         множества H и хвостом, полученным
                         вычитанием из хвоста первого
                         множества T второго множества S2 */
```

Можно попробовать реализовать пересечение через разность. Из математики нам известно тождество `A∩B=A\(A\B)`. Попробуем проверить

это тождество, записав соответствующий предикат, реализующий пересечение множеств, через взятие разности.

```
intersection2(A,B,S):-
                minus(A,B,A_B), /*A_B=A\B */
                minus(A,A_B,S). /* S = A\A_B = A\(A\B) */
```

Проверка на примерах показывает, что этот предикат, так же, как, впрочем, и ранее созданный предикат `intersection`, возвращает именно те результаты, которые ожидаются.

Пример. Не помешает иметь предикат, позволяющий проверить, является ли одно множество *подмножеством* другого. В каком случае одно множество *содержится* в другом? В случае, если каждый элемент первого множества принадлежит второму множеству. Тот факт, что множество `A` является подмножеством множества `B`, обозначается через `A⊆B`. В математической записи это выглядит следующим образом: `A⊆B⇔∀x(x∈A → x∈B)`.

Предикат, реализующий данное отношение, будет иметь два параметра, оба – входные. В качестве первого параметра будем указывать множество, включение которого мы хотим проверить. То множество, включение в которое первого аргумента нужно проверить, указывается в качестве второго параметра.

Решение, как обычно, будет рекурсивным. Базис рекурсии будет представлен фактом, утверждающим, что пустое множество является подмножеством любого множества. Шаг рекурсии: чтобы одно множество было подмножеством другого, нужно, чтобы его первый элемент принадлежал второму множеству (проверить это нам позволит предикат `member3`, рассмотренный нами ранее в седьмой лекции), а его хвост, в свою очередь, должен быть подмножеством второго множества. Этих рассуждений достаточно, чтобы записать предикат, реализующий операцию включения:

```
subset([],_). /* пустое множество является подмножеством
                 любого множества */
subset([H|T],S):- /* множество [H|T] является
                     подмножеством множества S */
              member3(H,S), /* если его первый элемент H
                               принадлежит S */
              subset(T,S). /* и его хвост T является
                              подмножеством множества S */
```

Можно также определить это отношение, воспользовавшись уже определенными предикатами `union` и `intersection`.

Из математики известно, что A⊆B⇔A∪B=B. То есть одно множество является подмножеством другого тогда и только тогда, когда их объединение совпадает со вторым множеством. Или, аналогично, A⊆B⇔A∩B=A. То есть одно множество является подмножеством другого тогда и только тогда, когда их пересечение совпадает с первым множеством.

Запишем эти математические соотношения на Прологе:

```
subsetU(A,B):-
          union(A,B,B). /* объединение множеств совпадает
                           со вторым множеством */
subsetI(A,B):-
          intersection(A,B,A).
                      /* пересечение множеств
                         совпадает с первым множеством*/
```

Проверка на примерах показывает, что оба предиката, как и ранее созданный предикат subset, возвращают именно те результаты, какие и должны возвращать.

Используя только что написанный предикат, реализующий отношение включения множеств, можно создать предикат, осуществляющий проверку совпадения двух множеств. Напомним, что два множества A и B называются равными, если одновременно выполнено A⊆B и B⊆A, т.е. множество A содержится во множестве B и множество B содержится во множестве A. Другими словами, два множества равны, если все элементы первого множества содержатся во втором множестве, и наоборот. Отсюда следует, что эти множества состоят из одних и тех же элементов.

Напишем предикат, реализующий отношение равенства двух множеств:

```
equal(A,B):- /* множество A совпадает со множеством B, */
          subset(A,B), /* если множество A содержится
                          во множестве B */
          subset(B,A). /* и множество B является
                          подмножеством множества A*/
```

Убедимся, что множество [1,2,3] и множество [3,4,5] не равны, а множества [1,2,3] и [2,1,3] совпадают.

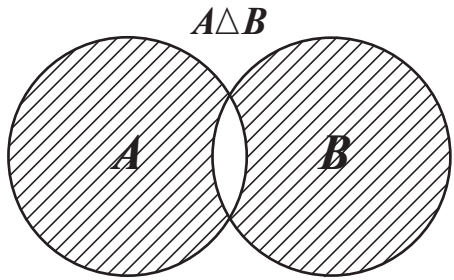
Если множество A содержится во множестве B, причем во множестве B имеются элементы, не принадлежащие множеству A, то говорят, что A — собственное подмножество множества B. Обозначается этот факт как A⊂B.

Закодируем это отношение:

```
Prop_subset(A,B):-
            subset(A,B),
            /* множество A содержится во множестве B */
            not(equal(A,B)).
            /* множества A и B не совпадают*/
```

Проверим, что множество [1,3] является собственным подмножеством множества [1,2,3], в отличие от множеств [1,4] и [2,1,3].

Пример. Рассмотрим еще одну операцию на множествах. Она называется симметрическая разность и, как видно из ее названия, в отличие от обычной разности, не зависит от порядка ее аргументов. *Симметрической разностью* двух множеств называется множество, чьи элементы либо принадлежат первому и не принадлежат второму множеству, либо принадлежат второму и не принадлежат первому множеству. Она не столь известна, как предыдущие рассмотренные нами операции, однако тоже имеет право на существование. Обозначается симметрическая разность множеств A и B через A△B. В математических обозначениях это выглядит следующим образом: A△B={x|(x∈A и x∉B) или (x∈B и x∉A)}. В отличие от обычной разности, в симметрической разности, если поменять аргументы местами, результат останется неизменным (A△B=B△A).



Например, при A={1,2,3,4}, B={3,4,5}, A△B=B△A={1,2,5}.

Воспользуемся тем, что симметрическую разность можно выразить через уже реализованные нами операции. А именно, A△B=(A\B)∪(B\A). Словесно эта формула читается так: симметрическая разность двух множеств есть разность первого и второго множеств, объединенная с разностью второго и первого множеств.

Запишем это на Прологе:

```
Sim_minus(A,B,SM):-
            minus(A,B,A_B), /* A_B – это разность
                               множеств A и B */
```

```
        minus(B,A,B_A), /* B_A — это разность
                           множеств B и A */
        union(A_B,B_A,SM). /* SM — это объединение
                              множеств A_B и B_A */
```

Убедимся, что симметрическая разность множеств `[1,2,3,4]` и `[3,4,5]` равна множеству `[1,2,5]`, а симметрическая разность множеств `[3,4,5]` и `[1,2,3,4]` равна множеству `[5,1,2]`. Множество `[1,2,5]` с точностью до порядка элементов совпадает с множеством `[5,1,2]`. Таким образом, мы выяснили, что результат не зависит от порядка аргументов.

Пример. Еще одна операция, которую обычно используют при работе со множествами, это дополнение. *Дополнением* множества обычно называется множество, чьи элементы не принадлежат исходному множеству. Обозначается дополнение множества `A` через $\overline{A}$. В математических обозначениях это выглядит следующим образом: $\overline{A}=\{x/x\notin A\}$. Обычно имеет смысл говорить о дополнении только в ситуации, когда имеется некоторое *универсальное множество*, т.е. множество, которому принадлежат все рассматриваемые элементы. Оно может зависеть от решаемой задачи. Например, в качестве такого множества может выступать множество натуральных чисел, множество русских букв, множество символов, обозначающих арифметические действия и т.д.

Давайте, для определенности, возьмем в качестве универсального множества множество цифр (`{0,1,2,3,4,5,6,7,8,9}`). Напишем дополнение над этим универсальным множеством.

Воспользуемся при этом очередным тождеством, которое известно в математике. А именно, тем, что $\overline{A}=U\setminus A$, где символ `U` обозначает универсальное множество. Операция разности двух множеств у нас уже реализована.

Закодируем вышеприведенную формулу на Прологе:

```
supp(A,D):-
        U=[0,1,2,3,4,5,6,7,8,9],
        minus(U,A,D). /* D — это разность универсального
                         множества U и множества A */
```

Проверяем, что дополнение множества `[1,2,3,4]` равно множеству `[0,5,6,7,8,9]`.

Имея дополнение, можно выразить операцию объединения через пересечение и дополнение, или, наоборот, операцию пересечения через объединение и дополнение, используя законы де Моргана ($A\cup B=\overline{\overline{A}\cap\overline{B}}$ и $A\cap B=\overline{\overline{A}\cup\overline{B}}$).

Запишем эти соотношения на Прологе:

```
unionI(A,B,AB):-
          supp(A,A_), /* A_ — это дополнение
                         множества A */
          supp(B,B_), /* B_ — это дополнение
                         множества B */
          intersection(A_,B_,A_B),
                      /* A_B — это пересечение
                         множеств A_ и B_ */
          supp(A_B,AB). /* AB — это дополнение
                           множества A_B */
intersectionU(A,B,AB):-
          supp(A,A_), /* A_ — это дополнение
                         множества A */
          supp(B,B_), /* B_ — это дополнение
                         множества B */
          union(A_,B_,A_B), /* A_B — это объединение
                               множеств A_ и B_ */
          supp(A_B,AB). /* AB — это дополнение
                           множества A_B */
```

Проверка на примерах показывает, что оба предиката работают на множествах, являющихся подмножествами универсального множества (в нашем примере это множество `{0,1,2,3,4,5,6,7,8,9}`), как и ранее созданные предикаты `union` и `intersection`.

Лекция 10. Деревья

Бинарные деревья, двоичные справочники и операции над ними.

Ключевые слова: граф, путь, цикл, корень, предок, потомок, дерево, лист дерева, крона дерева, высота дерева, двоичный справочник.

Данная лекция будет посвящена изучению и реализации на Прологе такой структуры данных, как деревья.

Начнем с маленького введения из теории графов, частным случаем которых являются деревья.

Обычно *графом* называют пару множеств: множество *вершин* и множество *дуг* (множество пар из множества вершин). Различают *ориентированные* и *неориентированные* графы. В ориентированном графе каждая дуга имеет направление (рассматриваются упорядоченные пары вершин). Графически обычно принято изображать вершины графа точками, а связи между ними — линиями, соединяющими точки-вершины.

Путем называется последовательность вершин, соединенных дугами. Для ориентированного графа направление пути должно совпадать с направлением каждой дуги, принадлежащей пути. *Циклом* называется путь, у которого совпадают начало и конец.

Две вершины ориентированного графа, соединенные дугой, называются *отцом* и *сыном* (или *главной* и *подчиненной* вершинами). Известно, что если граф не имеет циклов, то обязательно найдется хотя бы одна вершина, которая не является ничьим сыном. Такую вершину называют *корневой*. Если из одной вершины достижима другая, то первая называется *предком*, вторая — *потомком*.

Деревом называется граф, у которого одна вершина корневая, остальные вершины имеют только одного отца и все вершины являются потомками корневой вершины.

Листом дерева называется его вершина, не имеющая сыновей. *Кроной* дерева называется совокупность всех листьев. *Высотой* дерева называется наибольшая длина пути от корня к листу.

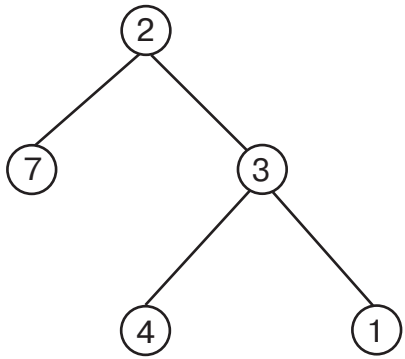
Нам будет удобно использовать следующее рекурсивное определение бинарного дерева: дерево либо пусто, либо состоит из корня, а также левого и правого поддеревьев, которые в свою очередь также являются деревьями.

В вершинах дерева может храниться информация любого типа. Для простоты в этой лекции будем считать, что в вершинах дерева располагаются целые числа. Тогда соответствующее этому определению описание альтернативного домена будет выглядеть следующим образом:

```
DOMAINS
tree=empty;tr(i,tree,tree)
                        /* дерево либо пусто, либо
                           состоит из корня (целого числа),
                           левого и правого поддеревьев,
                           также являющихся деревьями */
```

Заметим, что идентификатор `empty` не является зарезервированным словом Пролога. Вместо него вполне можно употреблять какое-нибудь другое обозначение для пустого дерева. Например, можно использовать для обозначения дерева, не имеющего вершин, идентификатор `nil`, как в Лиспе, или `void`, как в Си. То же самое относится и к имени домена (и имени функтора): вместо `tree (tr)` можно использовать любой другой идентификатор.

Например, дерево



можно задать следующим образом:

```
tr(2,t(7,empty, empty),tr(3,tree(4,empty,empty),
tr(1,empty,empty))).
```

Теперь займемся написанием предикатов для реализации операций на бинарных деревьях.

Пример. Начнем с реализации предиката, который будет проверять принадлежность значения дереву. Предикат будет иметь два аргумента. Первым аргументом будет исходное значение, вторым — дерево, в котором мы ищем данное значение.

Следуя рекурсивному определению дерева, заметим, что некоторое значение принадлежит данному дереву, если оно либо содержится в кор-

не дерева, либо принадлежит левому поддереву, либо принадлежит правому поддереву. Других вариантов нет.

Запишем это рассуждение на Прологе:

```
tree_member(X,tr(X,_,_)):-!. /* X — является корнем
                                дерева */
tree_member(X,tr(_,L,_)):-
            tree_member(X,L),!. /* X принадлежит левому
                                   поддереву */
tree_member(X,tr(_,_,R)):-
            tree_member(X,R). /* X принадлежит правому
                                 поддереву */
```

Пример. Разработаем предикат, который будет заменять в дереве все вхождения одного значения на другое. У предиката будет четыре аргумента: три входных (значение, которое нужно заменять; значение, которым нужно заменять; исходное дерево), четвертым — выходным — аргументом будет дерево, полученное в результате замены всех вхождений первого значения на второе.

Базис рекурсивного решения будет следующий. Из пустого дерева можно получить только пустое дерево. При этом абсолютно неважно, что на что мы заменяем. Шаг рекурсии зависит от того, находится ли заменяемое значение в корне дерева. Если находится, то нужно заменить корневое значение вторым значением, после чего перейти к замене первого значения на второе в левом и правом поддереве. Если же в корне содержится значение, отличное от заменяемого, то оно должно остаться. Замену нужно произвести в левом и правом поддеревьях:

```
tree_replace(_,_,empty,empty). /* пустое дерево остается
                                  пустым деревом*/
tree_replace(X,Y,tr(X,L,R),tr(Y,L1,R1)):-
                          /* корень содержит заменяемое
                             значение X*/
                !,tree_replace(X,Y,L,L1),
                          /* L1 — результат замены
                             в дереве L всех вхождений X
                             на Y */
                tree_replace(X,Y,R,R1).
                          /* R1 — результат замены
                             в дереве R всех вхождений X
                             на Y */
```

```
tree_replace(X,Y,tr(K,L,R),tr(K,L1,R1)):-
                          /* корень не содержит
                             заменяемое значение X */
                tree_replace(X,Y,L,L1),
                          /* L1 – результат замены
                             в дереве L всех вхождений X
                             на Y */
                tree_replace(X,Y,R,R1).
                          /* R1 – результат замены
                             в дереве R всех вхождений X
                             на Y */
```

Пример. Напишем предикат, подсчитывающий общее количество вершин дерева. У него будет два параметра. Первый (входной) параметр — дерево, второй (выходной) — количество вершин в дереве.

Как всегда, пользуемся рекурсией. Базис: в пустом дереве количество вершин равно нулю. Шаг рекурсии: чтобы посчитать количество вершин дерева, нужно посчитать количество вершин в левом и правом поддереве, сложить полученные числа и добавить к результату единицу (посчитать корень дерева).

Пишем:

```
tree_length (empty,0). /* В пустом дереве нет вершин */
tree_length(tr(_,L,R),N):-
                     tree_length (L,N1),
                               /* N1 – число вершин
                                  левого поддерева */
                     tree_length (R,N2),
                               /* N2 – число вершин
                                  правого поддерева */
                     N=N1+N2+1. /* число вершин исходного
                                   дерева получается
                                   сложением N1, N2
                                   и единицы */
```

Пример. Решим еще одну подобную задачу. Разработаем предикат, подсчитывающий не общее количество вершин дерева, а только количество листьев, т.е. вершин, не имеющих сыновей. Предикат будет иметь два параметра. Входной — исходное дерево, выходной — количество листьев дерева, находящегося в первом параметре.

Понятно, что, так как в пустом дереве нет вершин, в нем нет и вершин, являющихся листьями. Это первый базис рекурсии. Второй базис

будет заключаться в очевидном факте, что дерево, состоящее из одной вершины, имеет ровно один лист. Шаг: для того, чтобы посчитать количество листьев дерева, нужно просто сложить количество листьев в левом и правом поддереве.

Запишем:

```
tree_leaves(empty,0). /* в пустом дереве листьев нет */
tree_leaves(tr(_,empty,empty),1):-!.
                          /* в дереве с одним корнем —
                             один лист */
tree_leaves(tr(_,L,R),N):-
                      tree_leaves(L,N1),
                           /* N1 — количество листьев
                              в левом поддереве */
                      tree_leaves(R,N2),
                           /* N2 — количество листьев
                              в правом поддереве */
                      N=N1+N2.
```

Пример. Создадим предикат, находящий сумму чисел, расположенных в вершинах дерева. Он будет иметь два аргумента. Первый — исходный список, второй — сумма чисел, находящихся в вершинах дерева, расположенного в первом аргументе.

Идея реализации будет очень простой и немного похожей на подсчет количества вершин. Базис рекурсии: сумма элементов пустого дерева равна нулю, потому что в пустом дереве нет элементов. Чтобы подсчитать сумму значений, находящихся в вершинах непустого дерева, нужно сложить сумму элементов, хранящихся в левом и правом поддереве, и не забыть добавить корневое значение.

На Прологе это записывается следующим образом:

```
tree_sum (empty,0). /* В пустом дереве вершин нет */
tree_sum(tr(X,L,R),N):-
                    tree_sum (L,N1),
                              /* N1 — сумма элементов
                                 левого поддерева */
                    tree_sum (R,N2),
                              /* N2 — сумма элементов
                                 правого поддерева */
                    N=N1+N2+X. /* складываем N1, N2
                                  и корневое значение */
```

Пример. Создадим предикат, позволяющий вычислить высоту дерева. Напомним, что высота дерева — это наибольшая длина пути от корня дерева до его листа. Предикат будет иметь два параметра. Первый (входной) — дерево, второй (выходной) — высота дерева, помещенного в первый параметр.

Базис рекурсии будет основан на том, что высота пустого дерева равна нулю. Шаг рекурсии — на том, что для подсчета высоты всего дерева нужно найти высоты левого и правого поддеревьев, взять их максимум и добавить единицу (учесть уровень, на котором находится корень дерева). Предикат `max` (или `max2`), вычисляющий максимум из двух элементов, был разработан нами еще в третьей лекции. Мы воспользуемся им при вычислении высоты дерева.

Получается следующее:

```
tree_height(empty,0). /* Высота пустого дерева равна
                         нулю */
tree_height(tr(_,L,R),D) :-
                      tree_height(L,D1),
                         /* D1 – высота левого поддерева */
                      tree_height(R,D2),
                         /* D2 – высота правого поддерева */
                      max(D1,D2,D_M),
                         /* D_M – максимум из высот левого
                            и правого поддеревьев */
                      D=D_M+1.
                         /* D – высота дерева получается
                            путем увеличения числа D_M
                            на единицу*/
```

Существует особый вид бинарных деревьев — так называемые *двоичные справочники*. В двоичном справочнике все значения, входящие в левое поддерево, меньше значения, находящегося в корне, а все значения, расположенные в вершинах правого поддерева, больше корневого значения, а левое и правое поддеревья, в свою очередь, также являются двоичными справочниками. Такие деревья еще называют *упорядоченными слева направо*.

Пример. Усовершенствуем предикат `tree_member` для проверки принадлежности значения двоичному справочнику. Повысить эффективность этого предиката мы сможем, воспользовавшись тем, что если в двоичном справочнике искомое значение не совпадает с тем, которое хранится в корне, то его имеет смысл искать только в левом поддереве, если

оно меньше корневого, и, соответственно, только в правом поддереве, если оно больше корневого значения.

Модифицированный предикат будет выглядеть следующим образом:

```
tree_member2(X,tr(X,_,_)):-!. /* X – корень дерева */
tree_member2(X,tr(K,L,_)):-
                      X<K,!,
                      tree_member2(X,L).
                             /* X – принадлежит
                                левому поддереву */
tree_member2(X,tr(K,_,R)):-
                      X>K,!,
                      tree_member2(X,R).
                             /* X – принадлежит
                                правому поддереву */
```

Пример. Создадим предикат, позволяющий добавить в двоичный справочник новое значение. При этом результирующее дерево должно получиться двоичным деревом. Предикат будет иметь три аргумента. Первым аргументом будет добавляемое значение, вторым — дерево, в которое нужно добавить данное значение, третьим — результат вставки первого аргумента во второй.

Решение, конечно, будет рекурсивным. На чем оно будет основано? Наша рекурсия будет основана на двух базисах и двух правилах. Первый базис: если вставлять любое значение в пустое дерево, то в результате получим дерево, у которого левое и правое поддеревья — пустые, в корне записано добавляемое значение. Второй базис: если вставляемое значение совпадает со значением, находящимся в корневой вершине исходного дерева, то результат не будет отличаться от исходного дерева (в двоичном справочнике все элементы различны). Два правила рекурсии определяют, как нужно действовать, если исходное дерево непустое и его корневое значение отличается от вставляемого значения. В этой ситуации, если добавляемое значение меньше корневого, то его нужно добавить в левое поддерево, иначе — искать ему место в правом поддереве.

Запишем на Прологе реализацию этих рассуждений:

```
tree_insert(X,empty,tr(X,empty,empty)).
                  /* вставляем X в пустое дерево, получаем
                     дерево с X в корневой вершине, пустыми
                     левым и правым поддеревьями */
tree_insert(X,tr(X,L,R),tr(X,L,R)):-!.
                  /* вставляем X в дерево со значением X
```

```
                    в корневой вершине, оставляем исходное
                    дерево без изменений */
tree_insert(X,tr(K,L,R),tr(K,L1,R)):-
                        X<K,!,
                        tree_insert(X,L,L1).
                        /* вставляем X в дерево с большим
                        X элементом в корневой вершине,
                        значит, нужно вставить X в левое
                        поддерево исходного дерева */
tree_insert(X,tr(K,L,R),tr(K,L,R1)):-
                        tree_insert(X,R,R1).
                        /* вставляем X в дерево с меньшим
                        X элементом в корневой вершине,
                        значит, нужно вставить X в правое
                        поддерево исходного дерева */
```

Можно обратить внимание на две особенности работы данного предиката. Во-первых, вершина, содержащая новое значение, будет добавлена в качестве нового листа дерева. Это следует из первого предложения нашей процедуры. Во-вторых, если добавляемое значение уже содержится в нашем двоичном справочнике, то оно не будет добавлено, и дерево останется прежним, без изменений. Это следует из второго предложения процедуры, описывающей наш предикат.

Пример. Создадим предикат, генерирующий дерево, которое является двоичным справочником и состоит из заданного количества вершин, в которых будут размещены случайные целые числа.

Как можно было заметить, записывать деревья вручную довольно сложно. Этот предикат позволит нам автоматически создавать деревья с нужным количеством элементов. В дальнейшем он пригодится для проверки других предикатов, обрабатывающих деревья.

Предикат будет иметь два аргумента. Первый, входной, будет задавать требуемое количество элементов. Второй, выходной, будет равен сгенерированному дереву.

Решение будет, естественно, рекурсивным. Рекурсия по количеству вершин дерева. Базис рекурсии: нулевое количество вершин имеется только в пустом дереве. Если количество вершин должно быть больше нуля, то нужно (с помощью встроенного предиката `random`, рассмотренного в пятой лекции) сгенерировать случайное значение, построить дерево, имеющее вершин на одну меньше, чем итоговое дерево, вставить случайное значение в построенное дерево, воспользовавшись созданным перед этим предикатом `tree_insert`:

```
tree_gen(0,empty):-!. /* ноль вершин соответствует пустому
                         дереву */
tree_gen (N,T):-
                    random(100,X),
                            /* X – случайное число
                               из промежутка [0,100) */
                    N1= N-1,
                    tree_gen (N1,T1),
                            /* T1 – дерево, имеющее
                               N-1 вершин */
                    tree_insert(X,T1,T). /* вставляем X
                                            в дерево T1 */
```

Обратите внимание на то, что на самом деле дерево, сгенерированное этим предикатом, не обязательно будет иметь столько вершин, сколько было указано в первом параметре. Если вспомнить реализацию предиката `tree_insert`, то можно обратить внимание на то, что в ситуации, когда вставляемое значение уже содержится в двоичном справочнике, оно не будет добавлено в дерево. Т.е. всякий раз, когда случайное число, генерируемое встроенным предикатом `random`, уже содержится в некоторой вершине дерева, оно не попадет в дерево, и, следовательно, итоговое дерево будет содержать на одну вершину меньше. Если во время построения двоичного справочника такая ситуация будет возникать несколько раз, то в итоговом дереве будет на соответствующее количество вершин меньше, чем можно было ожидать.

Если нам обязательно нужно по какой-то причине получить дерево, содержащее ровно столько вершин, сколько было указано в первом параметре, нужно модифицировать этот предикат. Это можно сделать несколькими способами.

Первый вариант: можно модифицировать предикат, осуществляющий добавление значения в двоичный справочник так, чтобы в случае, когда вставляемое значение совпадало с корневым значением, генерировалось новое случайное число, после чего еще раз осуществлялась попытка вставки вновь сгенерированного значения в дерево.

Другой вариант: можно поменять местами вызов предикатов `random` и `tree_gen` и после генерации случайного числа проверять с помощью предиката `tree_member2`, не содержится ли это значение в уже построенном дереве. Если его там нет, значит, его можно спокойно вставить в двоичный справочник с помощью предиката `tree_insert`. Если же это значение уже содержится в одной из вершин дерева, значит, нужно сгенерировать новое случайное число, после чего опять проверить его наличие и т.д.

Надо заметить, что если задать требуемое количество вершин дерева, заведомо большее, чем первый аргумент предиката `random` (количество различных случайных чисел, генерируемых этим предикатом), мы получим зацикливание. Например, в приведенном выше примере вызывается предикат `random(100,X)`. Этот предикат будет возвращать целые случайные числа из промежутка от 0 до 99. Различных чисел из этого промежутка всего сто. Следовательно, и справочник, генерируемый с помощью нашего предиката, может содержать не более ста вершин.

Эту проблему можно обойти, если сделать первый аргумент предиката `random` зависящим от заказанного числа вершин дерева (заведомо большим).

Пример. Далее логично заняться предикатом, который будет удалять заданное значение из двоичного справочника. У него будет три параметра. Два входных (удаляемое значение и исходное дерево) и результат удаления первого параметра из второго.

Реализовать этот предикат оказывается не так просто, как хотелось бы. Без особых проблем можно написать базисы рекурсии для случая, когда удаляемое значение является корневым, а левое или правое поддерево пусты. В этом случае результатом будет, соответственно, правое или левое поддерево. Шаг рекурсии для случая, когда значение, содержащееся в корне дерева, отличается от удаляемого значения, также реализуется без проблем. Нам нужно выяснить, меньше удаляемое значение корневого или больше. В первом случае перейти к удалению данного значения из левого поддерева, во втором — к удалению этого значения из правого поддерева. Проблема возникает, когда нам нужно удалить корневую вершину в дереве, у которого и левое, и правое поддеревья не пусты.

Есть несколько вариантов разрешения возникшей проблемы. Один из них заключается в следующем. Можно удалить из правого поддерева минимальный элемент (или из левого дерева максимальный) и заменить им значение, находящееся в корне. Так как любой элемент правого поддерева больше любого элемента левого поддерева, дерево, получившееся в результате такого удаления и замены корневого значения, останется двоичным справочником.

Для удаления из двоичного справочника вершины, содержащей минимальный элемент, нам понадобится отдельный предикат. Его реализация будет состоять из двух предложений. Первое будет задавать базис рекурсии и срабатывать в случае, когда левое поддерево пусто. В этой ситуации минимальным элементом дерева является значение, находящееся в корне, потому что, по определению двоичного справочника, все значения, находящиеся в вершинах правого поддерева, больше значения, находящегося в корневой вершине. И, значит, нам достаточно удалить корень, а результатом будет правое поддерево.

Второе предложение будет задавать шаг рекурсии и выполняться, когда левое поддерево не пусто. В этой ситуации минимальный элемент находится в левом поддереве и его нужно оттуда удалить. Так как минимальное значение нам потребуется, чтобы вставить его в корневую вершину, у этого предиката будет не два аргумента, как можно было бы ожидать, а три. Третий (выходной) аргумент будет нужен нам для возвращения минимального значения.

Запишем оба эти предиката.

Начнем со вспомогательного предиката, удаляющего минимальный элемент двоичного справочника:

```
tree_del_min(tr(X,empty,R), R, X).
                    /* Если левое поддерево пусто,
                       то минимальный элемент – корень,
                       а дерево без минимального
                       элемента – это правое поддерево.*/
tree_del_min(tr(K,L,R), tr(K,L1,R), X):-
                    tree_del_min(L, L1, X).
                    /* Левое поддерево не пусто, значит,
                       оно содержит минимальное значение
                       всего дерева, которое нужно
                       удалить */
```

Основной предикат, выполняющий удаление вершины из дерева, будет выглядеть следующим образом:

```
tree_delete(X,tr(X,empty,R), R):-!.
                    /* X совпадает с корневым значением
                       исходного дерева, левое поддерево
                       пусто */
tree_delete (X,tr(X,L,empty), L):-!.
                    /* X совпадает с корневым значением
                       исходного дерева, правое поддерево
                       пусто */
tree_delete (X,tr(X,L,R), tr(Y,L,R1)):-
                    tree_del_min(R,R1, Y).
                         /* X совпадает с корневым
                            значением исходного дерева,
                            причем ни левое, ни правое
                            поддеревья не пусты */
tree_delete (X,tr(K,L,R), tr(K,L1,R)):-
                    X<K,!,
```

```
                    tree_delete (X,L,L1).
                         /* X меньше корневого значения
                            дерева */
tree_delete (X,tr(K,L,R), tr(K,L,R1)):-
                    tree_delete (X,R,R1).
                         /* X больше корневого значения
                            дерева */
```

Пример. Создадим предикат, который будет преобразовывать произвольный список в двоичный справочник. Предикат будет иметь два аргумента. Первый (входной) — произвольный список, второй (выходной) — двоичный справочник, построенный из элементов первого аргумента.

Будем переводить список в дерево рекурсивно. То, что из элементов пустого списка можно построить лишь пустое дерево, даст нам базис рекурсии. Шаг рекурсии будет основан на той идее, что для того, чтобы перевести непустой список в дерево, нужно перевести в дерево его хвост, после чего вставить голову в полученное дерево.

То же самое на Прологе:

```
list_tree([],empty). /* Пустому списку соответствует
                        пустое дерево */
list_tree([H|T],Tr):-
                    list_tree(T,Tr1),
                         /* Tr1 – дерево, построенное
                            из элементов хвоста
                            исходного списка */
                    tree_insert(H,Tr1,Tr).
                         /* Tr – дерево, полученное
                            в результате вставки
                            головы списка в дерево Tr1 */
```

Пример. Создадим обратный предикат, который будет «сворачивать» двоичный справочник в список с сохранением порядка элементов. Предикат будет иметь два аргумента. Первый (входной) — произвольный двоичный справочник, второй (выходной) — список, построенный из элементов первого аргумента:

```
tree_list(empty,[]). /* Пустому дереву соответствует
                        пустой список */
tree_list(tr(K,L,R),S):-
                    tree_list(L,T_L),
                           /* T_L – список,
```

```
                построенный из элементов
                левого поддерева */
        tree_list(R,T_R),
                /* T_L – список,
                построенный из элементов
                правого поддерева */
        conc(T_L,[K|T_R],S).
                /* S – список, полученный
                соединением списков T_L
                и [K|T_R] */
```

Заметьте, что, используя предикаты `list_tree` и `tree_list`, можно отсортировать список, состоящий из различных элементов, переписав его в двоичный справочник, а затем переписав двоичный справочник обратно в список.

Запишем предикат, выполняющий сортировку списка, переписывая его в двоичный список и обратно:

```
sort_listT(L,L_S):-
        list_tree(L,T),
                /* T- двоичный справочник,
                построенный из элементов
                исходного списка L */
        tree_list(T,L_S).
                /* L_S – список, построенный из
                элементов двоичного справочника
                T */
```

Так как в двоичном справочнике все элементы различны, при переписывании списка в двоичный справочник и обратно повторяющиеся элементы будут из него удалены. Неизменным останется количество элементов списка, только если все его элементы были различны.

Лекция 11. Строки

Обработка строк.

Ключевые слова: строка, длина строки, конкатенация строк.

В этой лекции мы займемся изучением строк. Такая структура данных, как строки, имеется практически в каждом языке программирования. Попробуем разобраться с тем, как можно обрабатывать строки в Прологе. На всякий случай напомню, что под строкой в Прологе понимается последовательность символов, заключенная в двойные кавычки.

Знакомство со спецификой обработки строк мы начнем с изучения некоторых встроенных предикатов Турбо Пролога, предназначенных для работы со строками, которыми нам предстоит в дальнейшем воспользоваться.

Начнем со встроенного предиката *str_len*, который предназначен для определения длины строки, т.е. количества символов, входящих в строку. Он имеет два аргумента: первый — строка, второй — количество символов. Имеется три варианта использования данного предиката.

Первый, наиболее естественный вариант использования этого предиката, когда первый аргумент связан, а второй свободен. В этом случае во второй аргумент будет помещено количество символов в первом аргументе.

Второй, также ожидаемый для стандартного «прологовского» предиката вариант использования этого предиката, когда оба аргумента связаны. В этом случае предикат будет успешен, если длина первого аргумента будет совпадать со вторым аргументом, и неуспешен – в противном случае.

И, наконец, третий, не столь распространенный вариант использования, в случае, когда второй аргумент связан, а первый — свободен. В этой ситуации первый аргумент будет означен строкой, состоящей из пробелов, причем количество пробелов будет равно второму аргументу.

Следующий стандартный предикат *concat* предназначен, вообще говоря, для соединения двух строк, или, как еще говорят, для их *конкатенации*. У него три аргумента, каждый строкового типа, по крайней мере, два из трех аргументов должны быть связаны. Итого получаем четыре возможных шаблона или четыре варианта использования этого предиката.

Первый вариант, когда связаны первые два аргумента. В этом случае третий аргумент будет означен строкой, полученной приписыванием второго аргумента к первому.

Второй вариант, когда связанными оказались первый и третий аргументы. В этой ситуации второй аргумент будет означен строкой, при приписывании которой к первому аргументу получится третий аргумент (если, конечно, такая строка существует). Если такой строки не может быть, предикат будет неуспешен.

Третий вариант аналогичен второму, за исключением того, что связанными аргументами оказываются второй и третий, а свободным — первый. В этом случае, естественно, первый аргумент будет означен строкой, при приписывании к которой можно получить третий аргумент, если это вообще возможно. Иначе предикат терпит неудачу.

И, наконец, четвертый вариант возникает, когда все три аргумента означены. Предикат будет успешен, если при соединении первого аргумента со вторым получится третий аргумент, и неуспешен — в противном случае.

Следующие три встроенных предиката предназначены для «разделки» строки на составляющие.

Предикат *frontchar* служит для разделения исходной строки на первый символ и «хвост», состоящий из символов строки оставшихся после удаления первого символа. Это чем-то напоминает представление списка в виде головы и хвоста. Причем первый и третий аргументы данного предиката принадлежат строковому домену, а второй — символьному. У этого предиката пять вариантов использования.

Первый вариант, когда первый аргумент связан, а второй и третий — свободны. В этом случае второй аргумент будет означен первым символом строки, а в третий аргумент будут записаны все символы исходной строки, начиная со второго.

Второй, также часто используемый вариант этого предиката, когда наоборот, первый аргумент свободен, а второй и третий — связаны. В этом случае в первый аргумент попадет строка, образованная приписыванием строки, находящейся в третьем аргументе, к символу, которым означен второй аргумент. Это некий аналог конкатенации, но соединяются не две строки, а символ и строка.

Третий вариант получается, когда первый и второй аргументы означены, а третий нет. В этой ситуации в третий аргумент будут переписаны все символы первого аргумента, начиная со второго, в случае, если первый символ первого аргумента совпадает со вторым аргументом. В противном случае предикат терпит неудачу.

Четвертый вариант похож на третий, но свободен второй аргумент, а связаны — первый и третий. В этом случае второй аргумент означивается первым символом первого аргумента, если оставшиеся символы первого аргумента образуют в точности третий аргумент. Иначе предикат будет неуспешен.

И, наконец, пятый вариант использования этого предиката возникает, когда все три его аргументы означены. В этом случае он будет истинен, если строка, хранящаяся в первом аргументе, совпадает со строкой, полученной приписыванием к символу из второго аргумента строки из третьего аргумента. В противном случае предикат будет ложен.

Предикат *frontstr* обобщает предикат *frontchar* в том смысле, что он тоже позволяет «откусить» от данной строки некоторое количество символов, но не обязательно один. Предикат имеет четыре параметра. В первом параметре указывается количество символов, которые копируются из второго параметра в третий; остатком второго параметра означивается четвертый аргумент. Как ни странно, этот предикат может использоваться только единственным описанным выше способом, а именно, первые два параметра у него входные, а третий и четвертый — выходные.

Если количество символов, указанных в первом параметре предиката *frontstr*, превышает длину строки из второго параметра, предикат терпит неудачу.

И, наконец, предикат *fronttoken* также дробит исходную строку, указанную в качестве первого параметра предиката, на две части. Во второй аргумент предиката попадает первый атом, входящий в строку, размещенную в первом аргументе, в третий — остаток входной строки, полученный после удаления из нее атома. Напомним, что атом — это или идентификатор, или беззнаковое число (целое или вещественное), или символ. У этого предиката существует пять вариантов использования.

Первый вариант, когда первый аргумент связан, а второй и третий — свободны. В этом случае второй аргумент будет означен первым атомом строки, находящейся в первом аргументе, а в третий аргумент будет записан остаток исходной строки.

Второй вариант использования этого предиката — когда, наоборот, первый аргумент свободен, а второй и третий — связаны. В таком случае этот предикат работает точно так же, как и конкатенация строк. В первый аргумент попадет строка, образованная приписыванием строки, находящейся в третьем аргументе, к строке, которой означен второй аргумент.

Третий вариант получается, когда первый и второй аргументы означены, а третий нет. В этой ситуации в третий аргумент будут переписаны все символы первого аргумента, остающиеся после удаления первого атома, в случае, если первый атом первого аргумента совпадает со вторым аргументом. В противном случае предикат терпит неудачу.

Четвертый вариант подобен третьему, но свободен второй аргумент, а связаны первый и третий. В этом случае второй аргумент означивается первым атомом первого аргумента, если оставшиеся символы первого аргумента совпадают со строкой, находящейся в третьем аргументе. Иначе предикат будет неудачен.

И, наконец, пятый вариант использования этого предиката возникает, когда все три его аргументы означены. В этом случае он будет истинным, если строка, хранящаяся в первом аргументе, совпадает со строкой, полученной приписыванием к атому из второго аргумента строки из третьего аргумента. В противном случае предикат будет ложным.

Для полноты картины следует еще упомянуть о предикате *isname*, который истинен, если его строковый аргумент является идентификатором, и ложен в противном случае.

Пример. Теперь попробуем применить рассмотренные предикаты. Создадим предикат, который будет преобразовывать строку в список символов. Предикат будет иметь два аргумента. Первым аргументом будет данная строка, вторым — список, состоящий из символов исходной строки.

Решение, как всегда, будет рекурсивным. Базис: пустой строке будет соответствовать пустой список. Шаг: с помощью встроенного предиката *frontchar* разобьем строку на первый символ и остаток строки, остаток строки перепишем в список, после чего добавим первый символ исходной строки в этот список в качестве первого элемента.

Запишем эту идею:

```
str_list("",[]). /* пустой строке соответствует пустой
                    список */
str_list(S,[H|T]):-
                frontchar(S,H,S1),
                   /* H — первый символ строки S,
                   S1 — остаток строки */
                str_list(S1,T).
                   /* T — список, состоящий из символов,
                      входящих в строку S1*/
```

Пример. Теперь попробуем немного модифицировать описанный предикат так, чтобы он преобразовывал строку не в список символов, а в список атомов.

Отличаться предыдущее решение будет заменой предиката `frontchar` на предикат `fronttoken`. Да еще надо не забыть в разделе описания предикатов заменить домен второго параметра со списка символов на список строк.

Запишем эту идею:

```
str_a_list("",[]). /* пустой строке по-прежнему
                      соответствует пустой список */
str_a_list(S,[H|T]):-
                   fronttoken(S,H,S1),
                /* H — первый атом строки S,
                   S1 — остаток строки */
                   str_a_list(S1,T).
                /* T — список, состоящий из атомов,
                   входящих в строку S1*/
```

Кстати, этот же предикат можно задействовать для создания списка слов, входящих в строку. Если слова разделены только пробелами, то он подойдет безо всяких изменений. Если же в строку могут входить еще и знаки препинания, то каждый из них попадет в итоговый список отдельным элементом. Для того чтобы получить список, элементами которого являются только слова, можно просто удалить из него все знаки препинания, что не сложно. К сожалению, работать эта идея будет только на английских словах. Русский текст этот предикат разобьет на отдельные символы, а не на слова.

Пример. Разработаем предикат, который будет преобразовывать список символов в строку. Предикат будет иметь два аргумента. Первым аргументом будет список символов, вторым — строка, образованная из элементов списка.

Базис рекурсии: пустому списку соответствует пустая строка. Шаг: если исходный список не пуст, то нужно перевести в строку его хвост, после чего, используя стандартный предикат *frontchar*, приписывать к первому элементу списка строку, полученную из хвоста исходного списка.

Запишем эту идею:

```
list_str([],"").  /* пустой строке соответствует пустой
                     список */
list_str([H|T],S):-
                 list_str(T,S1),
                         /* S1 – строка, образованная
                            элементами списка T */
                 frontchar(S,H,S1).
                         /* S – строка, полученная
                            дописыванием строки S1
                            к первому элементу списка H */
```

Пример. Создадим предикат, который по строке и символу подсчитает количество вхождений этого символа в данную строку. Предикат будет иметь три аргумента: первые два — входные (строка и символ), третий — выходной (количество вхождений второго аргумента в первый).

Решение, как обычно, будет рекурсивным. Рекурсия по строке, в которой ведется подсчет количества вхождений данного символа. Если строка пустая, то не важно, вхождения какого символа мы считаем – все равно ответом будет ноль. Это базис. Шагов рекурсии будет два в зависимости от того, будет ли первым символом строки символ, вхождения которого мы считаем, или нет. В первом случае нужно подсчитать, сколько раз искомый символ встречается в остатке строки, и увеличить полученное число на единицу. Во втором случае (когда первый символ строки от-

личен от символа, который мы считаем) увеличивать полученное число не нужно. При расщеплении строки на первый символ и хвост нужно воспользоваться уже знакомым нам предикатом *frontchar*:

```
char_count("",_,0). /* Любой символ не встречается
                       в пустой строке ни разу*/
char_count(S,C,N):-
                    frontchar(S,C,S1),!,
                      /* символ C оказался первым символом
                         строки S, в S1 — оставшиеся
                         символы строки S */
                    char_count(S1,C,N1),
                            /* N1 — количество вхождений
                               символа C в строку S1 */
                    N=N1+1.
                      /* N — количество вхождений
                         символа C в строку S получается
                         из количества вхождений символа C
                         в строку S1 добавлением единицы */
char_count(S,C,N):-
                    frontchar(S,_,S1),
                         /* первым символом строки S
                            оказался символ, отличный
                            от исходного символа C, в S1 —
                            оставшиеся символы строки S */
                    char_count(S1,C,N).
                         /* в этом случае количество
                            вхождений символа C в строку S
                            совпадает с количеством
                            вхождений символа C
                            в строку S1 */
```

Пример. Попробуем разработать предикат, который по символу и строке будет возвращать первую позицию вхождения символа в строку, если символ входит в строку, и ноль, если не входит. У предиката будет три параметра. Первые два — входные — символ и строка, третий — выходной — первая позиция вхождения первого параметра во второй параметр или ноль.

Не самая легкая задачка, но мы с ней справимся. Можно, конечно, записать в качестве базиса, что в пустой строке не встречаются никакие символы, но мы пойдем другим путем.

Вначале с помощью предиката `frontchar` разделим исходную строку на первый символ и остаток строки. Если первым символом строки

окажется тот самый символ, позицию которого мы ищем, значит, больше ничего делать не нужно. Ответом будет единица. В этом случае нам даже неважно, какие символы оказались в хвосте строки, поскольку мы ищем первое вхождение данного символа в строку.

В противном случае, если первым символом исходной строки является какой-то символ, отличный от искомого, нам нужно искать позицию вхождения символа в остаток строки. Если искомый символ найдется в хвосте, позиция вхождения символа в исходную строку будет на единицу больше, чем позиция вхождения этого символа в остаток строки. Во всех остальных ситуациях наш символ не встречается в исходной строке и, следовательно, мы должны означить третий аргумент нулем.

Давайте попробуем записать эти рассуждени:

```
str_pos(C,S,1):-
            frontchar(S,C,_),!.
                    /* Искомый символ C оказался первым
                       символом данной строки S */
str_pos(C,S,N) :-
            frontchar(S,_,S1),
                    /* S1 — состоит из всех символов
                       строки S, кроме первого, который
                       отличается от искомого символа C */
            str_pos(C,S1,N1),
                    /* N1 — это позиция, в которой
                       символ C встречается первый раз
                       в хвосте S1 или ноль*/
            N1<>0,!, /* если позиция вхождения символа C
                        в строку S1 не равна нулю,
                        то есть если он встречается
                        в строке S1, /
            N=N1+1. /* то, увеличив позицию его
                       вхождения на единицу, мы получим
                       позицию его вхождения в исходную
                       строку */
str_pos(_,_,0). /* искомый символ не входит в данную
                   строку */
```

Пример. Создадим предикат, который будет заменять в строке все вхождения одного символа на другой символ. У предиката будет четыре параметра. Первые три — входные (исходная строка; символ, вхождения которого нужно заменять; символ, которым нужно заменять первый сим-

вол); четвертым — выходным — параметром должен быть результат замены в первом параметре всех вхождений второго параметра на третий параметр.

Решение, как обычно, будет рекурсивным. Если строка пустая, значит, в ней нет символов, и, следовательно, заменять нечего. Результатом будет тоже пустая строка. Если же строка непустая, то мы должны разделить ее с помощью предиката `frontchar` на первый символ и строку, состоящую из остальных символов исходной строки.

Возможны два варианта. Либо первый символ исходной строки совпадает с тем, который нужно заменять, либо не совпадает.

В первом случае заменим все вхождения первого символа вторым символом в хвосте исходной строки, после чего, опять-таки с помощью предиката `frontchar`, приклеим полученную строку ко второму символу. В итоге в результирующей строке все вхождения первого символа будут заменены вторым символом.

Во втором случае, когда первый символ исходной строки не равен заменяемому символу, заменим в хвосте данной строки все вхождения первого символа на второй, после чего присоединим полученную строку к первому символу первоначальной строки:

```
str_replace("",_,_,""):-!. /* из пустой строки можно
                              получить только пустую
                              строку */
str_replace(S,C,C1,SO):-
                    frontchar(S,C,S1),!,
                         /* заменяемый символ C оказался
                            первым символом строки S,
                            S1 — остаток строки S */
                    str_replace(S1,C,C1,S2),
                         /* S2 — результат замены
                            в строке S1 всех вхождений
                            символа C на символ C1 */
                    frontchar(SO,C1,S2).
                         /* SO — результат склейки
                            символа C1 и строки S2 */
str_replace(S,C,C1,SO):-
                    frontchar(S,C2,S1),
                         /* разделяем исходную строку S
                            на первый символ C2
                            и строку S2, образованную
                            всеми символами строки S,
                            кроме первого */
```

```
                    str_replace(S1,C,C1,S2),
                           /* S2 – результат замены
                              в строке S1 всех вхождений
                              символа C на символ C1 */
                    frontchar(SO,C1,S2).
                           /* SO – результат соединения
                              символа C1 и строки S2 */
```

Если нам понадобится предикат, который будет заменять не все вхождения первого символа на второй, а только первое вхождение первого символа, то нужно просто из первого правила удалить вызов предиката `str_replace(S1,C,C1,S2)`.

Пример. Разработаем предикат, который будет удалять часть строки. Предикат будет иметь четыре параметра. Первые три входные: первый — исходная строка, второй — позиция, начиная с которой нужно удалять символы, третий — количество удаляемых символов. Четвертым — выходным — параметром будет результат удаления из строки, указанной в первом параметре, символов, в количестве, указанном в третьем параметре, начиная с позиции, указанной во втором параметре.

Запишем решение этой задачи. Начнем с того, что при помощи стандартного предиката `frontstr` разобьем исходную строку на две подстроки. Во вторую попадут все символы, начиная с той позиции, с которой нужно удалять символы. В первую — начало исходной строки. Вторую подстроку еще раз разделим на две подстроки. В первую подстроку поместим те символы, которые нужно удалить. В этом месте можно будет воспользоваться анонимной переменной. Во вторую подстроку попадут оставшиеся символы остатка исходной строки. Чтобы получить ответ, нам остается только соединить первую подстроку исходной строки с последней подстрокой второй подстроки. Мы получим строку, состоящую в точности из тех символов, которые и должны были остаться в итоговой строке.

Давайте запишем эти немного путаные размышления в виде предложения на Прологе.

```
str_delete(S,I,C,SO) :-
                I1=I-1, /* I1 – количество символов,
                           которые должны остаться
                           в начале строки S */
                frontstr(I1,S,S1,S2),
                        /* S1 – первые I1 символов
                           строки S, S2 – символы
```

```
                строки S, с I –го
                до последнего */
        frontstr(C,S2,_,S3),
                /* S3 – последние символы
                строки S2 ( или, что тоже
                самое, последние символы
                строки S */
        concat(S1,S3,SO).
                /* SO – строка, полученная
                соединением строк S1 и S3 */
```

Пример. Не помешает иметь в нашем хозяйстве предикат, который будет копировать часть строки. Предикат будет иметь четыре параметра. Первые три входные: первый — исходная строка, второй — позиция, начиная с которой нужно копировать символы, третий — количество копируемых символов. Четвертым — выходным — параметром будет результат копирования символов из строки, указанной в первом параметре, в количестве, указанном в третьем параметре, начиная с позиции, указанной во втором параметре.

Для решения этой задачи опять воспользуемся предикатом `frontstr`. Сначала получим хвост нашей строки, начиная с той позиции, с которой нужно копировать символы. Если после этого взять столько первых символов новой строки, сколько их нужно копировать, получим в точности ту подстроку исходной строки, которую требуется получить.

Зафиксируем наши рассуждения:

```
str_copy(S,I,C,SO) :-
                I1=I-1, /* I1 – это количество
                           символов, расположенных
                           в начале строки S, которые
                           не нужно копировать */
                frontstr(I1,S,_,S1),
                        /* S1 – строка, состоящая
                           из всех символов строки S,
                           с I-го и до последнего */
                frontstr(C,S1,SO,_).
                        /* SO – первые C символов
                           строки S1 */
```

Пример. Мы реализовали почти все операции, которые есть в большинстве стандартных алгоритмических языков типа Паскаля. Недостает, наверное, только предиката, который позволит нам вставить одну строку

внутрь другой строки. Предикат будет иметь четыре параметра. Первые три входные: первый — вставляемая строка; второй — строка, в которую нужно вставить первый аргумент; третий — позиция, начиная с которой нужно вставить первый параметр во второй. Четвертым — выходным — параметром будет результат вставки строки, указанной в первом параметре, в строку, указанную во втором параметре, начиная с позиции, указанной в третьем параметре.

Для реализации этого предиката разделим, используя предикат `frontstr`, исходную строку на две подстроки. Во вторую поместим все символы, начиная с позиции, в которую должна быть вставлена вторая строка, в первую — оставшееся начало исходной строки. После этого припишем, используя конкатенацию, к полученной строке ту строку, которую нужно было вставить. Для получения окончательного результата нам остается только дописать вторую подстроку исходной строки.

Запишем:

```
str_insert(S,S1,I,SO) :-
                    I1=I-1, /* I1 – это количество
                               символов, расположенных
                               в начале строки S, после
                               которых нужно вставить новые
                               символы */
                    frontstr(I1,S1,S1_1,S1_2),
                            /* S1_1 – первые I1 символов
                               строки S1, S1_2 –остаток
                               строки S1, с I–го и до
                               последнего */
                    concat(S1_1,S,S2),
                            /* S2 – строка, полученная
                               объединением строк S1_1
                               и S */
                    concat(S2,S1_2,SO).
                            /* SO – строка, полученная
                               слиянием строк S2 и S1_2 */
```

Пример. Создадим, для разнообразия, предикат, который будет подсчитывать количество имеющихся в строке цифр. Предикат будет иметь всего два аргумента. Входным аргументом будет строка, количество цифр в которой нужно подсчитать, выходным аргументом будет количество цифр в первом аргументе.

Для того чтобы реализовать этот предикат, нам придется разработать вспомогательный предикат, который будет означивать второй аргумент

единицей (если его первый аргумент является цифрой) и нулем (в противном случае). Основной предикат будет использовать рекурсию по длине строки. Базисом будет очевидный факт, говорящий, что в пустой строке цифр нет. Непустую строку с помощью предиката `frontchar` разделим на первый символ и хвост. Подсчитаем количество цифр в хвосте, после чего к полученному числу добавим единицу, если первый символ — цифра, и ноль, если первый символ — не цифра.

Запишем оба предиката, вспомогательный и основной:

```
dig(C,1):-
        '0'<=C,C<='9',!. /* C – цифра*/
dig(_,0).

count_digit("",0):-!. /* В пустой строке цифр нет */
count_digit(S,N):-
                  frontchar(S,C,S2),
                         /* C – первый символ строки S,
                            S2 – хвост строки S */
                  dig(C,M), /* M равен единице, если C –
                               цифра, и нулю – иначе */
                  count_digit(S2,N2),
                            /* N2 – количество цифр
                               в строке S2*/
                  N=N2+M. /* Количество  цифр во всей
                             строке больше на единицу,
                             чем количество цифр
                             в хвосте, если первый символ
                             строки – цифра, и равно
                             количеству цифр в хвосте –
                             иначе */
```

Лекция 12. Файлы

Описание файлового домена. Стандартные предикаты Турбо-Пролога для работы с файлами. Запись информации в файл. Чтение информации из файла. Переписывание информации из файла в файл.

Ключевые слова: файл, внутренне имя файла, внешнее имя файла.

Данная лекция посвящена работе с файлами. Обычно *файлом* называют именованную (то есть имеющую имя) совокупность данных, записанных на диске. Файл состоит из компонентов (элементов). При чтении или записи файловая переменная перемещается к очередному компоненту и делает его доступным для обработки. Попробуем разобраться с тем, как можно работать с файлами в Прологе.

Для начала вспомним, что пользовательские файлы описываются в разделе описания доменов следующим образом:

```
file = <символическое имя файла1>;...;
<символическое имя файлаN>
```

Обратите внимание, что при описании файловых доменов тип домена *file* располагается слева от равенства, а *символические имена* файлов — справа. Их еще называют *внутренними* или *логическими именами* файлов, в отличие от *внешних* или *физических имен* файлов. Символическое имя файла должно начинаться со строчной буквы.

Кроме пользовательских файлов, имеются стандартные файлы (или устройства), которые не нужно описывать в разделе описания доменов. Это:

- *stdin* (стандартное устройство ввода);
- *stdout* (стандартное устройство вывода);
- *stderror* (стандартное устройство вывода сообщений об ошибках);
- *keyboard* (клавиатура);
- *screen* (монитор);
- *printer* (параллельный порт принтера);
- *coml* (последовательный порт).

По умолчанию стандартным устройством ввода является клавиатура, а стандартным устройством вывода — монитор. Чтобы начать работу с пользовательским файлом, его нужно открыть, а по завершении работы — закрыть. Стандартные устройства ввода/вывода, а также параллельный и последовательный порты открывать и закрывать не нужно.

Далее мы познакомимся со встроенными предикатами Турбо Пролога, с помощью которых можно осуществлять операции открытия и закрытия файлов, а также многие другие операции с файлами.

Начнем наше знакомство со встроенных предикатов, предназначенных для открытия файлов. Каждый из следующих четырех предикатов имеет два входных параметра. Первый параметр — это внутреннее символическое имя, указанное в разделе описания доменов, второй параметр — это строка, представляющая внешнее имя файла.

Предикат `openread` открывает файл только для чтения. Если файл с указанным внешним именем не будет обнаружен, предикат терпит неудачу и выводит соответствующее сообщение об ошибке.

Предикат `openwrite` открывает файл только для записи. Этот предикат создает на диске новый файл. Если файл с указанным внешним именем уже существует, он будет стерт. Если по какой-то причине файл не может быть создан, предикат терпит неудачу и выводит соответствующее сообщение об ошибке.

Предикат `openappend` открывает файл только для дозаписи в конец файла. Если файл с указанным именем не будет обнаружен, предикат выводит соответствующее сообщение об ошибке.

Предикат `openmodify` открывает файл для чтения и записи одновременно. Если файл с указанным именем не будет обнаружен, предикат выводит соответствующее сообщение об ошибке.

Для того чтобы проверить, существует ли файл с указанным именем в указанном месте, используется предикат `existfile`. Этот предикат имеет один аргумент. Предикат истинен, если файл с именем, указанным в качестве его единственного параметра, существует, и ложен — в противном случае.

Обратите внимание на то, что эти предикаты связывают символическое имя файла с физическим именем открываемого файла. Поэтому, в отличие от других языков программирования, например Паскаля, нам нет необходимости перед операцией открытия файла проводить операцию связывания внутреннего и внешнего имен файла.

Поскольку символ «\», обычно используемый для разделения имен каталогов, применяется в Турбо Прологе для записи кодов символов, требуется использовать вместо одного обратного слэша два («\\»). Например, чтобы указать путь *«C:\Prolog\BIN»*, нужно записать строку *«C:\\Prolog\\BIN»*.

Пример. Напишем замену для стандартного предиката `openread`. Предикат, который будет открывать файл на чтение (в случае, если он существует) и выводить сообщение о том, что файл с таким именем не найден (иначе). Этот предикат, как и предикат `openread`, будет иметь два аргумента. Первым аргументом будет внутреннее символическое имя файла, вторым — строка, представляющая внешнее дисковое имя файла. Наша модификация предиката должна быть корректной и завершаться успехом в любом случае, вне зависимости от того, наличествует открываемый файл или отсутствует.

При реализации этого предиката воспользуемся встроенными предикатами: existfile для проверки существования; предикатом openread для открытия существующего файла на чтение; предикатом write для вывода сообщения:

```
openFile(F,N):-
            existfile(N),!, /* проверяем существование
                               файла с именем N */
            openread(F,N). /* связываем внешний файл
                              с именем N с файловой
                              переменной F и открываем
                              его на чтение */
openFile(_,N):-
            write("Файл с именем ",N," не найден!").
                        /* выдаем сообщение, если предикат
                           existfile потерпел неудачу */
```

Аналогичным образом можно модифицировать предикаты openappend и openmodify. Предикат openwrite можно модифицировать таким образом, чтобы при попытке открыть существующий файл на запись предикат вначале выдавал бы предупреждение о том, что содержимое этого файла будет уничтожено.

Для того чтобы корректно закрыть открытый файл, используется предикат closefile. В качестве его единственного параметра указывается символическое имя файла. Предикат в любом случае успешен, даже если соответствующий файл не был открыт.

С закрытым файлом можно работать только целиком. Он может быть переименован или удален с помощью предикатов renamefile и deletefile.

Предикат deletefile удаляет файл, указанный в качестве его единственного параметра. Если по какой-то причине удалить файл не получается, этот предикат выдает сообщение об ошибке.

Предикат renamefile изменяет имя файла, указанного в качестве его первого параметра, на имя, указанное в качестве его второго параметра. Если не существует файла, чье имя указано в первом параметре, или существует файл, чье имя указано во втором параметре, предикат выдаст сообщение об ошибке.

Предикат disk позволяет задать или узнать текущий диск и/или каталог, в зависимости от того, связан его единственный аргумент или свободен.

Кроме того, имеется предикат dir, который позволяет выбрать из списка файлы, соответствующие шаблону, указанному в качестве второго параметра этого предиката, и находящиеся в каталоге, указанном в пер-

вом параметре. Выбранное нажатием клавиши *Enter* имя попадает в переменную, чье имя указано в качестве третьего параметра этого предиката.

Есть еще вариант этого предиката, имеющий три дополнительных входных параметра: четвертый — включает/отключает отображение подкаталогов; пятый — разрешает/запрещает изменять нажатием клавиши *F4* шаблон, в соответствии с которым отображаются файлы; шестой — разрешает/запрещает отображение пути в заголовке окна. Ноль в четвертом, пятом и шестом параметрах означает запрет соответствующей опции, ненулевое значение — разрешение.

Предикат `eof` (сокращение от `End Of File` — «конец файла») успешен, если достигнут конец файла, в противном случае он – неуспешен. В качестве его единственного входного параметра указывается символическое имя файла. Он обычно используется при организации рекурсивного считывания всех компонентов файла. Если его попытаться применить к файлу, открытому на запись, будет выдано сообщение об ошибке.

Содержимое файла можно рассматривать как поток компонентов. Каждый компонент файла находится на какой-то позиции. Для того чтобы узнать текущую позицию чтения или записи в файле, либо для того, чтобы изменить эту позицию, служит предикат `filepos`.

У него три аргумента. Первый аргумент — это символическое имя файла, второй — позиция внутри первого аргумента, которую нужно узнать или установить, третий — номер режима, который задает, откуда отсчитывается позиция.

Номер режима может принимать одно из трех значений: ноль, единица или двойка.

Если он равен нулю, то позиция считается от начала файла. Если он равен единице, позиция отсчитывается от текущей позиции. Этот режим имеет смысл только при означенном втором аргументе, когда предикат используется для изменения текущей позиции. Потому что если второй аргумент предиката не был означен, а третий аргумент равен единице, то во второй аргумент будет помещен ноль. Смещение текущей позиции чтения или записи в файле относительно текущей позиции, естественно, равно нулю. Если же третий аргумент равен двойке, то позиция отсчитывается от конца файла.

Предикат может быть использован двояко. Если все три его аргумента связаны, то позиция, из которой осуществляется чтение или в которую производится запись, будет изменена в соответствии с числом, которым означен второй аргумент. Если его второй аргумент свободен, а первый и третий связаны, то второй аргумент будет означен текущей позицией чтения или записи.

Опишем ниже два предиката, служащие для перенаправления потоков ввода-вывода.

Для переопределения текущего устройства ввода или для того, чтобы узнать, какое устройство ввода является текущим, служит предикат `readdevice`. Этот предикат имеет один параметр, в качестве которого указывается символическое имя файла. Если предикат `readdevice` используется с несвязанным параметром, то он возвращает в него имя устройства, которое в настоящий момент является активным устройством чтения информации. Если его параметр означен именем открытого для чтения устройства, то этот предикат переопределит активное устройство ввода информации.

Предикат `writedevice` подобен предикату `readdevice`, однако используется для переопределения текущего устройства вывода или для получения имени текущего устройства вывода. Он также имеет один параметр, который может быть использован либо как входной, либо как выходной. В первом случае будет переопределено текущее устройство записи информации (должно быть открыто на запись или дозапись), во втором — параметр будет связан с именем активного устройства вывода.

Для записи данных в текущее устройство записи служит уже знакомый нам предикат `write`, который до этого мы использовали для вывода информации на монитор. Для чтения информации из активного устройства вывода — также уже знакомые нам предикаты `readln` (читает строку), `readint` (читает целое), `readreal` (читает вещественное число), `readchar` (читает символ), `readterm` (читает терм).

Имеется также предикат `file_str`, который целиком читает символы файла в строку или, наоборот, записывает содержимое строки в файл, в зависимости от того, свободен ли второй параметр этого предиката. Первым входным параметром этого предиката является символическое имя файла, а вторым — строка, в которую считывается содержимое файла или из которой записывается информация в него.

Предикат `flush` используется для принудительной записи в файл содержимого внутреннего буфера, выделенного для файла, указанного в его единственном параметре. Обычно он используется при работе с принтером.

И, наконец, последний встроенный предикат, о котором будет упомянуто в этой лекции, это предикат `filemode`. Он позволяет узнать или задать режим доступа к файлу. У этого предиката два параметра. Первый параметр — символическое имя файла, второй параметр задает или принимает режим работы с файлом, имя которого указано в первом параметре. Если второй параметр свободен, то он будет означен текущим режимом работы с файлом, а если связан, то указанный режим будет установлен. Второй параметр может принимать одно из двух значений. Значение ноль соответствует двоичному (бинарному) режиму работы с файлом, а значение единица — текстовому режиму. В текстовом режиме к строкам добавляются символы возврата каретки и перевода строки. Чаще исполь-

зуется текстовый режим, поскольку при работе в двоичном режиме данные можно записывать только посимвольно.

Рассмотрим на примерах, как можно использовать описанные предикаты.

Пример. Начнем с предиката, выводящего содержимое произвольного файла на экран. Предикат будет иметь один параметр строкового типа, представляющий собой внешнее дисковое имя файла.

При реализации этого предиката воспользуемся встроенными предикатами: `existfile` для проверки наличия файла на диске; `openread` для открытия существующего файла на чтение; `readdevice` для перенаправления ввода; `eof` для проверки, не исчерпали ли мы содержимое файла; `readchar` для чтения символов из файла; `write` для вывода прочитанного символа на экран.

Сначала напишем вспомогательный предикат, который будет читать символы из файла и выводить их на экран до тех пор, пока не будет достигнут конец файла.

Основной предикат будет проверять существование файла, указанного в качестве его параметра. Если файла с таким именем не существует, будет выдано соответствующее сообщение. Если файл с указанным именем существует, откроем его на чтение предикатом `openread` и определим его в качестве текущего устройства ввода информации, используя предикат `readdevice`. Далее воспользуемся нашим вспомогательным предикатом для вывода содержимого файла на экран. Закроем файл предикатом `closefile`. Сделаем текущим устройством чтения клавиатуру, воспользовавшись предикатом `readdevice`. В принципе, можно на этом остановиться, а можно организовать паузу до нажатия любой клавиши предикатом `readchar(_)`. До этого неплохо бы вывести сообщение о том, что работа будет продолжена после нажатия любой клавиши.

Так как это — наш первый опыт работы с файлами в Прологе, приведем всю программу целиком, со всеми разделами. В разделе описания внутренней цели организуем ввод имени файла и вызов основного предиката:

```
DOMAINS /* раздел описания доменов */
file = f /* f – внутреннее имя файла */
PREDICATES /* раздел описания предикатов */
write_file(file)
writeFile(string)
CLAUSES /* раздел описания предложений */
write_file(f):-
          not(eof(f)),!, /* если файл f еще
                            не закончился */
          readchar(C), /* читаем из него символ */
```

```
            write(C," "), /* выводим символ на экран*/
            write_file(f). /* продолжаем процесс */
    write_file(_). /* это предложение нужно, чтобы предикат
                      не потерпел неудачу в случае, когда
                      будет достигнут конец файла */

    writeFile(F_N):-
              existfile(F_N),!, /* убеждаемся в существовании
                                   файла с именем F_N */
              openread(f,F_N), /* связываем внешний файл F_N
                                  с внутренним файлом f
                                  и открываем его на чтение */
              readdevice(f), /* устанавливаем в качестве
                                устройства чтения файл f */
              write_file(f), /* вызываем предикат, выводящий
                                на экран все символы
                                файла f */
              closefile(f), /* закрываем файл */
              readdevice(keyboard), /* перенаправляем ввод
                                       на клавиатуру */
              nl,nl, /* пропускаем строку */
              write("Нажмите любую клавишу"),
                           /* выводим сообщение на экран */
              readchar(_)./* ждем нажатия любой клавиши */

    writeFile(F_N):-
              write("Файл с именем ",F_N," не найден!").
                           /* выдаем сообщение, если предикат
                              existfile потерпел неудачу */

    GOAL /* раздел описания внутренней цели*/

    write("Введите имя файла: "),
    readln(F_N), /* читаем название файла в переменную F_N */
    writeFile(F_N).
```

Пример. Теперь создадим предикат, который будет формировать файл из символов, вводимых с клавиатуры. Предикат будет иметь один параметр, представляющий собой внутреннее имя файла.

Начнем реализацию предиката с того, что сделаем экран текущим устройством вывода информации, выведем пользователю предложение ввести символ, сообщим о том, какой символ будет завершать ввод. Про-

читаем символ с клавиатуры, выведем его на экран и сравним с тем символом, который завершает ввод. Если символы совпадают, закроем файл. Иначе: сделаем текущим устройством записи файл, запишем символ в файл, вызовем основной предикат.

Еще раз процитируем программу целиком, а в дальнейшем будем приводить только основной предикат. В разделе описания внутренней цели организуем ввод имени файла, открытие файла на запись и вызов основного предиката:

```
DOMAINS
file=f
PREDICATES
Readfile
CLAUSES
readfile:-
        writedevice(screen), /* назначаем текущим
                                устройством записи
                                экран */
        write("Введите символ (# – конец ввода)"), nl,
                             /* выводим сообщение */
        readchar(C), /* читаем символ с клавиатуры */
        write(C), /* дублируем символ на экран */
        C<>'#',!, /* если это не #*/
        writedevice(f), /* , то перенаправляем вывод
                           в файл */
        write(C), /* записываем символ в файл */
        readfile.
readfile:-
        closefile(f). /* если введенный символ оказался
                         равен символу '#', то закрываем
                         файл */
GOAL

write("Введите имя файла: "),
readln(F_N), /* читаем название файла в переменную F_N */
openwrite(f,F_N), /* связываем внешний файл F_N
                     с внутренним файлом f и открываем
                      его на запись */
readfile(f).
```

Попробуйте изменить эту программу так, чтобы в файл записывались с клавиатуры не символы, а целые строки. При этом не забудьте при

записи строки в файл добавить символы конца строки и перевода каретки. Это можно сделать с помощью встроенного предиката `nl` или сцепив записываемую строку с двумя символами с кодами `13` и `10`.

Пример. Запишем предикат, который будет выводить содержимое файла на экран и принтер. Предикат будет иметь один параметр, представляющий собой внутреннее имя файла.

Напишем только этот предикат, оставив «за кадром» проверку на существование внешнего файла и его открытие на чтение. Все это мы делали не раз. Наш предикат должен проверять, не достигнут ли конец файла. В случае достижения конца файла его нужно закрыть. Если мы еще не добрались до конца файла, читаем символ, используя предикат `readchar`. Выводим его на экран посредством предиката `write`, перенаправляем вывод на принтер предикатом `writedevice`, выводим символ на принтер, заставляем принтер немедленно напечатать символ, используя предикат `flush`, устанавливаем для последующего вывода символа текущим устройством записи информации экран. Повторяем этот процесс, пока не будут исчерпаны все символы исходного файла:

```
writeFile_to_scr_and_pr(f):-
                  not(eof(f)),!, /* если файл f еще
                                    не закончился */
                  readchar(C), /* читаем из файла f
                                  символ в переменную C */
                  write(C), /* выводим символ C на экран */
                  writedevice(printer),
                            /* устанавливаем текущим
                               устройством записи
                               принтер*/
                  write(C), /* выводим символ C на экран */
                  flush(printer),
                          /* сбрасываем данные из буфера
                             на принтер */
                  writedevice(screen),
                          /* перенаправляем вывод
                             на экран*/
                  writeFile_to_scr_and_pr(f).
                          /* продолжаем */
writeFile_to_scr_and_pr:-
        closefile(f). /* закрываем файл f */
```

Обратите внимание на использование стандартного предиката `flush`. Как правило, принтер выводит данные на печать только тогда, ко-

гда будет заполнен его внутренний буфер. Предикат `flush` позволяет осуществить немедленный вывод на печатающее устройство накопленной в буфере информации.

Попробуйте изменить этот предикат так, чтобы чтение из файла и, соответственно, вывод на экран и принтер осуществлялись не посимвольно, а построчно. Для этого замените предикат `readchar` на предикат `readln`, а также не забудьте добавить символы конца строки и перевода каретки при выводе.

Пример. Напишем предикат, который будет переписывать компоненты одного файла в другой файл так, чтобы в итоговом файле все английские буквы были большими. У предиката будет два аргумента строкового типа. Первым параметром будет внешнее имя исходного файла, вторым параметром — внешнее имя итогового файла.

Начнем с создания вспомогательного предиката, который будет читать из первого файла строки, переводить их символы в верхний регистр, используя встроенный предикат `upper_lower`, а полученные строки записывать во второй файл. Когда первый файл буден исчерпан, этот предикат должен закрыть оба файла.

Основной предикат будет проверять существование исходного файла. Если файла с указанным именем не существует, он должен вывести соответствующее сообщение. Если файл существует, он должен быть открыт на чтение. Итоговый файл должен быть открыт на запись, первый файл установлен в качестве текущего устройства чтения информации, второй — в качестве текущего устройства записи информации. После этого остается только вызвать вспомогательный предикат.

Запишем эти два предиката. Надеюсь, что недостающие разделы описаний читатели уже в состоянии дописать самостоятельно.

```
transform:-
          not(eof(f)),!, /* если файл f еще
                            не закончился */
          readln(S), /* читаем из файла f строку
                        в переменную S */
          upper_lower(S_U,S),
                     /* S_U — результат преобразования
                        всех символов строки S в верхний
                        регистр */
          write(S_U),nl, /* записываем строку S_U
                            в файл f_o */
          transform. /* продолжаем процесс */
transform:-
          closefile(f), /* закрываем файл f */
```

```
          closefile(f_o). /* закрываем файл f_o */
upper_file(N_F,N_o_F):-
          existfile(N_F),!, /* проверяем существование
                               файла с именем N_F */
          openread(f,N_F), /* связываем внешний файл
                              с именем N_F с внутренним
                              файлом f и открываем его
                              на чтение */
          readdevice(f), /* устанавливаем в качестве
                            текущего устройства чтения
                            файл f */
          openwrite(f_o,N_o_F),
                    /* связываем внешний файл с именем
                       N_o_F с внутренним файлом f
                       и открываем его на запись */
          writedevice(f_o), /* устанавливаем в качестве
                               текущего устройства записи
                               файл f_o */
          transform. /* вызываем вспомогательный
                        предикат */
upper_file(N_F,_):-
          write("Файл с именем ",N_F," не найден!").
                       /* выдаем сообщение, если предикат
                          existfile был неуспешен */
```

Лекция 13. Внутренние (динамические) базы данных

Работа с внутренними (динамическими) базами данных: добавление фактов в базу, удаление фактов из базы. Проекты «Телефонный справочник», «Словарь».

Ключевые слова: внутренняя база данных, внешняя база данных.

В этой лекции мы начнем изучать работу с базами данных в Прологе.

С одной стороны, Пролог-программы не зря называют базами знаний. На Прологе легко реализуются реляционные базы данных, наиболее распространенные в настоящее время. Любая таблица реляционной базы данных может быть описана соответствующим набором фактов, где каждой записи исходной таблицы будет соответствовать один факт. Каждому полю будет соответствовать аргумент предиката, реализующего таблицу. Многие дистрибутивы Пролога содержат в качестве примера реализацию базовой части языка SQL. Можно сказать, что структура реляционных баз данных включается в структуру Пролог-программ.

С другой стороны, Турбо Пролог, на который мы все-таки ориентируемся в нашем курсе, имеет встроенные средства для работы с двумя типами баз данных: *внутренними* и *внешними*. *Внутренние* базы данных так называются потому, что они обрабатываются исключительно в оперативной памяти компьютера, в отличие от *внешних* баз данных, которые могут обрабатываться на диске или в памяти. Так как внутренние базы данных размещаются в оперативной памяти компьютера, конечно, работать с ними можно существенно быстрее, чем с внешними. С другой стороны, емкость оперативной памяти, как правило, намного меньше, чем емкость внешней памяти. Отсюда следует, что объем внешней базы данных может быть существенно больше объема внутренней базы данных. И если предполагается, что база может оказаться довольно большой, то следует использовать именно внешние базы данных.

Изучение внешних баз данных выходит за рамки данного курса.

В этой лекции мы займемся изучением *внутренних* или, как их еще называют, *динамических* баз данных.

Внутренняя база данных состоит из фактов, которые можно динамически, в процессе выполнения программы, добавлять в базу данных и удалять из нее, сохранять в файле, загружать факты из файла в базу данных. Эти факты могут использовать только предикаты, описанные в разделе описания предикатов базы данных.

```
DATABASE [ – <имя базы данных>]
<имя предиката>(<имя домена первого аргумента>,...,
< имя домена n-го аргумента>)
...
```

Если раздел описания предикатов базы данных в программе только один, то он может не иметь имени. В этом случае он автоматически получает стандартное имя dbasedom. В случае наличия в программе нескольких разделов описания предикатов базы данных только один из них может быть безымянным. Все остальные должны иметь уникальное имя, которое указывается после названия раздела DATABASE и тире. Когда объявлен раздел описания предикатов базы данных, компилятор внутренне объявляет соответствующий домен с таким же именем, как у этого раздела; это позволяет специальным предикатам обрабатывать факты как термы.

Описание предикатов базы данных совпадает с их описанием в разделе описания предикатов PREDICATES. Однако эти предикаты можно задействовать в качестве параметров встроенных предикатов, с которыми мы познакомимся чуть позже. Кроме того, факты, использующие эти предикаты, могут добавляться и удаляться во время выполнения программы.

Обратите внимание на то, что в базе данных могут содержаться только факты, а не правила вывода, причем факты базы данных не могут содержать свободных переменных. Это еще одно существенное отличие Турбо Пролога от классического Пролога, в котором во время работы программы можно добавлять и удалять не только факты, но и правила. Заметим, что в Visual Prolog, который является наследником Турбо Пролога, в названии раздела описания предикатов внутренней базы данных слово DATABASE заменено синонимом FACTS, что еще больше подчеркивает, что во внутренней базе данных могут храниться только факты, а не правила.

Давайте познакомимся со встроенными предикатами Турбо Пролога, предназначенными для работы с внутренней базой данных. Все рассматриваемые далее предикаты могут использоваться в варианте с одним или двумя аргументами. Причем одноаргументный вариант используется, если внутренняя база данных не имеет имени. Если же база поименована, то нужно использовать двухаргументный предикат, в котором второй аргумент — это имя базы.

Начнем с предикатов, с помощью которых во время работы программы можно добавлять или удалять факты базы данных.

Для добавления фактов во внутреннюю базу данных может использоваться один из трех предикатов – assert, asserta или assertz. Разница между этими предикатами заключается в том, что предикат asserta добавляет факт перед другими фактами (в начало внутренней базы дан-

ных), а предикат assertz добавляет факт после других фактов (в конец базы данных). Предикат assert добавлен для совместимости с другими версиями Пролога и работает точно так же, как и assertz. В качестве первого параметра у этих предикатов указывается добавляемый факт, а в качестве второго, необязательного — имя внутренней базы данных, в которую добавляется факт. Можно сказать, что предикаты assert и assertz работают с совокупностью фактов, как с очередью, а предикат asserta — как со стеком.

Для удаления фактов из базы данных служат предикаты retract и retractall. Предикат retract удаляет из внутренней базы данных первый с начала факт, который может быть отождествлен с его первым параметром. Вторым необязательным параметром этого предиката является имя внутренней базы данных.

Для удаления всех предикатов, соответствующих его первому аргументу, служит предикат retractall. Для удаления всех фактов из некоторой внутренней базы данных следует вызвать этот предикат, указав ему в качестве первого параметра анонимную переменную. Так как анонимная переменная сопоставляется с любым объектом, а предикат retractall удаляет все факты, которые могут быть отождествлены с его первым аргументом, из внутренней базы данных все факты будут удалены. Если вторым аргументом этого предиката указано имя базы данных, то факты удаляются из указанной базы данных. Если второй аргумент не указан, факты удаляются из единственной неименованной базы данных. Заметим, что предикат retractall может быть заменен комбинацией предикатов retract и fail следующим образом:

```
retractall2(Fact):-
              retract(Fact),
              fail.
retractall2(_).
```

Для сохранения динамической базы на диске служит предикат save. Он сохраняет ее в текстовый файл с именем, которое было указано в качестве первого параметра предиката. Если второй необязательный параметр был опущен, происходит сохранение фактов из единственной неименованной внутренней базы данных. Если было указано имя внутренней базы данных, в файл будут сохранены факты именно этой базы данных.

Факты, сохраненные в текстовом файле на диске, могут быть загружены в оперативную память командой consult. Первым параметром этого предиката указывается имя текстового файла, из которого нужно загрузить факты. Если второй параметр опущен, факты будут загружены в единственную неименованную внутреннюю базу данных. Если второй

параметр указан, факты будут загружены в ту внутреннюю базу данных, чье имя было помещено во второй параметр предиката. Предикат будет неуспешен, если для считываемого файла недостаточно свободного места в оперативной памяти или если указанный файл не найден на диске, или если он содержит ошибки (ниже будет разъяснено чуть подробнее, какими они бывают).

Заметим, что сохраненная внутренняя база данных представляет собой обычный текстовый файл, который может быть просмотрен и/или изменен в любом текстовом редакторе. При редактировании или создании файла, который планируется применить для последующей загрузки фактов с использованием предиката `consult`, нужно учитывать, что каждый факт должен занимать отдельную строку. Количество аргументов и их тип должны соответствовать описанию предиката в разделе database. В файле не должно быть пустых строк, внутри фактов не должно быть пробелов, за исключением тех, которые содержатся внутри строк в двойных кавычках, других специальных символов типа конца строки, табуляции и т.д. Давайте на примере разберемся со всеми этими предикатами.

Пример. Напишем программу, реализующую компьютерный вариант телефонного справочника. Основное назначение этой не очень сложной программы — находить по фамилии человека его телефонный номер или, наоборот, по телефонному номеру — фамилию владельца телефона. У пользователя нашей программы должна быть возможность добавлять информацию в базу данных, а также удалять и изменять устаревшую информацию.

Приступим к реализации нашего проекта. Внутренняя база данных будет содержать факты, описывающие единственный предикат, имеющий два аргумента. Первым аргументом предиката будет фамилия человека, а вторым — его телефонный номер. Для упрощения программы будем считать, что соответствие между фамилиями и номерами телефонов – взаимооднозначное, то есть каждой фамилии соответствует не более одного телефонного номера, и наоборот.

Сделаем так, чтобы при запуске программы появлялось меню, из которого пользователь мог выбрать, какое действие с телефонной базой он хотел бы осуществить. Реализуем пять операций:

1. Получение информации о телефонном номере по фамилии человека.
2. Получение информации о фамилии абонента по телефонному номеру.
3. Добавление новой записи в телефонную базу.
4. Изменение существующей в телефонной базе записи.
5. Удаление записи из телефонной базы.

Нужно учесть, что пользователь может ошибиться и нажать клавишу, не соответствующую ни одной из пяти указанных операций. После вы-

полнения каждой из операций программа должна вернуться обратно в меню, чтобы у пользователя не было необходимости запускать программу заново, если ему нужно выполнить еще одно действие.

Кроме того, у пользователя должна быть возможность выйти из программы, не совершая никаких действий. При выходе из программы факты телефонной базы должны быть сохранены из оперативной памяти в файл на диске, а оперативная память очищена от ненужных фактов.

Эти действия выполняет следующее правило (символ '0' означает, что пользователь нажал соответствующую клавишу):

```
m('0'):-
        save("phones.ddb "), /* сохраняем телефонную базу
                                в файл */
        retractall(_)./* удаляем все факты из внутренней
                         базы данных */
```

В начале работы программы факты из телефонной базы, хранящейся в файле на диске, должны загружаться во внутреннюю базу данных, в случае, если такой файл существует.

Предикат, предназначенный для выполнения этих действий, выглядит следующим образом:

```
start:-
        existfile("phones.ddb"),!,
              /* если существует файл с телефонной базой */
        consult("phones.ddb "),
              /* , то загружаем факты во внутреннюю базу
                   данных */
        menu. /* и вызываем меню */
start:-
        menu. /* если такого файла еще нет, просто
                 вызываем меню */
```

Если пользователь выбрал первую операцию, должен быть выдан телефонный номер абонента (если в телефонной базе имеется соответствующий факт) или сообщение о том, что в телефонной базе нет такой информации.

Это реализуют два приведенных ниже предиката:

```
m('1'):-
        write("Введите фамилию"), nl,
                /* выводим приглашение ввести фамилию */
```

```
        readln(Name), /* читаем введенную фамилию
                         в переменную Name */
        name_phone(Name, Phone),
                     /* вызываем предикат, который
                        помещает в переменную Phone
                        телефонный номер, соответствующий
                        фамилии Name или сообщение
                        об отсутствии информации */
        write("Номер телефона: ",Phone),
                     /* выводим значение переменной
                        Phone */
        readchar(_), /* ждем нажатия любой клавиши */
        menu. /* возвращаемся в меню */
name_phone(Name,Phone):-
                   phone(Name,Phone),!.
name_phone(_,"Нет информации о телефонном номере").
                     /* если нужного факта во внутренней
                        базе данных не нашлось,
                        то вместо телефонного номера
                        возвращаем соответствующее
                        сообщение */
```

Если пользователь желает выполнить вторую операцию, то должна быть выведена фамилия абонента, если в нашей телефонной базе имеется соответствующий факт. Иначе выводится сообщение о том, что у нас нет такой информации.

Соответствующие предикаты будут выглядеть следующим образом:

```
m('2'):-
        write("Введите номер телефона"),nl,
        readln(Phone),
        phone_name(Name, Phone),
        write("Фамилия абонента: ",Name),
        readchar(_),
        menu. /* вызываем меню */
phone_name(Name,Phone):-
                phone(Name,Phone).
phone_name("Нет информации о владельце телефона",_).
                /* если нужного факта во внутренней базе
                   данных не нашлось, то вместо фамилии
                   абонента возвращаем соответствующее
                   сообщение */
```

Если пользователем была выбрана третья операция, то нужно дать ему возможность ввести фамилию и номер абонента, после чего добавить соответствующий факт в базу данных.

Это будет выглядеть следующим образом:

```
m('3'):-
        write("Введите фамилию"),nl,
        readln(Name),
        write("Введите номер телефона"),nl,
        readln(Phone),
        assert(phone(Name,Phone)),
                        /* добавляем факт во внутреннюю
                           базу данных */
        menu. /* вызываем меню */
```

Если пользователь желает выполнить четвертую операцию, то нужно дать ему возможность ввести фамилию абонента и его новый телефонный номер, после чего удалить устаревшую информацию из телефонной базы (с помощью предиката `retract`) и добавить туда новую информацию (используя встроенный предикат `assert`).

Соответствующее этим рассуждениям предложение:

```
m('4'):-
        clearwindow,
        write("Введите фамилию"),nl,
        readln(Name),
        write("Введите новый номер телефона"),nl,
        readln(Phone),
        retract(phone(Name,_)),
                    /* удаляем устаревшую информацию
                       из внутренней базы данных */
        assert(phone(Name,Phone)),
                    /* добавляем новую информацию
                       в телефонную базу */
        menu. /* вызываем меню */
```

Если пользователем была выбрана пятая операция, то нужно узнать у него, например, номер (или фамилию) абонента, после чего удалить соответствующую информацию из внутренней базы данных, воспользовавшись предикатом `retract`.

Запишем это предложение:

```
m('5'):-
        write("Укажите номер телефона, запись о котором
        нужно удалить из телефонной базы"), nl,
        readln(Phone),
        retract(phone(_,Phone)),
                    /* удаляем соответствующий факт
                       из внутренней базы данных */
        menu. /* вызываем меню */
```

Приведем полный текст программы:

```
DOMAINS /* раздел описания доменов */
name, number = String /* фамилию абонента и телефонный
                         номер будем хранить в виде
                         строк */
file=f /* файловый домен будем использовать для
          считывания с диска и записи на диск нашей
          телефонной базы */
DATABASE /* раздел описания предикатов внутренней
            базы данных */
phone(name, number)
PREDICATES /* раздел описания предикатов */
name_phone(name, number) /* этот предикат находит номер
                            телефона по фамилии абонента */
phone_name(name, number) /* этот предикат находит фамилию
                            абонента по номеру телефона */
m(char) /* этот предикат реализует выполнение
           соответствующего пункта меню */
menu /* этот предикат реализует вывод меню и обработку
        выбора пользователя */
start /* этот предикат проверяет наличие файла
         с телефонной базой на диске и либо загружает
         факты из нее во внутреннюю базу данных,
         если такой файл существует, либо создает
         этот файл, если его не было */
CLAUSES /* раздел описания предложений */
name_phone(Name,Phone):-
                    phone(Name,Phone),!.
name_phone(_,"Нет информации о телефонном номере").
                  /* если соответствующего факта
                     во внутренней базе данных не нашлось,
```

```
                    вместо телефонного номера возвращаем
                    соответствующее сообщение */
phone_name(Name,Phone):-
                      phone(Name,Phone).
phone_name("Нет информации о владельце телефона",_).
                  /* если соответствующего факта
                    во внутренней базе данных не нашлось,
                    вместо фамилии абонента возвращаем
                    соответствующее сообщение */
menu:-
     clearwindow, /* очистка текущего окна */
     write("1- Получение телефонного номера
     по фамилии "),nl,
     write("2 – Получение фамилии абонента по номеру
     телефона "),nl,
     write("3 – Добавление новой записи в телефонную
     базу."),nl,
     write("4 – Изменение номера абонента"),nl,
     write("5 – Удаление записи из телефонной базы"),nl,
     write("0 – Выйти"),nl,
     readchar(C), /* читаем символ с клавиатуры */
m(C). /* вызываем выполнение соответствующего пункта
         меню */
m('1'):-
     clearwindow,
     write("Введите фамлию"), nl,
     readln(Name),
     name_phone(Name, Phone),
     write("Номер телефона: ",Phone),
     readchar(_),
     menu.
m('2'):-
     clearwindow,
     write("Введите номер телефона"),nl,
     readln(Phone),
     phone_name(Name, Phone),
     write("Фамилия абонента: ",Name),
     readchar(_),
     menu.
m('3'):-
     clearwindow,
     write("Введите фамилию"),nl,
```

```
      readln(Name),
      write("Введите номер телефона"),nl,
      readln(Phone),
      assert(phone(Name,Phone)),
                        /* добавляем факт во внутреннюю
                           базу данных */
      menu.
m('4'):-
      clearwindow,
      write("Введите фамилию"),nl,
      readln(Name),
      write("Введите новый номер телефона"),nl,
      readln(Phone),
      retract(phone(Name,_)),
                        /* удаляем устаревшую информацию
                           из внутренней базы данных */
      assert(phone(Name,Phone)),
                        /* добавляем новую информацию
                           в телефонную базу */
      menu.
m('5'):-
      clearwindow,
      write("Укажите номер телефона, запись о котором
      нужно удалить из телефонной базы"), nl,
      readln(Phone),
      retract(phone(_,Phone)), /* удаляем соответствующий
                                  факт из внутренней базы
                                   данных */
      menu.
m('0'):-
save("phones.ddb "), /* сохраняем телефонную базу
                        в файл */
retractall(_)./* удаляем все факты из внутренней
                 базы данных */
m(_):-
      menu. /* если пользователь по ошибке нажал клавишу,
               отличную от тех, реакция на которые
               предусмотрена, ничего плохого
               не произойдет, будет отображено меню
               еще раз */
start:-
```

```
      existfile("phones.ddb"),!, /* если файл с телефонной
                                    базой существует */
      consult("phones.ddb "), /* загружаем факты во
                                 внутреннюю базу данных */
      menu. /* вызываем меню */
start:-
      openwrite(f,"phones.ddb"),
                        /*  если файла с телефонной
                            базой не существует, создаем
                            его */
      closefile(f),
      menu. /* вызываем меню */
GOAL /* раздел внутренней цели*/
Start
```

Пример. Другой распространенный вариант использования внутренних баз данных — это повышение эффективности программ за счет добавления уже вычисленных фактов в базу данных. При попытке вычислить предикат сначала проверяется, нет ли в базе данных уже вычисленного значения, и если оно там уже есть, то просто берется это значение. Если же ответа еще нет, он вычисляется обычным способом, после чего добавляется в базу данных для повторного использования. Эта техника еще называется *мемоизация* или *табулирование*.

Давайте разработаем табулированную версию предиката, вычисляющего число Фиббоначи по его номеру. В пятой лекции мы уже рассматривали предикат, вычисляющий числа Фиббоначи. Выглядел он следующим образом:

```
fib(0,1):-!. /* нулевое число Фиббоначи равно единице */
fib(1,1):-!. /* первое число Фиббоначи равно единице */
fib(N,F) :-
          N1=N-1, fib(N1,F1), /* F1 это N-1-е число
                                 Фиббоначи */
          N2=N-2, fib(N2,F2), /* F2 это N-2-е число
                                 Фиббоначи */
          F=F1+F2. /* N-е число Фиббоначи равно сумме
                      N-1-го числа Фиббоначи и N-2-го
                      числа Фиббоначи */
```

Чем плох этот вариант предиката, вычисляющего числа Фиббоначи? Получается, что при вычислении очередного числа происходит мно-

гократное перевычисление предыдущих чисел Фиббоначи, что не может не приводить к замедлению работы программы.

Изменим нашу программу следующим образом: добавим в нее раздел описания предикатов внутренней базы данных. В этот раздел добавим описание одного-единственного предиката, который будет иметь два аргумента. Первый аргумент — это номер числа Фиббоначи, а второй аргумент — само число.

Сам предикат, вычисляющий числа Фиббоначи, будет выглядеть следующим образом. Базис индукции для первых двух чисел Фиббоначи оставим без изменений. Для шага индукции добавим еще одно правило. Первым делом будем проверять внутреннюю базу данных на предмет наличия в ней уже вычисленного числа. Если оно там есть, то никаких дополнительных вычислений проводить не нужно. Если же числа в базе данных не окажется, вычислим его по обычной схеме как сумму двух предыдущих чисел, после чего добавим соответствующий факт в базу данных.

Попробуем придать этим рассуждениям некоторое материальное воплощение:

```
fib2(0,1):-!. /* нулевое число Фиббоначи равно единице */
fib2(1,1):-!. /* первое число Фиббоначи равно единице */
fib2(N,F):-
          fib_db(N,F),!. /* пытаемся найти N-е число
                            Фиббоначи среди уже
                            вычисленных чисел, хранящихся
                            во внутренней базе данных */
fib2(N,F) :-
          N1=N-1, fib2(N1,F1), /* F1 это N-1-е число
                                  Фиббоначи */
          N2=N-2, fib2(N2,F2), /* F2 это N-2-е число
                                  Фиббоначи */
          F=F1+F2, /* N-е число Фиббоначи равно сумме
                      N-1-го числа Фиббоначи и N-2-го
                      числа Фиббоначи */
          asserta(fib_db(N,F)).
                  /* добавляем вычисленное N-е число
                     Фиббоначи в нашу внутреннюю базу
                     данных*/
```

Заметьте, что при каждом вызове подцели `fib2(N2,F2)` используются значения, уже вычисленные при предыдущем вызове подцели `fib2(N1,F1)`.

Попробуйте запустить два варианта предиката вычисляющего числа Фиббоначи для достаточно больших номеров (от 30 и выше) и почувствуйте разницу во времени работы. Минуты — работы первого варианта и доли секунды — работы его табулированной модификации.

Справедливости ради стоит заметить, что существует другой вариант ускорения работы предиката, вычисляющего числа Фиббоначи, без использования баз данных.

Будем искать сразу два числа Фиббоначи. То, которое нам нужно найти, и следующее за ним. Соответственно, предикат будет иметь третий дополнительный аргумент, в который и будет помещено следующее число. Базис рекурсии из двух предложений сожмется в одно, утверждающее, что первые два числа Фиббоначи равны единице.

Вот как будет выглядеть этот предикат:

```
fib_fast(0,1,1):-!.
fib_fast(N,FN,FN1):-
                N1=N-1,fib_fast(N1,FN_1,FN),
                FN1=FN+FN_1.
```

Если следующее число Фиббоначи искать не нужно, можно сделать последним аргументом анонимную переменную или добавить описанный ниже двухаргументный предикат:

```
fib_fast(N,FN):-
             fib_fast(N,FN,_).
```

Лекция 14. Пролог и искусственный интеллект

Работа с внутренними (динамическими) базами данных: добавление фактов в базу, удаление фактов из базы. Проекты «Телефонный справочник», «Словарь»

Ключевые слова: искусственный интеллект, экспертные системы.

В этой лекции речь пойдет о возможных применения Пролога в области искусственного интеллекта. Конечно, ознакомиться с данной темой достаточно полно в рамках одной лекции мы не успеем. Однако хочется надеяться, что мы сможем хотя бы пробежаться по верхушкам и рассмотреть пару простых примеров.

В 1950 году Алан Тьюринг в статье «Вычислительная техника и интеллект» (книга «Может ли машина мыслить?») предложил эксперимент, позднее названный «тест Тьюринга», для проверки способности компьютера к «человеческому» мышлению. В упрощенном виде смысл этого теста заключается в том, что можно считать искусственный интеллект созданным, если человек, общающийся с двумя собеседниками, один из которых человек, а второй — компьютер, не сможет понять, кто есть кто. То есть в соответствии с тестом Тьюринга, компьютеру требуется научиться имитировать человека в диалоге, чтобы его можно было считать «интеллектуальным».

Такой подход к распознаванию искусственного интеллекта многие критиковали, однако никаких достойных альтернатив тесту Тьюринга предложено не было.

Первый пример, который мы рассмотрим, будет относиться к области обработки естественного языка.

Пример. Создадим программу, имитирующую разговор психотерапевта с пациентом. Прообразом нашей программы является «Элиза», созданная Джозефом Вейценбаумом в лаборатории искусственного интеллекта массачусетского технологического института в 1966 году (названная в честь Элизы из «Пигмалиона»). Она была написана на языке Лисп и состояла всего из нескольких десятков строк программного кода. Эта программа моделировала методику известного психотерапевта Карла Роджерса. В этом подходе психотерапевт играет роль «вербального зеркала» пациента. Он переспрашивает пациента, повторяет его слова, позволяя ему самому найти выход из сложившейся ситуации, прийти в состояние душевного равновесия.

На самом деле эта программа пытается сопоставить вводимые пользователем ответы с имеющимися у нее шаблонами и, если ей это удается, шаблонно же отвечает.

Вейценбаум создал эту программу в качестве шутки. Многие пациенты, пообщавшись с детищем Вейценбаума, утверждали, что «Элиза» им помогла, и отказывались верить, что их собеседником был не психотерапевт, а компьютерная программа. Всю оставшуюся жизнь Вейценбаум пытался охладить восторженных поклонников его программы и убедить общественность, что машина не может мыслить.

Наша программа будет действовать по следующему алгоритму:

1. Попросит человека описать имеющуюся у него проблему.
2. Прочитает строку с клавиатуры.
3. Попытается подобрать шаблон, которому соответствует введенная человеком строка.
4. Если удалось — выдаст соответствующий этому шаблону ответ пользователю.
5. Если подобрать шаблон не удалось — попросит продолжать рассказ.
6. Возвращаемся к пункту 2 и продолжаем процесс.

Для решения этой задачи нам понадобится предикат, преобразующий строку, вводимую пользователем в список слов. Можно было бы воспользоваться модифицированной версией предиката `str_a_list`, рассмотренного нами в одиннадцатой лекции. Однако он использует предикат `fronttoken`, который в Турбо Прологе, в отличие от Visual Prolog, из русских предложений выделяет не слова, а отдельные символы. Поэтому мы напишем новый вспомогательный предикат, который будет считывать символ за символом до тех пор, пока не встретит символ-разделитель (пробел, запятая, точка и другой знак препинания). Так как проверять совпадение очередного символа с символом-разделителем нам придется не раз, заведем список символов-разделителей. Поместим его в раздел описания констант и назовем separators (символы-разделители). После этого все символы до символа-разделителя будут помещены в первое слово строки, а все символы, идущие после символа-разделителя, обработаны подобным образом.

Кроме того, при переписывании строки в список ее слов мы переведем все русские символы, записанные в верхнем регистре (большие буквы), в нижний регистр (маленькие буквы). Это облегчит в дальнейшем процесс распознавания слов. Нам не придется предусматривать всевозможные варианты написания пользователем слова (например, «Да», «да», «ДА»), мы будем уверены, что все символы слова — строчные («да»).

При реализации этого предиката нам понадобится три вспомогательных предиката.

Первый предикат будет преобразовывать прописные русские буквы в строчные, а все остальные символы оставлять неизменными. У него будет два аргумента: первый (входной) — исходный символ, второй (выходной) — символ, полученный преобразованием первого аргумента.

При написании данного предиката стоит учесть, что строчные русские буквы расположены в таблице символов двумя группами. Первая группа (буквы от ‘а’ до ‘п’) имеют, соответственно, коды от 160 до 175. Вторая группа (буквы от ‘р’ до ‘я’) — коды от 224 до 239.

С учетом вышеизложенного предикат можно записать, например, так:

```
lower_rus(C,C1):-
            'А'<=C,C<='П',!, /*  символ С лежит между
                                 буквами 'А' и 'П' */
            char_int(C,I), /* I – код символа С */
            I1=I+(160-128), /* 160 – код буквы 'а',
                               128 – код буквы 'А'*/
            char_int(C1,I1).
                          /* C1 – символ с кодом I1 */
lower_rus(C,C1):-
            'Р'<=C,C<='Я',!, /*  символ С лежит между
                                 буквами 'Р' и 'Я' */
            char_int(C,I), /* I – код символа С */
            I1=I+(224-144), /* 224 – код буквы 'р',
                               144 – код буквы 'Р'*/
            char_int(C1,I1).
                           /* C1 – символ с кодом I1 */
lower_rus(C,C). /* символ С отличен от прописной русской
                   буквы и, значит, мы не должны его
                   изменять */
```

Второй предикат `first_word` будет иметь три аргумента. Первый (входной) — исходная строка, второй и третий (выходные) — соответственно, первое слово строки (не содержащее прописных русских букв) и остаток строки, полученный удалением из него первого слова.

Выглядеть его реализация будет следующим образом:

```
first_word("","","");-!. /* из пустой строки можно
                            выделить только пустые
                            подстроки */
first_word(S,W,R):- /* W – первое слово строки S, R –
                       остальные символы исходной
                       строки S */
                frontchar(S,C,R1),
                   /* C – первый символ строки S, R1 –
                      остальные символы */
```

```
                not(member(C,separators)),!,
                        /* символ C не является
                           символом-разделителем */
                first_word(R1,S1,R),
                        /* S1 — первое слово строки R1,
                           R — оставшиеся символы
                           строки R1 */
                lower_rus(C,C1),
                        /* если C — прописная русская
                           буква , то C1 — соответствующая
                           ей строчная буква, иначе
                           символ C1 не отличается
                           от символа C */
                frontchar(W,C1,S1).
                        /* W — результат «приклеивания»
                           символа C1 в начало строки S1 */
first_word(S,"",R):- /* в случае, если первый символ
                        оказался символом-разделителем, */
                frontchar(S,_,R). /* его нужно выбросить, */
```

Третий предикат `del_sep` будет предназначен для удаления из начала строки символов-разделителей. У него будет два аргумента. Первый (входной) — исходная строка, второй (выходной) — строка, полученная из первого аргумента удалением символов-разделителей, расположенных в начале строки, если таковые имеются:

```
del_sep("","")):-!.
del_sep(S,S1):-
            frontchar(S,C,R),
                        /* C — первый символ строки,
                           R — остальные символы */
            member(C,separators),!,
                        /* если C является
                           символом-разделителем, */
            del_sep(R,S1).
                        /* то переходим к рассмотрению
                           остатка строки */
del_sep(S,S) . /* если первый символ строки не является
                  символом-разделителем, то удалять
                  нечего */
```

И наконец, предикат, преобразующий строку в список слов:

```
str_w_list("",[]):-!. /* пустой строке соответствует
                         пустой список слов, входящих
                         в нее */
str_w_list(S,[H T]):-
                    first_word(S,H,R),!,
                         /* H — первое слово строки S,
                            R — оставшиеся символы
                            строки S */
                    str_w_list(R,T).
                         /* T — список, состоящий из слов,
                            входящих в строку R */
```

Основную работу в программе будет осуществлять предикат `recognize`, задачей которого будет распознавать шаблон, которому можно сопоставить введенную строку. Этот предикат на входе будет получать список слов строки, а на выходе будет выдавать номер шаблона. По этому номеру другой предикат должен будет выдать на экран соответствующую реакцию (вопрос, реплику, уточнение).

Наша учебная программа будет распознавать одиннадцать шаблонов:

0. Человек хочет закончить работу с программой. Об этой ситуации свидетельствует наличие в списке таких слов, как «пока», «свидания» (часть словосочетания «до свидания»). В ответ программа также прощается и выражает надежду, что она смогла чем-нибудь помочь.
1. Человек испытывает какое-то чувство (наличие в списке слова *«испытываю»*). Программа реагирует вопросом о том, как давно человек испытывает это чувство.
2. Если во вводимой строке встретились слова *«любовь»* или *«чувства»*, то программа поинтересуется, не боится ли человек эмоций.
3. При обнаружении слова *«секс»* во входном списке слов будет выдано сообщение о важности сообщения.
4. В случае наличия слов *«бешенство»*, *«гнев»* или *«ярость»*, программа уточнит, что человек испытывает в данный момент времени.
5. В ответ на краткий ответ (*«да»* или *«нет»*) будет выдана просьба рассказать подробнее.
6. Если в списке слов найдутся слова *«комплекс»* или *«фиксация»*, программа отреагирует замечанием о том, что человек слишком много «играет».
7. Появление слова *«всегда»* в строке, введенной человеком, приводит к ответной реакции — вопросу о том, может ли человек привести какой-нибудь пример.

8. В случае, если человек упомянул кого-то из своих родных (*«папа»*, *«мама»*, *«жена»*, *«муж»*, *«брат»*, *«сестра»*, *«сын»*, *«дочь»* и т.д.), программа попросит рассказать поподробнее о его семье. При этом упомянутый родственник будет помещен в базу данных, чтобы потом продолжить этот разговор.
9. Если в процессе разговора была сделана запись во внутреннюю базу данных и в данный момент спросить больше не о чем, программа «вспомнит» об упомянутом родственнике и выдаст фразу: «ранее Вы упоминали...»
10. И наконец, если введенная строка не подходит ни под один шаблон, программа просит продолжить рассказ.

А теперь запишем всю программу целиком:

```
CONSTANTS /* раздел описания констант */
separators=[' ', ',', '.', ';']
                  /* символы-разделители (пробел,
                     запятая, точка, точка с запятой
                     и т.д.) */
DOMAINS /* раздел описания доменов */
i=integer
s=string
ls=s* /* список слов */
lc=char* /* список символов */
DATABASE /* раздел описания предикатов базы данных */
Important(s)
PREDICATES /* раздел описания предикатов */
member(s,ls) /* проверяет принадлежность строки списку
                строк */
member(char,lc) /* проверяет принадлежность символа списку
                   символов */
lower_rus(char,char) /* преобразует прописную русскую
                        букву в строчную букву */
del_sep(s,s) /* удаляет из начала строки
                символы-разделители */
first_word(s,s,s) /* делит строку на первое слово
                     и остаток строки */
str_w_list(s,ls) /* преобразует строку в список слов */
read_words(ls)  /* читает строку с клавиатуры, возвращает
                   список слов, входящих в строку*/
recognize(ls,i) /* сопоставляет списку слов число,
                   кодирующее шаблон */
answ(ls) /* выводит ответ человеку */
```

```
eliz /* основной предикат */
repeat
CLAUSES /* раздел описания предложений */
eliz:-
     repeat,
     read_words(L), /*  читаем строку с клавиатуры,
                        преобразуем ее в список слов L */
     recognize(L,I), /* сопоставляем списку слов L
                        номер шаблона I */
     answ(I),nl, /* выводим ответ, соответствующий номеру
                    шаблона I */
     I=0 /* номер шаблона I, равный нулю, означает,
            что человек попрощался */.
read_words(L):-
              readln(S), /* читаем строку */
              str_w_list(S,L). /* преобразуем строку
                                  в список слов */
recognize(L,0):-
              member("пока",L),!;
              member("свидания",L),!.
recognize(L,1):-
              member("испытываю",L),!.
recognize(L,2):-
              member("любовь",L),!;
              member("чувства",L),!.
recognize(L,3):-
              member("секс",L),!.
recognize(L,4):-
              member("бешенство",L),!;
              member("гнев",L),!;
              member("ярость",L),!.
recognize(L,5):-
              L=["да"],!;
              L=["нет"],!.
recognize(L,6):-
              member("комплекс",L),!;
              member("фиксация",L),!.
recognize(L,7):-
              member("всегда",L),!.
recognize(L,8):-
              member("мать",L),assert(important "своей
              матери")),!;
```

```
                member("мама",L),assert(important("своей
                маме")),!;
                member("отец",L),assert(important("своем
                отце")),!;
                member("папа",L),assert(important("своем
                папе")),!;
                member("муж",L),assert(important("своем
                муже")),!;
                member("жена",L),assert(important("своей
                жене")),!;
                member("брат",L),assert(important("своем
                брате")),!;
                member("сестра",L),assert(important("своей
                сестре")),!;
                member("дочь",L),assert(important("своей
                дочери")),!;
                member("сын",L),assert(important("своем
                сыне")),!.
recognize(_,9):-
                important(_),!.
recognize(_,10).
answ(0):-
        write("До свидания"),nl,
        write("Надеюсь наше общение помогло Вам").
answ(1):-
        write("Как давно Вы это испытываете?").
answ(2):-
        write("Вас пугают эмоции?").
answ(3):-
        write("Это представляется важным").
answ(4):-
        write("А что Вы испытываете сейчас?").
answ(5):-
        write("Расскажите об этом подробнее").
answ(6):-
        write("Слишком много игр").
answ(7):-
        write("Вы можете привести какой-нибудь пример?").
answ(8):-
        write("Расскажите мне подробнее о своей семье").
answ(9):-
        important(X),!,
```

```
        write("Ранее Вы упомянули о ",X),
        retract(X).
answ(10):-
        write("Продолжайте, пожалуйста").
repeat.
repeat:-
        repeat.
member(X,[X|_]):-!.
member(X,[_|S]):-member(X,S).
lower_rus(C,C1):-
             'А'<=C,C<='П',!, /* символ С лежит между
                                 буквами 'А' и 'П' */
             char_int(C,I), /* I — код символа С */
             I1=I+(160-128), /* 160 — код буквы 'а',
                                128 — код буквы 'А'*/
             char_int(C1,I1). /* C1 — символ с кодом I1 */
lower_rus(C,C1):-
             'Р'<=C,C<='Я',!, /* символ С лежит между
                                 буквами 'Р' и 'Я' */
             char_int(C,I), /* I — код символа С */
             I1=I+(224-144), /* 224 — код буквы 'р',
                                144 — код буквы 'Р'*/
             char_int(C1,I1). /* C1 — символ с кодом I1 */
lower_rus(C,C). /* символ С отличен от прописной русской
                   буквы и, значит, мы не должны его
                   изменять */
del_sep("","");-!.
del_sep(S,S1):-
             frontchar(S,C,R),
                         /* С — первый символ строки,
                            R — остальные символы */
             member(C,separators),!,
                         /* если С является
                            символом-разделителем, */
             del_sep(R,S1). /* то переходим
                               к рассмотрению остатка
                               строки */
del_sep(S,S) . /* если первый символ строки не является
                  символом-разделителем, то удалять
                  нечего */
```

```
str_w_list("",[]):-!.
          /* пустой строке соответствует пустой список
             слов, входящих в нее */
str_w_list(S,[H|T]):-
                  first_word(S,H,R),!,
                       /* H — первое слово строки S, R —
                          оставшиеся символы строки S */
                  str_w_list(R,T).
                       /* T — список, состоящий из слов,
                          входящих в строку R */
first_word("","",""):-!. /* из пустой строки можно
                            выделить только пустые
                            подстроки */
first_word(S,W,R):- /* W — первое слово строки S, R —
                       остальные символы исходной строки S */
                  frontchar(S,C,R1),
                          /* C — первый символ строки S,
                             R1 — остальные символы */
                  not(member(C,separators)),!,
                          /* символ C не является
                             символом-разделителем */
                  first_word(R1,S1,R),
                          /* S1 — первое слово строки R1,
                             R — оставшиеся символы
                             строки R1 */
                  lower_rus(C,C1),
                       /* если C — прописная русская
                          буква, то C1 — соответствующая
                          ей строчная буква, иначе символ
                          C1 не отличается от символа C */
                  frontchar(W,C1,S1).
                          /* W — результат «приклеивания»
                             символа C1 в начало
                             строки S1 */
first_word(S,"",R):- /* в случае, если первый символ
                        оказался символом-разделителем, */
                  frontchar(S,_,R). /* его нужно
                                       выбросить, */
GOAL /* раздел описания цели */
write("Расскажите, в чем заключается Ваша проблема?"),nl,
eliz,
readchar(_).
```

Усовершенствовать работу этой программы можно двумя способами. С одной стороны, можно увеличить количество шаблонов, с другой стороны, можно организовать разные реакции на некоторые из шаблонов (например, используя случайные числа).

В 1977 г. Кеннет Колби, основываясь на принципах организации «Элизы», создал программу, которая подобным образом вводила в заблуждение уже не клиентов психиатров, а самих докторов. Большинство из них после общения с его программой решили, что имели дело с реальным параноиком.

В 1996 г. Грег Гарви создал программную модель католического исповедника, которая опиралась на те же идеи, что и «Элиза».

Другие варианты «Элизы» можно найти в следующих книгах:

- Л. Стерлинг, Э. Шапиро. Искусство программирования на языке Пролог. — М.:Мир, 1990.
- Д. Марселлус. Программирование экспертных систем на Турбо-Прологе. — М.: Финансы и статистика, 1994.

Теперь рассмотрим еще один пример применения Пролога в области искусственного интеллекта. Создадим небольшую экспертную систему. **Экспертными системами** обычно называют программы, которые могут заменить эксперта в какой-то предметной области. Мы построим классификационную экспертную систему, которая будет пытаться угадать загаданное человеком животное. Если загаданное человеком животное окажется неизвестно нашей программе, у нее будет возможность пополнить свою базу знаний новой информацией.

В связи с тем, что мы планируем пополнять нашу базу знаний, мы будем по окончании работы сохранять ее в файл, а при начале работы считывать информацию из файла в оперативную память. Для этого мы воспользуемся изученными в предыдущей лекции внутренними (динамическими) базами данных.

Так как в Турбо Прологе в базе данных можно размещать только факты, представим правила, определяющие животных в виде фактов.

Определим два предиката внутренней базы данных, которые позволят нам хранить информацию о животных.

Один из них предназначен для хранения характеристик животных и будет иметь два аргумента: первый — номер свойства, второй — его словесное описание.

Небольшой базовый набор свойств может выглядеть, например, так:

```
cond(1,"кормит детенышей молоком").
cond(2,"имеет перья").
cond(3,"плавает").
cond(4,"ест мясо").
```

```
cond(5,"имеет копыта").
cond(6,"летает").
cond(7,"откладывает яйца").
cond(8,"имеет шерсть").
cond(9,"имеет полосы").
cond(10,"имеет пятна").
cond(11,"имеет черно-белую окраску").
cond(12,"имеет длинную шею").
cond(13,"имеет длинные ноги").
cond(14,"имеет щупальца").
```

Второй предикат будет хранить описание животных. Первый его аргумент — название животного, второй — список, элементами которого являются номера свойств, присущих данному животному.

Выглядеть эта база знаний может примерно следующим образом:

```
rule("гепард",[1,4,8,10]).
rule("тигр",[1,4,8,9]).
rule("жираф",[1,5,8,10,12,13]).
rule("зебра",[1,5,8,9,11]).
rule("страус",[2,14]).
rule("пингвин",[2,3,11]).
rule("орел",[2,6]).
rule("кит",[1,3,11]).
rule("осьминог",[3,14]).
```

По сути дела, в виде фактов записаны правила. Например, правило: «если животное кормит детенышей молоком, имеет копыта, пятна, длинную шею и ноги, то это жираф», записано в виде `rule("жираф", [1,5,11,13,14])`.

Во второй базе мы будем хранить ответы человека в следующем виде:

```
cond_is(N,'1') /*  если загаданное животное имеет свойство
                   с номером N */
cond_is(N,'2') /*  если загаданное животное не имеет
                   свойства с номером N */
```

Первую базу назовем `knowledge`, а вторую — `dialog`.

Процесс отгадывания задуманного животного будет проходить следующим образом. Будем последовательно перебирать животных, имеющихся в нашей базе знаний. Если загаданное животное обладает всеми характеристиками известного программе животного, делается вывод о том,

кто был загадан. Если не удается сопоставить загаданному животному ни одно из животных, имеющихся в базе знаний, производится пополнение базы.

Вот как будет выглядеть реализация написанного выше:

```
animals:-
        rule(X,L),
        check(L),
        nl,write("Я думаю это ",X),
        nl,write("Я прав? (1 — да, 2 — нет)"),
        read_true_char(C),C='1',!.
animals:-
        nl,write("Я не знаю, что это за животное"),nl,
        nl,write("Давайте добавим его в мою базу
                знаний."),nl,
        update.
```

Предикат `check` осуществляет проверку свойств, номера которых входят в список, указанный в качестве его единственного аргумента:

```
check([H|T]):-
           test_cond(H),
           check(T).
check([]).
```

Предикат `test_cond` проверяет наличие у загаданного животного свойства с номером, указанным в качестве его единственного аргумента. Если человеком ранее уже был дан ответ (положительный или отрицательный) по поводу наличия данного свойства, информация об этом имеется в базе данных. Если же в базе нет никакой информации о наличии данного свойства у загаданного животного, нужно задать человеку соответствующий вопрос и добавить его ответ в базу.

Вот как это можно записать:

```
test_cond(H):-
             cond_is(H,'1'),!. /* в базе имеется
                                  информация о наличии
                                  данного свойства */
test_cond(H):-
             cond_is(H,'2'),!,
             fail. /* в базе имеется информация
                      об отсутствии данного свойства */
```

```
test_cond(H):- /* в базе нет никакой информации о данном
                  свойстве, получаем ее у человека */
               cond(H,S),
               nl,write("Оно ",S,"? (1 — да, 2 — нет)»),
               read_true_char(A),
               assert(cond_is(H,A)),
               test_cond(H).
```

Предикат `read_true_char` осуществляет проверку нажатой пользователем клавиши, и если она отлична от '1' или '2', выводит соответствующее сообщение и повторно считывает символ с клавиатуры:

```
read_true_char(C):-
                readchar(C1),
                test(C1,C).
test(C,C):-
                '1'<=C,C<='2',!.
test(_,C):-
                write("Нажмите 1 или 2!"),nl,
                readchar(C1),
                test(C1,C).
```

Предикат `update` осуществляет добавление новой информации в базу знаний. Он читает название нового животного, с помощью предиката `add_cond` формирует список номеров свойств загаданного животного, добавляет соответствующий факт в базу знаний, сохраняет базу в файл.

Вот как он будет выглядеть:

```
update:-
        nl,write("Введите название животного:"),
        readln(S),
        add_cond(L), /* указываем свойства животного */
        assert(rule(S,L),knowledge),
                     /* добавляем информацию в базу
                        знаний*/
        save("animals.ddb",knowledge)
                     /* сохраняем базу знаний в файл */.
```

Предикат `add_cond` с помощью предиката `print_cond` выводит уже имеющуюся в базе информацию о свойствах загаданного животного и спрашивает, известно ли еще что-нибудь о нем. В случае необходимости добавляет его новые характеристики, используя предикат `read_cond`.

```
add_cond(L):-
             cond_is(_,'1'),!,
                       /* имеется информация о свойствах
                          животного */
             nl,write("О нем известно, что оно: "),
             print_cond(1,[],L1),
                       /* вывод имеющейся информации
                          о животном */
             nl,write("Известно ли о нем еще что-нибудь?
             (1 — да, 2 — нет)"),
             read_true_char(C),
             read_cond(C,L1,L).
add_cond(L):-
             read_cond('1',[],L).
```

Предикат read_cond, используя предикат ex_cond, добавляет в список номера свойств, уже имеющихся в базе; используя предикат new_cond, добавляет в список номера новых свойств, а также описания самих свойств — в базу знаний:

```
read_cond('1',L,L2):-
                  ex_cond(1,L,L1,N),
                  new_cond(N,L1,L2),!.
read_cond(_,L,L):-!.
```

Основные предикаты мы рассмотрели, а вот как будет выглядеть вся программа целиком:

```
DOMAINS
i=integer
s=string
c=char
li=i* /* список целых чисел */
DATABASE — knowledge
cond(i,s) /* свойства животных */
rule(s,li) /* описания животных */
DATABASE — dialog
cond_is(i,c) /* номер условия; '1' — имеет место,
                '2' — не имеет места у загаданного
                животного */
PREDICATES
start
```

```
animals /* отгадывает животное */
check(li) /* добавляет в базу информацию о новом
             животном */
test_cond(i) /* проверяет, обладает ли загаданное
                животное свойством с данным номером */
update /* добавляет в базу информацию о новом животном */
add_cond(li) /* возвращает список, состоящий из номеров
                свойств, имеющихся у нового животного */
print_cond(i,li,li) /* добавляет в список номера свойств,
                       относительно которых уже были даны
                       утвердительные ответы */
read_cond(c,li,li) /* добавляет в список номера свойств,
                      о которых еще не спрашивалось */
ex_cond(i,li,li,i) /* добавляет в список номера имеющихся
                      в базе свойств, которыми обладает
                      новое животное */
wr_cond(c,i,li,li)
new_cond(i,li,li) /* добавляет в список номера новых
                     свойств, которыми обладает новое
                     животное, а также добавляет описания
                     новых свойств в базу знаний */
read_true_char(c) /* с помощью предиката test читает
                     символ с клавиатуры, пока он
                     не окажется равен '1' или '2'*/
test(c,c) /* добивается, чтобы пользователь нажал один
             из символов, '1' или '2' */
CLAUSES
start:-
     consult("animals.ddb",knowledge),
                      /*  загружаем в базу информацию
                          из базы знаний */
     write("Загадайте животное, а я попытаюсь его
     отгадать"),nl,
     animals, /* попытка отгадать загаданное животное */
     retractall(_,dialog), /* очищаем текущую
                              информацию */
     retractall(_,knowledge),
                    /* очищаем информацию об известных
                       животных и свойствах */
     nl,nl,write("Хотите еще раз сыграть? (1 — Да,
     2 — Нет)"),
     read_true_char(C),
```

```
     C='1',!,start.
start:-
     nl,nl,write("Всего доброго! До новых встреч"),
     readchar(_).
animals:-
     rule(X,L),
     check(L),
     nl,write("Я думаю, это ",X),
     nl,write("Я прав? (1 — да, 2 — нет)"),
     read_true_char(C),C='1',!.
animals:-
     nl,write("Я не знаю, что это за животное"),nl,
     nl,write("Давайте добавим его в мою базу зна-
ний."),nl,
     update.
update:-
     nl,write("Введите название животного:"),
     readln(S),
     add_cond(L), /* указываем свойства животного */
     assert(rule(S,L),knowledge), /* добавляем информацию
                                     в базу знаний*/
     save("animals.ddb",knowledge) /* сохраняем базу
                                     знаний в файл */.
add_cond(L):-
          cond_is(_,'1'),!, /* имеется информация
                               о свойствах животного */
          nl,write("О нем известно, что оно: "),
          print_cond(1,[],L1),
                    /* вывод имеющейся информации о
                       животном*/
          nl,write("Известно ли о нем еще что-нибудь?
          (1 — да, 2 — нет)"),
          read_true_char(C),
          read_cond(C,L1,L).
add_cond(L):-
          read_cond('1',[],L).
print_cond(H,L,L):-
               not(cond(H,_)),!.
print_cond(H,L,L1):-
               cond_is(H,'1'),!,
               cond(H,T),
               H1=H+1,
```

```
                nl,write(T),
                print_cond(H1,[H  L],L1).
print_cond(H,L,L1):-
                H1=H+1,
                print_cond(H1,L,L1).
read_cond('1',L,L2):-
             ex_cond(1,L,L1,N),
             new_cond(N,L1,L2),!.
read_cond(_,L,L):-!.
ex_cond(N,L,L,N):-
             not(cond(N,_)),!.
ex_cond(N,L,L1,N2):-
             cond_is(N,_),!,
             N1=N+1,
             ex_cond(N1,L,L1,N2).
ex_cond(N,L,L1,N2):-
             cond(N,S),
             nl,write("Оно ",S,"? (1 — да, 2 — нет)»),
             read_true_char(A),
             wr_cond(A,N,L,L2),
             N1=N+1,
             ex_cond(N1,L2,L1,N2).
wr_cond('1',N,L,[N L]):-!.
wr_cond('2',_,L,L):-!.
new_cond(N,L,L1):-
                nl,write("Есть еще свойства? (1 — да,
                2- нет)"),
                read_true_char(A),
A='1',!,
                nl,write("Укажите новое свойство,
                которым обладает животное"),
                nl,write("в виде 'оно <описание
                нового свойства>'"), readln(S),
                assertz(cond(N,S)),
                          /* добавление нового свойства
                             в базу знаний */
                N1=N+1,
             new_cond(N1,[N  L],L1).
new_cond(_,L,L).
check([HT]):-
          test_cond(H),
          check(T).
```

```
check([]).
test_cond(H):-
          cond_is(H,'1'),!. /* в базе имеется информация
                               о наличии свойства */
test_cond(H):-
          cond_is(H,'2'),!,
          fail. /* в базе имеется информация
                   об отсутствии свойства */
test_cond(H):- /* в базе нет никакой информации о данном
                  свойстве, получаем ее у человека */
          cond(H,S),
          nl,write("Оно ",S,"? (1 — да, 2 — нет)»),
          read_true_char(A),
          assert(cond_is(H,A)),
          test_cond(H).
read_true_char(C):-
          readchar(C1),
          test(C1,C).
test(C,C):-
          '1'<=C,C<='2',!.
test(_,C):-
          write("Нажмите 1 или 2!"),nl,
          readchar(C1),
          test(C1,C).
GOAL
start
```

В идеале экспертная система должна уметь объяснять пользователю свое решение, а также то, почему она задает тот или иной вопрос. Попробуйте добавить в нашу экспертную систему механизм объяснения.

Две версии этой программы, основанной на правилах (реализующие, соответственно, прямую и обратную цепочки рассуждений), можно найти в книге Д. Марселлус, Программирование экспертных систем на Турбо-Прологе. — М.: Финансы и статистика, 1994. Однако эти программы умеют распознавать только тех животных, которые заложены в них изначально. Возможности динамического пополнения базы знаний они не имеют. Для добавления описания нового животного требуется модификация программного кода.

Кроме того, я рекомендую читателям изучить программу GEOBASE, которая входит в состав и Турбо Пролога, и Visual Prolog. Эта программа содержит информацию по географии США и позволяет создавать запросы к базе данных на естественном (английском) языке.

Список литературы

1. А.Н. Адаменко, А.М. Кучуков. Логическое программирование и Visual Prolog. — СПб.: БХВ-Петербург, 2003.
2. И. Братко. Программирование на языке Пролог для искусственного интеллекта. —М.: Мир, 1990.
3. Дж. Доорс, А. Рейблейн, С. Вадера. Пролог — язык программирования будущего. —М.: Финансы и статистика, 1990.
4. А.В. Гаврилов, Ю.В. Новицкая. Основы программирования на Турбо Прологе — Новосибирск: НГТУ,1 993.
5. Ц. Ин, Д. Соломон. Использование Турбо-Пролога. —М.: Мир, 1993.
6. К. Кларк, Ф. Маккейб. Введение в логическое программирование на микро-Прологе. —М.: Радио и связь, 1987.
7. У. Клоксин, К. Меллиш. Программирование на языке Пролог. —М.: Мир, 1987.
8. А.Н. Адаменко, А.М. Кучуков. Логическое программирование и Visual Prolog. — Санкт-Петербург: БХВ-Петербург, 2003.
9. Дж. Малпас. Реляционный язык Пролог и его применение. —М.: Наука, 1990.
10. Д. Марселлус. Программирование экспертных систем на Турбо-Прологе. —М.: Финансы и статистика, 1994.
11. Л. Стерлинг, Э. Шапиро. Искусство программирования на языке Пролог. —М.: Мир, 1990.
12. Дж. Стобо. Язык программирования Пролог. — М.: Радио и связь, 1993.
13. К. Хоггер. Введение в логическое программирование. —М.: Мир, 1988.
14. Язык Пролог в пятом поколении ЭВМ: Сб. статей 1983—1986 гг.: Пер. с англ./ Сост. Н.И. Ильинский. —М.: Мир, 1990.
15. А. Янсон. Турбо-Пролог в сжатом изложении. —М.: Мир, 1991.

Серия «Основы информатики и математики»

1. **Преподавание информатики и математических основ информатики**, под. ред. А.В. Михалева, 2005, 144 с., ISBN 5-9556-0037-X.
2. **Начала алгебры, часть I**, А.В. Михалев, А.А. Михалев, 2005, 272 с., ISBN 5-9556-0038-8.
3. **Основы программирования**, В.В. Борисенко, 2005, 328 с., ISBN 5-9556-0039-6.
4. **Работа с текстовой информацией. Microsoft Office Word 2003**, О.Б. Калугина, В.С. Люцарев, 2005, 152 с., ISBN 5-9556-0041-8.

Серия «Основы информационных технологий»

1. **Основы Web-технологий**, П.Б. Храмцов и др., 2003, 512 с., ISBN 5-9556-0001-9.
2. **Основы сетей передачи данных**, В.Г. Олифер, Н.А. Олифер, 2005, 176 с., ISBN 5-9556-0035-3.
3. **Основы информационной безопасности**, 2-е издание, В.А. Галатенко, 2004, 264 с., ISBN 5-9556-0015-9.
4. **Основы микропроцессорной техники**, 2-е издание, Ю.В. Новиков, П.К. Скоробогатов, 2004, 440 с., ISBN 5-9556-0016-7.
5. **Язык программирования Си++**, 2-е издание, А.Л. Фридман, 2004, 264 с., ISBN 5-9556-0017-5.
6. **Программирование на Java**, Н.А. Вязовик, 2003, 592 с., ISBN 5-9556-0006-X.
7. **Стандарты информационной безопасности**, В.А. Галатенко, 2004, 328 с., ISBN 5-9556-0007-8.
8. **Основы функционального программирования**, Л.В. Городняя, 2004, 280 с., ISBN 5-9556-0008-6.
9. **Программирование в стандарте POSIX**, В.А. Галатенко, 2004, 560 с., ISBN 5-9556-0011-6.
10. **Введение в теорию программирования**, С.В. Зыков, 2004, 400 с., ISBN 5-9556-0009-4.
11. **Основы менеджмента программных проектов**, И.Н. Скопин, 2004, 336 с., ISBN 5-9556-0013-2.
12. **Основы операционных систем**, 2-е издание, В.Е. Карпов, К.А. Коньков, 2004, 536 с., ISBN 5-9556-0044-2.
13. **Основы SQL**, Л.Н. Полякова, 2004, 368 с., ISBN 5-9556-0014-0.

14. **Архитектуры и топологии многопроцессорных вычислительных систем**,
А.В. Богданов, В.В. Корхов, В.В. Мареев, Е.Н. Станкова, 2004, 176 с., ISBN 5-9556-0018-3.
15. **Операционная система UNIX**,
Г.В. Курячий, 2004, 320 с., ISBN 5-9556-0019-1.
16. **Основы сетевой безопасности: криптографические алгоритмы и протоколы взаимодействия**,
О.Р. Лапонина, 2005, 608 с., ISBN 5-9556-0020-5.
17. **Программирование в стандарте POSIX. Часть 2**,
В.А. Галатенко, 2005, 384 с., ISBN 5-9556-0021-3.
18. **Интеграция приложений на основе WebSphere MQ,**
В.А. Макушкин, Д.С. Володичев, 2005, 336 с., ISBN 5-9556-0031-0.
19. **Стили и методы программирования,**
Н.Н. Непейвода, 2005, 320 с., ISBN 5-9556-0023-X.
20. **Основы программирования на PHP,**
Н.В. Савельева, 2005, 264 с., ISBN 5-9556-0026-4.
21. **Основы баз данных**,
С.Д. Кузнецов, 2005, 488 с., ISBN 5-9556-0028-0.
22. **Интеллектуальные робототехнические системы**,
В.Л. Афонин, В.А. Макушкин, 2005, 208 с., ISBN 5-9556-0024-8.
23. **Программирование на языке Pascal**,
Т.А. Андреева, 2005, 240 с., ISBN 5-9556-0025-6.
24. **Основы тестирования программного обеспечения**,
В.П. Котляров, 2005, 360 с., ISBN 5-9556-0027-2.
25. **Программирование на языке Си**
Н.И. Костюкова, Н.А. Калинина, 2005, 224 с., ISBN 5-9556-0026-4.
26. **Основы локальных сетей**,
Ю.В. Новиков, С.В. Кондратенко, 2005, 360 с., ISBN 5-9556-0032-9.
27. **Операционная система Linux**,
Г.В. Курячий, К. Маслинский, 2005, 400 с., ISBN 5-9556-0029-9.
28. **Проектирование информационных систем**,
В.И. Грекул и др., 2005, 296 с., ISBN 5-9556-0033-7.
29. **Основы программирования на языке Пролог**,
П.А. Шрайнер, 2005, 176 с., ISBN 5-9556-0034-5.
30. **Операционная система Solaris**,
Ф.И. Торчинский, 2005, 472 с., ISBN 5-9556-0022-1.

Книги издательства Интернет-Университета Информационных Технологий всегда можно заказать на сайте: **shop.intuit.ru**
По вопросам оптовых закупок звоните **(095) 253-9312**.
Адрес: Россия, Москва 123056, Электрический пер., дом 8, строение 3.

Серия «Основы информационных технологий»

П.А. Шрайнер
Основы программирования на языке Пролог
Курс лекций. Учебное пособие

Литературный редактор Е. Петровичева
Корректор Ю. Голомазова
Компьютерная верстка О. Рыкалова
Обложка М. Автономова

Формат 60x90 $^{1}/_{16}$. Усл. печ. л. 11. Бумага офсетная.
Подписано в печать 29.04.2005. Тираж 2000 экз. Заказ № .

Санитарно-эпидемиологическое заключение о соответствии
санитарным правилам №77.99.02.953.Д.006052.08.03 от 12.08.2003

ООО «ИНТУИТ.ру»
Интернет-Университет Информационных Технологий, www.intuit.ru
123056, Москва, Электрический пер., д. 8, стр. 3.

Отпечатано с готовых диапозитивов на ФГУП ордена «Знак Почета»
Смоленская областная типография им. В.И. Смирнова.
Адрес: 214000, г. Смоленск, проспект им.Ю. Гагарина, д. 2.