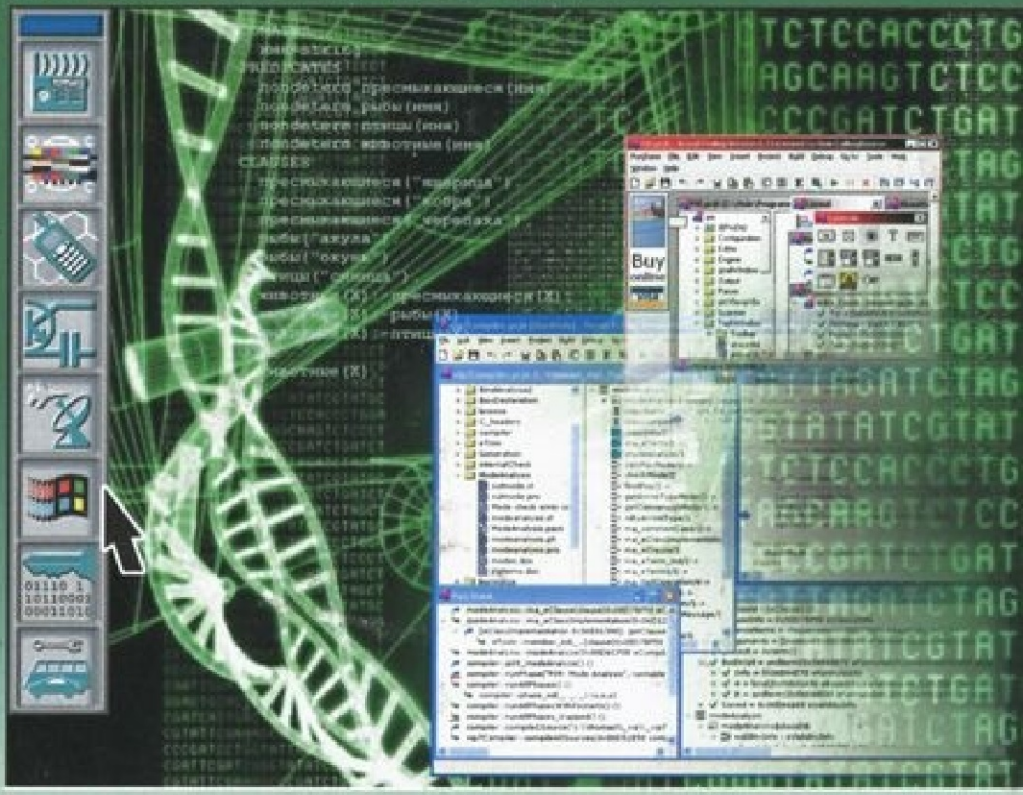
УЧЕБНОЕ ПОСОБИЕ

ДЛЯ ВЫСШИХ УЧЕБНЫХ ЗАВЕДЕНИЙ

СПЕЦИАЛЬНОСТЬ



Теория и практика логического программирования на языке Visual Prolog 7

Н. И. Цуканова
Т. А. Дмитриева

Горячая линия–Телеком

Н. И. Цуканова
Т. А. Дмитриева

Теория и практика логического программирования на языке Visual Prolog 7

Допущено УМО вузов по университетскому политехническому образованию в качестве учебного пособия для студентов высших учебных заведений, обучающихся по специальности 230105 – «Программное обеспечение вычислительной техники и автоматизированных систем»

Москва
Горячая линия - Телеком
2013

УДК 004.434:004.8
ББК 32.973.26-018.1
Ц85

Р е ц е н з е н т : доктор техн. наук, профессор *И. В. Солодовников*

Цуканова Н. И., Дмитриева Т. А.

Ц85 Теория и практика логического программирования на языке Visual Prolog 7. Учебное пособие для вузов. – М.: Горячая линия – Телеком, 2013. – 232 с.: ил.

ISBN 978-5-9912-0194-0.

Рассмотрены теоретические основы логического программирования. Даны примеры и описание предметной области с помощью логических моделей. Показана связь базовых понятий логики предикатов и основных конструкций языка логического программирования Пролог. Изложены основы логического программирования на примере языка Visual Prolog 7. Рассмотрены структура программы, алгоритм работы интерпретатора, ввод – вывод, приемы и средства организации интерактивных программ, вопросы недетерминированного программирования и управления выполнением программы, различные структуры данных и предикаты работы с ними. Книга содержит многочисленные примеры, а также контрольные вопросы и практические задания. Пособие будет полезно при изучении курса «Функциональное и логическое программирование».

Для студентов высших учебных заведений, программистов, специалистов в области искусственного интеллекта и баз данных.

ББК 32.973.26-018.1

Цуканова Нина Ивановна, **Дмитриева** Татьяна Александровна

Теория и практика логического программирования на языке Visual Prolog 7

Учебное пособие для вузов

Обложка художника В. Г. Ситникова

Подписано в печать 01.01.2013. Формат 60x90/16. Гарнитура Times. Усл.печ. л. 14,5. Тираж 1000 экз. (2-й завод 100 экз.)
ООО «Научно-техническое издательство «Горячая линия-Телеком»

ISBN 978-5-9912-0194-0

ПРЕДИСЛОВИЕ

В последние годы значительно возрос интерес к работам по искусственному интеллекту, особенно к практическому воплощению результатов этих работ в экспертных системах. Одной из наиболее важных задач, связанных с проектированием таких систем, является проблема представления знаний о предметной области. Исследования развиваются в двух направлениях: первое связано с выбором формализованных моделей представления знаний; второе – с разработкой и внедрением языков представления знаний, обладающих необходимыми изобразительными средствами.

В настоящее время большую популярность приобрел язык логического программирования Пролог, основанный на самой старой модели представления знаний – исчислении предикатов первого порядка. Само название Пролог есть сокращение, означающее программирование на языке логики.

Пролог предоставляет средства для описания знаний о предметной области в виде фактов и правил вывода. Он понятен экспертам-непрограммистам и даже с успехом может использоваться для обучения детей программированию. Пролог способствует формированию стиля программирования, основанного на идеях модульного программирования и программирования "сверху вниз". В то время как традиционные языки программирования являются процедурно-ориентированными, Пролог основан на описательной или декларативной парадигме в программировании. Это свойство коренным образом меняет программистское мышление и делает обучение программированию на Прологе увлекательным занятием, требующим определенных интеллектуальных усилий.

Пролог применяется при создании приложений в следующих областях:

- разработка быстрых прототипов прикладных программ;
- разработка приложений, связанных с защитой информации;
- управление производственными процессами;
- создание динамических реляционных баз данных;
- перевод с одного языка на другой;
- реализация экспертных систем и оболочек экспертных систем.

Системы программирования на Прологе эксплуатируются на ЭВМ самых разных типов и приобрели широкую известность благодаря наличию таких развитых версий на персональных компьютерах как WinProlog, Visual Prolog, Strawberry Prolog и Arity Prolog. В пособии рассматриваются примеры, подготовленные в среде Visual Prolog 5.2, 7.0 – 7.3, приводятся правила и пример создания законченного приложения с графическим интерфейсом в среде Visual Prolog 7.0 – 7.3. Эти версии являются в настоящее время наиболее развитыми промышленными версиями, работающими в среде ОС Windows.

Пособие делится на одиннадцать глав. В гл. 1 описываются теоретические основы логического программирования: вводится понятие формальной системы, затем рассматриваются такие формальные системы как исчисление высказываний и исчисление предикатов, а также алгоритм вывода, основанный на принципе резолюции. В гл. 2 рассматриваются базовые понятия языка логического программирования, структура программы на Прологе и правила составления программ. В гл. 3 описывается алгоритм работы интерпретатора Пролога. Подробно рассматривается каждая фаза циклического процесса доказательства текущей цели. Предлагается для анализа выполнения программы использовать схему в виде И-ИЛИ-дерева. В гл. 4 вводится понятие встроенного предиката, затем рассматриваются встроенные предикаты ввода-вывода термов, а также арифметические выражения и предикаты сравнения термов.

Пятая, шестая и седьмая главы посвящены способам управления выполнением программы на Прологе. В гл. 5 рассматриваются способы организации ветвления и повторяющихся процессов. Гл. 6 посвящена рекурсии как одному из основных методов программирования на Прологе. В гл. 7 описывается управление процессом возврата с помощью встроенного предиката "отсечение".

Гл. 8–10 посвящены различным структурам данных в языке Пролог. В гл. 8 рассматриваются списки, их структура и приводятся примеры наиболее часто используемых предикатов работы со списками. В гл. 9 уделяется внимание строкам, для работы с ними в языке Visual Prolog разработано много встроенных предикатов. В гл. 9 рассматриваются структуры. Гл. 11 посвящена описанию среды Visual Prolog 7.0 – 7.3 и правилам создания в этой среде законченного приложения с использованием графического интерфейса.

В конце каждой главы приведены вопросы для самоконтроля и задания для самостоятельной работы, рассматривается пример выполнения одного из заданий.

Для читателя этой книги можно дать следующие рекомендации. Если читателю знаком язык Пролог и его цель познакомиться с правилами работы в новой среде Visual Prolog 7, то ему лучше сразу перейти к гл. 11. Если читатель – новичок и только начинает изучение языка, но теоретические основы его не интересуют, ему следует обратиться к гл. 2. А если читатель специалист по представлению знаний в интеллектуальных системах и его интересуют логические модели, а также если он очень любознательный, то рекомендуем ему прочитать гл. 1.

Авторы выражают благодарность рецензентам, сделавшим ряд ценных замечаний при подготовке данного пособия.

Глава 1. ВВЕДЕНИЕ В ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ЛОГИЧЕСКОГО ПРОГРАММИРОВАНИЯ

Язык Пролог разрабатывался как язык систем искусственного интеллекта, и в 1984 году в проекте вычислительных систем пятого поколения был объявлен как базовый язык [3].

Система искусственного интеллекта содержит два обязательных элемента – базу знаний (БЗ) и машину вывода [2], которые тесно связаны между собой положенной в их основу моделью представления знаний.

Процесс работы системы искусственного интеллекта упрощенно можно описать следующим образом. На вход системы поступает вопрос пользователя B. Машина вывода, опираясь на хранимые в БЗ знания общего характера о предметной области A_1, A_2, …, A_n и на данные, вводимые пользователем, осуществляет вывод ответа на поставленный вопрос.

Для представления знаний о предметной области используются различные модели [2]. Одна из них – это логическая модель представления знаний [1]. В соответствии с ней знания о предметной области описываются логическими формулами A_1, A_2, …, A_n и считаются истинными утверждениями – аксиомами. Эти формулы хранятся в базе знаний. Вопрос пользователя или цель решения задачи также выражается логической формулой B, которая является гипотезой, истинность или ложность ее доказывается машиной вывода. Машина вывода в процессе своей работы доказывает логическое следование B из истинности A_1, A_2, …, A_n.

Для того чтобы знания описывать на языке логики в наиболее естественной форме для пользователя-непрофессионала, был разработан язык Пролог. Для написания программы на этом языке надо знать и понимать всего три конструкции: факт, правило, вопрос. С их помощью можно описать все базовые логические операции: конъюнкцию, дизъюнкцию, импликацию и отрицание. Кроме того, Пролог позволяет писать программу на близком к естественному для пользователя языке (например, на русском языке). В основе языка Пролог лежит исчисление предикатов первого порядка, а именно, логические конструкции, приводимые к Хорновским дизъюнктам. Чтобы понять выразительные возможности основных утверждений языка, как работает машина вывода и что такое Хорновские дизъюнкты, вам следует внимательно прочитать эту первую главу. В ней рассматриваются теоретические основы языка Пролог.

1.1. Формальная система

В формальной логике разрабатываются методы правильных рассуждений, представляющих собой цепь умозаключений. Для описания рассуждений и лежащих в их основе закономерностей используются логические модели, базирующиеся на понятии формальной системы.

Формальная система задаётся четвёркой вида

$M =< T, P, A, B >$

Множество T есть **множество базовых элементов** различной природы, например слов из некоторого ограниченного словаря, студентов некоторой группы, звуков определённого инструмента и т.д. Важно, что для множества T существует некоторый способ определения принадлежности или не принадлежности произвольного элемента к этому множеству. Процедура такой проверки может быть любой, но за конечное число шагов она должна давать положительный или отрицательный ответ на вопрос, является ли x элементом множества T. Обозначим эту процедуру $П(T)$.

Множество P есть множество синтаксических правил. С их помощью из элементов T образуют **синтаксически правильные совокупности** (правильно построенные формулы – ППФ). Например, из слов ограниченного словаря строятся синтаксически правильные фразы и т.д. Декларируется существование процедуры П(Р), с помощью которой за конечное число шагов можно получить ответ на вопрос, является ли совокупность X **синтаксически правильной**.

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

Множество B есть множество правил вывода. Применяя их к элементам A, можно получить новые синтаксически правильные совокупности, к которым снова можно применить правила из B. Так формируется множество **выводимых** в данной формальной системе совокупностей ППФ. Если имеется процедура $П(B)$, с помощью которой можно определить для любой синтаксически правильной совокупности (ППФ), является ли она выводимой, то соответствующая формальная система называется разрешимой. Это показывает, что именно правила вывода являются наиболее сложной составляющей формальной системы.

Для знаний, входящих в базу знаний (БЗ), можно считать, что множество A образуют все информационные единицы, которые введены в базу знаний извне (экстенсиональная БЗ), а с помощью правил вывода из них выводятся новые производные знания (интенсиональная БЗ). Формальная система представляет собой генератор порождения новых знаний, образующих множество выводимых в данной системе знаний. Это позволяет хранить в БЗ лишь те знания, которые образуют множество A, а все остальные получать путём вывода.

Формальное доказательство

Формальное доказательство [1] определяется как конечная последовательность формул ППФ M_1, M_2, …, M_r, такая, что каждая формула M_i либо является аксиомой, либо при помощи одного из правил вывода выводима из предшествующих ей формул M_j, где $j < i$. Формула t называется **теоремой**, если существует доказательство, в котором она является последней, т.е. $M_r = t$. В частности, всякая аксиома является теоремой. Если t есть теорема, то этот факт обозначается $\vdash t$, а знак $\vdash$ является символом выводимости (получения) некоторой ППФ с помощью правил вывода [1].

Интерпретация

Формальные системы (ФС) используются как модели некоторой реальности. Интерпретация представляет собой распространение исходных положений какой-либо формальной системы на реальный мир (см. рис. 1.1).

Рис. 1.1. Интерпретация формальной системы

Интерпретация придаёт смысл каждому символу формальной системы и устанавливает взаимно однозначное соответствие между символами ФС и реальными объектами. Теоремы ФС, будучи однажды интерпретированы, становятся после этого **утверждениями** в обычном смысле слова, и в этом случае уже можно делать выводы об их **истинности** или **ложности**. Формальная система может служить моделью многочисленных различных конкретных ситуаций. Необходимо, чтобы

для каждой формальной системы всегда существовала, по крайней мере, одна интерпретация, в которой каждая теорема ФС была бы истинной. Чем больше интерпретаций, тем богаче ФС [1].

Доказательство и истинность

Имеются 4 варианта (рис. 1.2) взаимоотношений между доказательством и значением истинности, поскольку формула, с одной стороны, может быть теоремой (Т) либо не теоремой (НТ), а с другой стороны, её интерпретация может быть истинной (И) или ложной (Л).

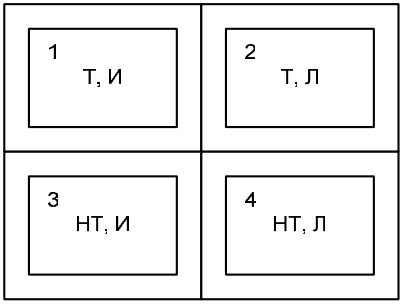


Рис. 1.2. Связь между свойствами выводимости и истинности ППФ

Если для любой интерпретации соблюдаются первый и четвертый варианты, то выбранная ФС хорошо описывает предметную область. Но это не всегда возможно.

Второй вариант, когда формула доказуема и соответствует определенной интерпретации, значение которой ложно, говорит о том, что нет соответствия между ФС и предметной областью. Следует либо заменить ФС, либо исключить данную интерпретацию из рассмотрения.

Наиболее сложный случай – третий. Ситуацию (НТ, И) трудно идентифицировать, т.к. не всегда есть процедура, позволяющая установить, что формула, описывающая истинную интерпретацию, не выводима в данной ФС.

Свойства формальных систем

1. **Разрешимость**. Если существует процедура $П(B)$, позволяющая для любой ППФ установить, выводима ли она в данной ФС (является теоремой), то такая система называется **разрешимой**.

2. **Полнота в широком смысле слова**. ФС называется полной в широком смысле слова, если любая ***выводимая*** ППФ общезначима (т.е. истинна в любой интерпретации) и, наоборот, любая общезначи-

мая ППФ является теоремой данной формальной системы (случай (Т, И)).

3. **Полнота в узком смысле слова**. ФС называется полной в узком смысле слова, если присоединение к аксиомам не выводимых в ФС ППФ приводит к противоречию (случай (НТ, И)).

4. **Непротиворечивость**. Любое логическое исчисление называется непротиворечивым, если в нём не выводимы никакие две ППФ, из которых одна является отрицанием другой.

5. **Независимость аксиом**. Таким свойством обладает ФС, у которой все аксиомы независимы, т.е. не выводимы с помощью правил вывода из других аксиом.

Пример формальной системы

Пример 1.1. Формальная система ДН.

Алфавит: M , I , U .

Синтаксис: ППФ – это любая последовательность символов данного алфавита.

Одна аксиома: MI .

Правила вывода:

правило 1: $mI \to mIU$; правило 2: $Mm \to Mmm$;

правило 3: $III \to U$; правило 4: $UU \to$;

где m – ППФ.

Примеры теорем:

а) MI (аксиома);

б) MII (правило 2 применено к а);

в) $MIII$ (правило 2 применено к б);

г) $MIIIU$ (правило 1 применено к в);

д) $MIUU$ (правило 3 к применено г);

е) MI (правило 4 к применено д).

1.2. Исчисление высказываний как формальная система

Исчисление высказываний (ИВ)

В исчислении высказываний рассматриваются формулы, составленные из элементарных формул, соединённых некоторыми связками. Элементарные формулы интерпретируются как простые высказывания, записанные в виде повествовательных предложений естественного языка и принимающие два значения: истина (И) или ложь (Л). Если простое высказывание X истинно, то говорят, что X принимает значение И, если X ложно – то Л. Символы И и Л называются истинностными значениями [1].

Сложное высказывание имеет истинностное значение, однозначно определяемое истинностными значениями простых высказываний, из которых оно составлено. Например: "Если студент ложится поздно спать и пьет на ночь кофе, то утром он встает в дурном расположении духа или с головной болью". Это сложное высказывание состоит из следующих простых высказываний:

"Студент ложится поздно спать";

"Студент пьет на ночь кофе";

"Утром студент встает в дурном расположении духа";

"Утром студент встает с головной болью".

Обозначим сложное высказывание через X, а простые – соответственно через Y, Z, U, V. Тогда сложное высказывание запишется в виде X = если Y и Z, то U или V.

Для обозначения используемых в этом примере связок введем специальные символы: "если…то" $\rightarrow$, "и" $\&$, "или" $\cup$. Тогда в символическом виде высказывание X запишется так: $X = (Y \,\&\, Z) \rightarrow \rightarrow (U \cup V)$.

Каждую логическую связку можно рассматривать как операцию, которая образует новое высказывание – сложное высказывание из более простых высказываний. Рассматриваются следующие логические связки (операции):

- отрицание $\overline{X}$; $\overline{X}$ истинно, когда X ложно, и ложно, когда X истинно;
- конъюнкция $X \,\&\, Y$; $X \,\&\, Y$ истинно, когда X и Y оба истинны, в противном случае $X \,\&\, Y$ ложно;
- дизъюнкция $X \cup Y$; $X \cup Y$ ложно, когда X и Y оба ложны, в противном случае $X \cup Y$ истинно;
- импликация $X \rightarrow Y$; $X \rightarrow Y$ ложно, когда X истинно, а Y ложно, в противном случае $X \rightarrow Y$ истинно ($X \rightarrow Y$ читается, как "если X, то Y" или "X влечет Y");
- эквивалентность $X \leftrightarrow Y$; $X \leftrightarrow Y$ истинно, когда X и Y имеют одинаковые истинностные значения, в противном случае $X \leftrightarrow Y$ ложно.

Всякое сложное высказывание можно записать в виде некоторой формулы, содержащей логические связки и символы, которые обозначают простые высказывания, называемые **атомами**. Для того чтобы узнать, истинно или ложно сложное высказывание, достаточно узнать истинностные значения всех атомов, из которых оно составлено. Для

этого составляется истинностная таблица, как, например, для выражения $(X \cup \overline{Y}) \to Z$ (табл. 1.1).

Табл. 1.1

X	Y	Z	$\overline{Y}$	$X \cup \overline{Y}$	$(X \cup \overline{Y}) \to Z$
Л	Л	Л	И	И	Л
Л	Л	И	И	И	И
Л	И	Л	Л	Л	И
Л	И	И	Л	Л	И
И	Л	Л	И	И	Л
И	Л	И	И	И	И
И	И	Л	Л	И	Л
И	И	И	Л	И	И

Пусть дана формула B и X_1, X_2, …, X_n – атомы, в ней встречающиеся. Тогда под интерпретацией формулы B понимается приписывание истинностных значений атомам X_1, X_2, …, X_n. Говорят, что формула B истинна в данной интерпретации тогда и только тогда, когда B принимает значение И в этой интерпретации; в противном случае говорят, что B ложна в этой интерпретации. Если имеется n различных атомов в формуле, то существует 2^n различных интерпретаций для этой формулы.

Формула ИВ, которая истинна во всех интерпретациях, называется **тавтологией**, или **общезначимой** формулой. Примерами тавтологий являются $X \cup \overline{X}$, $X \to (Y \to X)$. Чтобы в этом убедиться, необходимо рассмотреть их истинностные таблицы.

Формула ИВ называется **противоречием**, или тождественно ложной, если она принимает значение Л во всех интерпретациях. Например, $X \leftrightarrow \overline{X}$, $X \,\&\, \overline{X}$.

Истинностные таблицы дают эффективную процедуру для решения вопроса о том, является ли данная формула общезначимой или противоречием. Однако составление истинностных таблиц, особенно для сложных формул, является утомительным и непрактичным делом. Поэтому для образования тавтологий из тавтологий используются правила вывода, рассмотренные далее.

Исчисление высказываний как формальная система

Базовыми элементами (термами) ИВ являются прописные буквы латинского или русского алфавита с индексами или без них. Исходные термы называются атомами. Символами логических связок являются

$\bar{}$, $\&$, $\cup$, $\to$, которые интерпретируются как отрицание, конъюнкция, дизъюнкция и импликация. К базовым элементам относятся также скобки), (. Других символов множества T ИВ не имеет [1].

Правила образования ППФ:

а) все атомы есть ППФ;

б) если A и B – ППФ, то $\bar{A}$, $A\&B$, $A\cup B$, $A\to B$ – также ППФ;

в) других правил образования ППФ нет.

Система аксиом (П.С. Новиков):

a_1 $A\to(B\to A)$;

a_2 $(A\to(B\to C))\to((A\to B)\to(A\to C))$;

a_3 $A\&B\to A$;

a_4 $A\&B\to B$;

a_5 $(A\to B)\to((A\to C)\to(A\to B\&C))$;

a_6 $A\to A\cup B$;

a_7 $B\to A\cup B$;

a_8 $(A\to C)\to((B\to C)\to(A\cup B\to C))$;

a_9 $(A\to B)\to(\bar{B}\to\bar{A})$;

a_{10} $A\to\bar{\bar{A}}$;

a_{11} $\bar{\bar{A}}\to A$.

Все аксиомы – общезначимые ППФ.

Правила вывода.

1. Все аксиомы выводимы.

2. Правило подстановки (ПС). Пусть A – ППФ, содержащая атом X. Тогда, если A – теорема исчисления высказываний, то, заменив в ней атом X всюду, куда он входит, на произвольную ППФ B, мы получим также теорему.

3. Правило *modus ponens* (МР) (правило дедуктивного вывода). Если A и $A\to B$ – теоремы, то B – также теорема.

4. Других правил вывода нет.

Исчисление высказываний обладает следующими свойствами: полнотой в широком и узком смысле, непротиворечивостью и независимостью аксиом.

Любая теорема ИВ является общезначимой, т.к. все аксиомы тавтологии и правила вывода, примененные к тавтологиям, всегда приводят к тавтологиям. Обратное утверждение также справедливо. Если ППФ A ИВ является общезначимой, то она есть теорема этой системы.

Свойство полноты устанавливает равноценность понятий общезначимости и доказуемости.

Свойство непротиворечивости также является достоинством ИВ. Если система противоречива, то в ней можно вывести любые ППФ, что не представляет никакой ценности, т.к. в ней не будет различий между истиной и ложью.

Исчисление высказываний является **разрешимой системой**. Процедурой $П(B)$, устанавливающей для любой ППФ, теорема она или нет, является построение таблицы истинности для этой формулы. Если при любой интерпретации формула имеет значение И, она общезначима и по теореме о полноте является теоремой.

Интерпретация ИВ

Для того чтобы использовать методы логики высказываний применительно к конкретной области знаний (так называемой "предметной области"), сначала необходимо проанализировать структуру этой области. При выполнении анализа отыскиваются атомарные (простые) высказывания, действующие в указанной области, и им подбираются соответствующие обозначения A, B, C и т.д. Затем логические взаимосвязи между атомарными высказываниями выражаются с помощью знаков операций $\bar{}$, $\&$, $\cup$, $\rightarrow$, $\leftrightarrow$. В результате получается совокупность ППФ, описывающая данную предметную область. Множество таких ППФ называется **теорией** заданной области знаний, а каждая отдельная ППФ именуется **аксиомой**.

Цель теории – описать нужные знания экономичным способом. Если теория адекватно описывает предметную область, то любой факт, полученный путем применения правил вывода ИВ, является следствием аксиом теории и истинным в данной предметной области. Выполнив обратный процесс интерпретации, т.е. приписав обозначениям соответствующие атомарные высказывания, получим истинные в данной области новые утверждения.

Пример постановки задачи

Задача. Если я поеду автобусом, и автобус опоздает, то я опоздаю на работу. Если я опоздаю на работу и стану огорчаться, то я не попадусь на глаза моему начальнику. Если я не сделаю в срок важную работу, то я начну огорчаться и попадусь на глаза моему начальнику. Следовательно, если я поеду автобусом, а автобус опоздает, то я сделаю в срок важную работу.

Атомарные высказывания для первого предложения:

A – я поеду автобусом; B – автобус опоздает; C – я опоздаю на работу.

1. $A \& B \to C$ – формула (ППФ), описывающая первое предложение.

Атомарные высказывания второго предложения:

D – я стану огорчаться; E – я попадусь на глаза моему начальнику.

2. $C \& D \to \overline{E}$ – формула (ППФ), описывающая второе предложение.

Атомарные высказывания третьего предложения:

F – я сделаю в срок важную работу.

3. $\overline{F} \to D \& E$ – формула (ППФ), описывающая третье предложение.

4. $A \& B \to F$ – формула (ППФ), описывающая четвертое предложение.

Показать, что если ППФ 1, 2, 3, истинны, то истинна и ППФ 4

$[(A \& B \to C) \& (C \& D \to \overline{E}) \& (\overline{F} \to D \& E)] \to (A \& B \to F)$.

1.3. Исчисление предикатов первого порядка как формальная система

Основные понятия исчисления предикатов первого порядка

Более мощными выразительными средствами по сравнению с исчислением высказываний обладает язык исчисления предикатов первого порядка. Он позволяет представить такие обобщающие утверждения, как "Всякое положительное число есть натуральное число", а также описать внутреннюю структуру высказывания, истинностное значение которого зависит от входящих в него компонент. Рассмотрим основные понятия этого исчисления [1].

Пусть задано некоторое множество $V = \{v_1, v_2, \ldots, v_k, \ldots\}$, в котором v_1, v_2, …, v_k, … – какие-то определенные объекты предметной области. Обозначим любой предмет (объект) из этого множества через x и назовем x предметной переменной. Тогда высказывания об этих предметах (объектах) будем обозначать в виде $P(v_1)$, $Q(v_1, v_2)$ и т.д., причем такие высказывания могут быть как истинными, так и ложными. Например, если $V = \{1,2,3,\ldots\}$, то высказывание "4 есть четное число" является истинным, а "7 есть четное число" – ложным. Если вместо конкретных чисел 4, 7 и т.д. подставим предметную переменную x, то получим **предикат** "x есть четное число", обозначаемый $P(x)$.

Предикат на множестве V есть логическая функция, определенная на V, при фиксировании аргументов которой она превращается в высказывание со значениями {И, Л}. В общем случае через $P(x_1, x_2, \ldots, x_n)$ обозначается n-местный предикат, обладающий тем свойством, что, приписав значения переменным x_1, x_2, …, x_n из соответствующих областей определения, получим высказывание со значениями {И, Л}. С помощью предикатов можно описывать связи (отношения) между объектами(или объектами и их свойствами) предметной области. Если такая связь имеет место в предметной области, то предикат истинен, если нет такой связи, то предикат ложен.

Важную роль в исчислении предикатов (ИП) играют две связки: $\forall$ – квантор общности и $\exists$ – квантор существования. Формула $\forall x\, P(x)$ описывает утверждение: "Все x из области V обладают свойством P ", а формула $\exists x\, P(x)$ – "Существует предмет x в области V, обладающий свойством P ".

Если A – формула ИП, то в выражениях $\forall x\, A$, $\exists x\, A$ формула A является областью действия квантора $\forall x$, $\exists x$.

Вхождение переменной x в формулу называется связанным (а сама переменная – связанной), если x является переменной входящего в эту формулу квантора $\forall x$ (или $\exists x$), в противном случае вхождение переменной в данную формулу называется свободным (а сама переменная – с вободной).

В общем случае формула $\forall x_{i1} \forall x_{i2} \ldots \forall x_{ir}\, A(x_1, x_2, \ldots, x_n)$, где x_{i1}, x_{i2}, …, x_{ir} являются подмножеством переменных x_1, x_2, …, x_n, будет пониматься как утверждение, что для **любых** значений, придаваемых предметным переменным x_{i1}, x_{i2}, …, x_{ir} из области определения $A(x_1, x_2, \ldots, x_n)$, истинность A зависит только от свободных переменных, входящих в эту формулу. Аналогично $\exists x_{i1} \exists x_{i2} \ldots \exists x_{ir}\, A(x_1, x_2, \ldots, x_n)$ понимается как утверждение, что существуют значения переменных x_{i1}, x_{i2}, x_{i3},… такие, что истинность $A(x_1, x_2, \ldots, x_n)$ зависит лишь от свободных переменных, входящих в эту формулу . Если к формуле $A(x_1, x_2, \ldots, x_n)$ применяем n раз какие-либо кванторы, то получаем выражение, представляющее собой некоторое постоянное высказывание, которое называется **замкнутой** формулой, т.е. формулой без свободных переменных.

Рассмотрим выражение $\overline{\forall x\, A(x)}$, которое означает, что существует, по крайней мере, один предмет x из области определения $A(x)$, для которого $A(x)$ ложно, т.е.

$\overline{\forall x\, A(x)} = \exists x\, \overline{A(x)}$.

Аналогично можно показать [1], что

$\overline{\exists x\, A(x)} = \forall x\, \overline{A(x)}$.

Кванторы общности и существования двойственны друг другу.

Исчисление предикатов первого порядка как формальная система

Рассмотрим исчисление предикатов первого порядка как формальную систему [1].

Базовые элементы ИП:

1) счетное множество предметных переменных $x_1, x_2, \ldots, x_n, \ldots$;

2) конечное (может быть пустое) или счетное множество предметных констант $a_1, a_2, \ldots, a_n, \ldots$;

3) конечное (может быть пустое) или счетное множество функциональных букв $f_{11}, f_{12}, \ldots, f_{kr}, \ldots$;

4) непустое конечное или счетное множество предикатных букв (заглавных букв латинского или русского алфавитов) $A_{11}, A_{12}, \ldots, A_{nl}, \ldots, P_{rk}, \ldots$;

5) символы исчисления высказываний : $\bar{}$, &, $\cup$, $\rightarrow$;

6) скобки), (и запятая;

7) символы $\forall$ и $\exists$;

8) других базовых элементов нет.

Синтаксис ИП

Правила конструирования термов:

1) всякая предметная переменная или предметная константа есть терм;

2) если f_{in} – функциональная буква, $t_1, \ldots, t_n$ – термы, то $f_{in}(t_1, \ldots, t_n)$ –есть терм;

3) других правил образования термов нет.

Правила образования атомов:

1) всякая предикатная буква A, описывающая простое высказывание, есть атом;

2) если A_{in} –предикатная буква, а $t_1, \ldots, t_n$ –термы, то $A_{in}(t_1, \ldots, t_n)$ есть атом;

3) других правил образования атомов нет.

Таким образом атом – это предикат.

Правила образования ППФ:

1) всякий атом есть ППФ;

2) если A и B – ППФ и x – предметная переменная, то каждое из выражений $\overline{A}$, $A \to B$, $A \& B$, $A \cup B$, $\forall x\, A$, $\exists x\, A$ есть ППФ;

3) других правил образования ППФ нет.

Система аксиом.

К системе аксиом исчисления высказываний добавляются еще 2 аксиомы:

1. $\forall x\, A(x) \to A(t)$, где $A(x)$ есть ППФ и t – терм, свободный для x в $A(x)$.

2. $A(t) \to \exists x\, A(x)$, где A(x) есть ППФ и t – терм, свободный для x в $A(x)$.

Правила вывода.

1. Все аксиомы выводимы.

2. Правило подстановки. Это правило аналогично правилу подстановки, которое имело место для исчисления высказываний. Только для ИП следует иметь дело с такой подстановкой термов $t_1, t_2, \ldots, t_k$ вместо $x_{i1}, x_{i2}, \ldots, x_{ik}$ в $A[x_{i1}, x_{i2}, \ldots, x_{ik}]$, что $A[x_{i1}, x_{i2}, \ldots, x_{ik}]$ свободна для $t_1, t_2, \ldots, t_k$. Например, подстановка $t = x_2$ вместо x_1 в формулу $\overline{\forall x_2\, A(x_1, x_2)}$ приведет к неправильным выводам.

3. Правило modus ponens (MP)

Если $\vdash A$ и $\vdash A \to B$, то $\vdash B$, где $\vdash$ – знак выводимости формулы.

4. Правило обобщения (или правило связывания квантором общности)

Если ППФ $B \to A(x)$ при условии, что B не содержит свободных вхождений x, выводима, то выводима будет и ППФ $B \to \forall x\, A(x)$.

5. Правило конкретизации (или правило связывания квантором существования)

Если $A(x) \to B$ –выводимая ППФ (теорема) и B не содержит свободных вхождений x, то $\exists x\, A(x) \to B$ также теорема.

6. Если A – теорема, имеющая квантор общности и/или квантор существования, то одна связанная переменная в A может быть заменена другой связанной переменной, отличной от всех свободных пе-

ременных, одновременно во всех областях действия квантора и в самом кванторе. Полученная ППФ также является теоремой.

7. Других правил вывода нет.

Интерпретация ИП

Под интерпретацией (см. рис. 1.3) в исчислении предикатов будем понимать всякую систему, состоящую из непустого множества V, называемого областью интерпретации и какого-либо соответствия, относящего каждой предикатной букве A_{ni} некоторое n-местное отношение в V (т.е. $V^n \to \{И, Л\}$), каждой функциональной букве f_{in} – некоторую n-местную функцию в V (т.е. $V^n \to V$) и каждой предметной константе a_i – некоторый элемент из V. При заданной интерпретации предметные переменные мыслятся пробегающими область V этой интерпретации, а логическим связкам $\bar{}$, $\to$, $\leftrightarrow$, $\cup$, & и кванторам $\exists$ и $\forall$ придается их обычный смысл. Для данной интерпретации любая формула без свободных переменных представляет собой высказывание (предложение), а всякая формула со свободными переменными выражает некоторое отношение, причем это отношение может быть истинным для одних значений переменных и ложным для других.

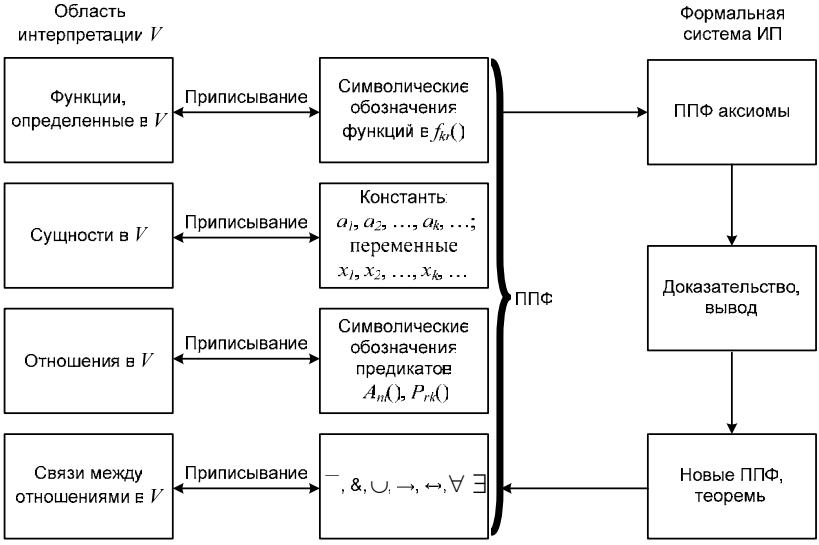


Рис. 1.3. Интерпретация исчисления предикатов

Для выяснения факта, истинна или ложна формула в данной интерпретации I, необходимо, задав область интерпретации, интерпретировать, прежде всего, все термы, входящие в формулу, затем атомы, и, наконец, саму формулу.

Пример 1.2. Пусть $P(x)$ и $N(x)$ обозначают соответственно "x есть положительное целое число" и "x есть натуральное число". Тогда утверждение "Всякое положительное целое число есть натуральное число. Число 5 есть положительное целое число. Следовательно, 5 есть натуральное число" будет записано на языке исчисления предикатов следующим образом.

$\forall x(P(x) \to N(x))$

$P(5)$

$\vdash N(5)$

Для перевода предложений с русского языка на язык ИП не существует механических правил. В каждом отдельном случае нужно сначала установить, каков смысл переводимого предложения, а затем пытаться передать тот же смысл с помощью предикатов, кванторов и термов.

Пример 1.3. Дана формула $\forall x\,(A(x) \to B(f(x), a))$. Имеем следующую интерпретацию: $V = \{1,2\}$;

$a = 2$;

$f(1) = 2$;

$f(2) = 1$;

$A(1)$	$A(2)$	$B(1,1)$	$B(1,1)$	$B(2,1)$	$B(2,2)$
И	Л	Л	Л	И	И

Если $x = 1$, то

$A(x) \to B(f(x), a) = A(1) \to B(f(1),2) = A(1) \to B(2,2) =$И

Если $x = 2$, то

$A(x) \to B(f(x), a) = A(2) \to B(f(2),2) = A(2) \to B(1,2) =$И

Следовательно, эта формула истинна в данной интерпретации.

Общезначимая формула (тавтология) в ИП истинна в любой интерпретации.

Противоречивая формула (противоречие) в ИП ложна в любой интерпретации.

Свойства исчисления предикатов первого порядка

Свойства ИП описываются следующими теоремами [1].

1. Исчисление предикатов первого порядка непротиворечиво.

2. Во всяком исчислении предикатов первого порядка всякая теорема является общезначимой ППФ (тавтологией).

3. Во всяком исчислении предикатов первого порядка всякая общезначимая ППФ (тавтология) является теоремой.

Таким образом, проблема полноты в широком смысле для ИП решается положительно. Что касается проблемы полноты в узком смысле, то исчисление предикатов является неполным, т.к. к его аксиомам можно присоединить без противоречия недоказуемую в нем следующую ППФ: $\exists x\, A(x) \to \forall x\, A(x)$.

Исчисление предикатов – пример неразрешимой формальной системы. В 1936 г. американским логиком Алонзо Черчем было доказано отсутствие эффективной процедуры определения для произвольной ППФ, является ли она теоремой.

1.4. Логические следствия

Описывая условия задачи в виде ППФ исчисления высказываний или исчисления предикатов, чаще всего получаем следующую ситуацию: несколько ППФ $A_1, A_2, \ldots, A_n$ (посылки) рассматриваются как имеющие место, т.е. истинные для данной задачи факты. А вопрос или цель решения задачи, выраженная формулой B (заключением), требует определения его истинностного значения. В процессе решения необходимо показать, что истинность заключения B следует из истинности посылок $A_1, A_2, \ldots, A_n$. Так, например, рассмотрим следующие утверждения.

Пример 1.4.

1. Кто может читать, тот грамотный.

$A_1 : (\forall x)[Ч(x) \to Г(x)]$.

2. Дельфины не грамотны.

$A_2 : (\forall x)[Д(x) \to \overline{Г(x)}]$.

3. Некоторые дельфины обладают интеллектом.

$A_3 : (\exists x)[Д(x) \,\&\, И(x)]$.

Из этих утверждений следует доказать, что

4. Некоторые из тех, кто обладает интеллектом, не могут читать.

$B : (\exists x)[И(x) \,\&\, \overline{Ч(x)}]$.

Если даны формулы $A_1, A_2, \ldots, A_n$ и B, то говорят, что формула B является логическим следствием $A_1, A_2, \ldots, A_n$ (или B логически сле-

дует из $A_1, A_2, \ldots, A_n$): $A_1, A_2, \ldots, A_n \models B$ тогда и только тогда, когда для любой интерпретации I, в которой $A_1 \& A_2 \& \ldots \& A_n$ – истинно, B – также истинно.

Логически правильные рассуждения важны для образования новых истинных утверждений из истинных посылок .

Задача установления логического следования B из посылок $A_1, A_2, \ldots, A_n$ может быть сведена к задачам установления общезначимости или противоречивости некоторых формул, что и утверждается в следующих теоремах.

1. Даны формулы $A_1, A_2, \ldots, A_n$ и B. Формула B является логическим следствием формул $A_1, A_2, \ldots, A_n$ тогда и только тогда, когда формула $A_1 \& A_2 \& \ldots \& A_n \to B$ общезначима (тавтология) ($\models$ – знак общезначимости).

$$\models A_1 \& A_2 \& \ldots A_n \to B$$

2. Даны формулы $A_1, A_2, \ldots, A_n$ и B. Формула B является логическим следствием формул $A_1, A_2, \ldots, A_n$ тогда и только тогда, когда формула $A_1 \& A_2 \& \ldots \& A_n \& \overline{B}$ противоречива.

В примере 1.4, чтобы доказать утверждение 4, необходимо доказать либо общезначимость формулы:

$$\{(\forall x)[Ч(x) \to Г(x)] \& (\forall x)[Д(x) \to \overline{Г(x)}] \& (\forall x)[Д(x) \& И(x)]\} \to$$
$$\to (\exists x)[И(x) \& \overline{Ч(x)}],$$

либо противоречивость формулы:

$$\{(\forall x)[Ч(x) \to Г(x)] \& (\forall x)[Д(x) \to \overline{Г(x)}] \& (\forall x)[Д(x) \& И(x)]\} \&$$
$$\& \overline{(\exists x)[И(x) \& \overline{Ч(x)}]}.$$

Доказательство на основе правил вывода ИП с использованием эквивалентных преобразований представляет собой трудоемкую, плохо (или вовсе) не формализуемую процедуру, требующую искусства математика-логика. Впервые формализовать процесс доказательства удалось с использованием принципа резолюции как правила вывода. Суть этого принципа будет представлена позже. Вначале необходимо научиться преобразовывать произвольную логическую формулу к стандартной нормальной форме в виде совокупности дизъюнктов. Такая форма требуется для формализованного доказательства на основе метода резолюций. Преобразование к ней рассматривается в следующем разделе.

1.5. Преобразование логических формул к множеству предложений – дизъюнктов

Постановка задачи

Разнообразие ППФ исчисления высказываний и исчисления предикатов требует, чтобы формулы для их анализа и использования в процедуре доказательства были бы приведены к некоторой нормальной форме. Такой формой является совокупность дизъюнктов (предложений), к которой всегда может быть приведена произвольная ППФ.

Таким образом, стоит задача преобразования произвольной логической формулы к конъюнктивной нормальной форме $D_1 \,\&\, D_2 \,\&\, \ldots \,\&\, D_L$ и записи её в виде множества

$$S = \{D_1, D_2, \ldots, D_L\},$$

где $D = L_1 \cup L_2 \cup \ldots L_k$ – дизъюнкт, L_i – литерал,

$$L_i = \begin{cases} A(x_1, x_2, \ldots, x_k) - \text{атом}, \\ \overline{A(x_1, x_2, \ldots, x_k)} - \text{отрицание атома}. \end{cases}$$

Процедура преобразования

Процесс преобразования рассмотрим на примере ППФ

$$(\forall x)\{P(x) \to \{(\forall y)[P(y) \to P(f(x,y))] \,\&\, \overline{(\forall y)[Q(x,y) \to P(y)]}\}\}.$$

1. Исключение логических связок $\leftrightarrow$ и $\to$. Многократно (пока это возможно) применяется следующее правило: найти самое первое вхождение связки $\leftrightarrow$ или $\to$ и сделать замены:

$$F \leftrightarrow \Phi = (\overline{F} \cup \Phi) \,\&\, (F \cup \overline{\Phi}) \qquad\qquad F \to \Phi = \overline{F} \cup \Phi.$$

Для нашего примера получаем:

$$(\forall x)\{\overline{P(x)} \cup \{(\forall y)[\overline{P(y)} \cup P(f(x,y))] \,\&\, \overline{(\forall y)[\overline{Q(x,y)} \cup P(y)]}\}\}.$$

2. Продвижение знака отрицания $\overline{}$ до атома. Многократно (пока это возможно) делаются замены:

$$\overline{\overline{F}} = F; \qquad \overline{F \cup \Phi} = \overline{F} \,\&\, \overline{\Phi}; \qquad \overline{F \,\&\, \Phi} = \overline{F} \cup \overline{\Phi};$$
$$\overline{\forall x\, F(x)} = \exists x\, \overline{F(x)}; \qquad \overline{\exists x\, F(x)} = \forall x\, \overline{F(x)}$$

Для нашего примера получаем:

$$(\forall x)\{\overline{P(x)} \cup \{(\forall y)[\overline{P(y)} \cup P(f(x,y))] \,\&\, (\exists y)[Q(x,y) \,\&\, \overline{P(y)}]\}\}.$$

3. Разделение переменных. В пределах области действия квантора переменная, связываемая этим квантором, представляет собой немую переменную. Её повсюду можно заменить любой другой (не встречающейся) переменной в пределах области действия квантора, при

этом значение истинности этой ППФ не изменится. Стандартизация переменных в пределах ППФ означает переименование немых переменных с той целью, чтобы каждый квантор имел свою, свойственную только ему, немую переменную. Так, вместо того, чтобы писать $(\forall x)[P(x) \to (\exists x)Q(x)]$, следует писать $(\forall x)[P(x) \to (\exists y)Q(y)]$.

Для нашего примера получаем:

$$(\forall x)\{\overline{P(x)} \cup \{(\forall y)[\overline{P(y)} \cup P(f(x,y))] \& (\exists w)[Q(x,w) \& \overline{P(w)}]\}\}$$

4. Сколемизация – исключение кванторов существования. Рассмотрим ППФ $(\forall y)[(\exists x)\,P(x,y)]$, которую можно прочесть так: "Для всех y существует некоторое x (возможно, зависящее от y) такое, что $P(x,y)$ истинно". Поскольку квантор существования находится в области действия некоторого квантора общности, то мы допускаем возможность того, что тот x, который существует, может зависеть от y. Пусть эта зависимость будет явно определена с помощью некоторой функции $g(x)$, отображающей каждое значение y в то x, которое "существует". Такая функция называется **сколемовской**. Если x, который существует, заменить на такую сколемовскую функцию, то можно исключить квантор существования и написать $(\forall y)\,P(g(y),y)$.

Общее правило исключения квантора существования можно сформулировать следующим образом. Если квантор существования не входит в область действия никакого из кванторов общности, то применяется сколемовская функция без аргументов, т.е. просто константа. Так, $(\exists x)\,P(x)$ становится $P(a)$, где символ константы a используется для ссылки на элемент, который, как известно, существует. Важно, чтобы a был новым символом константы, отличным от ранее использованных в формуле для обозначения констант. Если квантор существования входит в область действия m кванторов общности, то каждое вхождение переменной квантора существования заменяется m-местной функцией $f_j^{\,m}(x_1, x_2, \ldots, x_m)$ зависящей от переменных кванторов общности. Функциональные символы, используемые в сколемовских функциях, должны быть новыми в том смысле, что они не должны встречаться прежде в этой ППФ. Например, можно исключить $(\exists z)$ из выражения

$$[(\forall w)Q(w)] \to (\forall x)\{(\forall y)\{(\exists z)[P(x,y,z) \to (\forall u)R(x,y,u,z)]\}\}$$

и получить

$$[(\forall w)Q(w)] \to (\forall x)\{(\forall y)\{[P(x,y,g(x,y)) \to (\forall u)R(x,y,u,g(x,y))]\}\}.$$

Для исключения всех кванторов существования из некоторой ППФ указанная процедура применяется по очереди к каждой из составляющих её формул. Для рассматриваемого примера имеем:

$$(\forall x)\{\overline{P(x)} \cup \{(\forall y)[\overline{P(y)} \cup P(f(x,y))] \& [Q(x,g(x)) \& \overline{P(g(x))}]\}\}.$$

5. Преобразование в префиксную нормальную форму. На этом этапе не осталось кванторов существования, и каждый квантор общности можно переместить в начало ППФ и считать, что область действия каждого квантора включает всю последующую часть ППФ. Говорят, что результирующая ППФ находится в префиксной нормальной форме. Правильно построенная формула в префиксной форме состоит из цепочки кванторов, называемой **префиксом**, и следующей за ней безкванторной формулы, называемой **матрицей**. Префиксная форма для рассматриваемого примера имеет вид:

$$(\forall x)(\forall y)\{\overline{P(x)} \cup \{[\overline{P(y)} \cup P(f(x,y))] \& [Q(x,g(x)) \& \overline{P(g(x))}]\}\}.$$

6. Приведение матрицы к конъюнктивной нормальной форме (КНФ). Любая матрица может быть записана как конъюнкция конечного множества дизъюнкций литералов:

$M = S_1 \& S_2 \& \ldots \& S_n$ где $S_i = L_1 \cup L_2 \cup \ldots \cup L_n$, L_i – литерал.

Алгоритм приведения матрицы к КНФ может быть описан следующими шагами.

6.1. Начало: дана формула F, составленная из литер с применением связок $\&$ и $\cup$. Предполагается, что в формуле исключены скобки между одинаковыми связками.

6.2. Найти первое слева вхождение двух символов "$\cup$(" или ")$\cup$". Если таких вхождений нет, то закончить : формула F находится в КНФ.

6.3. Пусть первым вхождением указанной пары символов является "$\cup$(". Тогда часть формулы F

$$F_1 = \Phi_1 \cup \Phi_2 \cup \ldots \cup \Phi_r \cup (U_1 \& U_2 \& \ldots \& U_s)$$

заменить на формулу:

$$(\Phi_1 \cup \Phi_2 \cup \ldots \cup \Phi_r \cup U_1) \& (\Phi_1 \cup \Phi_2 \cup \ldots \cup \Phi_r \cup U_2) \& \ldots \&$$
$$\&(\Phi_1 \cup \Phi_2 \cup \ldots \cup \Phi_r \cup U_s).$$

6.4. Пусть первым вхождением является ")$\cup$". Тогда часть формулы F

$$F_2 = (U_1 \& U_2 \& \ldots \& U_s) \cup \Phi_1 \cup \Phi_2 \cup \ldots \cup \Phi_r$$

заменить на формулу

$$(U_1 \cup \Phi_1 \cup \Phi_2 \cup \ldots \cup \Phi_r) \& (U_2 \cup \Phi_1 \cup \Phi_2 \cup \ldots \cup \Phi_r) \& \ldots \&$$
$$\&(U_s \cup \Phi_1 \cup \Phi_2 \cup \ldots \cup \Phi_r).$$

6.5. Перейти к шагу 6.2.

После приведения матрицы рассматриваемого примера к КНФ формула приобретает вид :

$$(\forall x)(\forall y)\{[\overline{P(x)} \cup \overline{P(y)} \cup P(f(x,y))] \& [\overline{P(x)} \cup Q(x,g(x))] \& \\ \& [\overline{P(x)} \cup \overline{P(g(x))}]\}.$$

7. Исключение кванторов общности. Поскольку все переменные в формулах связаны кванторами общности и порядок кванторов общности роли не играет, можно не указывать явно вхождения кванторов. Тогда останется лишь матрица КНФ.

8. Исключение символов & . Можно исключить явное присутствие символов & , заменяя выражения вида $(x_1 \& x_2)$ на множество x_1, x_2 . В результате исключения символов & получим конечное множество ППФ, каждое из которых является дизъюнкцией литералов. Любая ППФ, состоящая только из дизъюнкции литералов, называется **предложением**. Для рассматриваемого примера имеем следующее множество предложений:

$$S = \{\overline{P(x)} \cup \overline{P(y)} \cup P(f(x,y)), \overline{P(x)} \cup Q(x,g(x)), \overline{P(x)} \cup \overline{P(g(x))}\}.$$

9. Переименование переменных. Символы переменных должны быть изменены так, чтобы каждый появлялся не более чем в одном предложении. Этот процесс называют разделением переменных.

Множество S для рассматриваемого примера будет теперь выглядеть так:

$$S = \{\overline{P(x_1)} \cup \overline{P(y)} \cup P(f(x_1,y)), \overline{P(x_2)} \cup Q(x_2,g(x_2)), \overline{P(x_3)} \cup \overline{P(g(x_3))}\}.$$

Множество дизъюнктов $S = \{D_1, D_2, \ldots, D_n\}$ сохраняет свойство невыполнимости (противоречивости), если этим свойством обладала исходная логическая формула F .

1.6. Принцип резолюции

Принцип резолюции в логике высказываний

Одной из наиболее важных задач в логике является задача разработки формализованных процедур доказательства теорем или установления логического следования одной ППФ из других. Все ранние попытки решить эту задачу наталкивались на большие сложности, особенно в исчислении предикатов. И только введение принципа резолюции в логику Дж. Робинсоном в 1964 – 1965 годах позволило показать принципиальную возможность машинного доказательства теорем [2].

При описании условий большинства задач логическими формулами получаем множество A_1, A_2, …, A_n правильно построенных формул,

называемых посылками, исходя из истинности которых, требуется доказать некоторую ППФ B, выражающую цель решения задачи и называемую заключением. В алгоритмах, основанных на резолюции, осуществляется доказательство противоречивости следующей формулы $A_1 \& A_2 \& \ldots \& \overline{B}$, которая описывает известный метод доказательства "от противного".

Вначале эта формула преобразуется во множество предложений, после чего резолюция используется при попытке вывести противоречие, представляемое пустым предложением [] (NIL).

Основная идея принципа резолюции заключается в проверке, содержит ли множество дизъюнктов S пустой (ложный) дизъюнкт []. Если это так, то S не выполнимо. Если S не содержит [], то следующие шаги заключаются в выводе новых дизъюнктов до тех пор, пока не будет получен пустой дизъюнкт [] (что всегда будет иметь место для невыполнимого, т.е. противоречивого S). Таким образом, принцип резолюции рассматривается как правило вывода, с помощью которого порождаются новые дизъюнкты из S, которое формулируется следующим образом.

Если в любых двух дизъюнктах D_1 и D_2 имеется контрарная пара литер (L и $\overline{L}$), то, вычёркивая её, формируем новый дизъюнкт из оставшихся частей дизъюнктов. Этот вновь сформированный дизъюнкт называется резольвентой дизъюнктов D_1 и D_2.

$$L \in D_1;$$
$$\overline{L} \in D_2;$$
$$R = (D_1 - L) \cup (D_2 - \overline{L}) - \text{резольвента}$$

Здесь "–" – знак операции разности множеств.

Пример 1.5.

$D_1 : P \cup \overline{Q} \cup \overline{R}$

$D_2 : Q \cup P$

Резольвента $D_R : P \cup \overline{R}$

Известна следующая **теорема**. Резольвента D_R, полученная из двух дизъюнктов D_1 и D_2, является логическим следствием этих дизъюнктов.

Алгоритм на основе резолюции может быть описан следующим образом.

На вход алгоритма поступает множество дизъюнктов S_0.

1. $S = S_0, k = 0$.

2. Если пустой дизъюнкт принадлежит множеству S ($[] \in S$), то процесс доказательства заканчивается успешно, функция возвращает значение TRUE. Если $[] \notin S$, то переходим к п. 3.

3. В множестве S находим два дизъюнкта D_i, D_j, содержащие контрарную пару: $L \in D_i$, $\overline{L} \in D_j$.

4. Вычисляем резольвенту по формуле: $R = (D_i - L) \cup (D_j - \overline{L})$.

5. Резольвенту добавляем к множеству S: $S = S \cup \{R\}$.

6. $k = k + 1$.

7. Если $k > k_d$, то процесс доказательства заканчивается безуспешно, функция возвращает значение FALSE. В противном случае осуществляем переход к п. 2. Здесь k_d – допустимое число итераций алгоритма.

Пример 1.6. Пусть $S = P \cup Q, \overline{P} \cup Q, P \cup \overline{Q}, \overline{P} \cup \overline{Q}$.

Тогда из

1. $P \cup Q$
2. $\overline{P} \cup Q$
3. $P \cup \overline{Q}$
4. $\overline{P} \cup \overline{Q}$
5. Q, получено из 1, 2
6. $\overline{Q}$, получено из 3, 4
7. NIL, получено из 5, 6.

Пример 1.7. Если команда A выигрывает в футбол, то город A' торжествует, а если выигрывает команда B, то торжествует город B'. Выигрывают или A или B. Однако, если выигрывает A, то город B' не торжествует, а если выигрывает B, то не будет торжествовать город A'. Следовательно, город B' будет торжествовать тогда и только тогда, когда не будет торжествовать город A'.

Условия задачи можно описать следующими ППФ :

$A_1 : A \to A'$, $A_3 : A \cup B$, $A_5 : B \to \overline{A'}$.

$A_2 : B \to B'$, $A_4 : A \to \overline{B'}$,

Требуется доказать $B_1 : B' \leftrightarrow \overline{A'}$.

Методом резолюции докажем противоречивость формулы $A_1 \,\&\, A_2 \,\&\, \ldots \,\&\, A_5 \,\&\, \overline{B_1}$.

Для этого приведём эту формулу к множеству следующих предложений.

1. $\overline{A} \cup A'$	Исходное множество	8. $A' \cup B$	(1, 3)
2. $\overline{B} \cup B'$		9. $\overline{B} \cup A'$	(2, 6)
3. $A \cup B$		10. A'	(8, 9)
4. $\overline{A} \cup \overline{B'}$		11. $A \cup \overline{A'}$	(3, 5)
5. $\overline{B} \cup \overline{A'}$		12. $\overline{A} \cup \overline{A'}$	(4, 7)
6. $A' \cup \overline{B'}$		13. $\overline{A'}$	(11, 12)
7. $\overline{A'} \cup B'$		14. NIL	(10, 13)

Принцип резолюции в логике предикатов первого порядка

Рассмотрим принцип резолюции в логике предикатов первого порядка [1]. Если имеются дизъюнкты типа

$$D_1 : P(x) \cup \overline{R(x)}, \; D_2 : \overline{P(g(x))} \cup Q(y),$$

то резольвента может быть получена только после применения к D_1 подстановки $g(x)$ вместо x. Имеем :

$D_1 : P(g(x)) \cup \overline{R(g(x))}$

$D_2 : \overline{P(g(x))} \cup Q(y)$

Резольвента $D_R : \overline{R(g(x))} \cup Q(y)$

Однако для случая

$$D_1 : P(f(x)) \cup \overline{R(x)}, \; D_2 : \overline{P(g(x))} \cup Q(y)$$

никакая подстановка неприменима.

Подстановкой будем называть конечное множество вида t_1/x_1, t_2/x_2, …, t_n/x_n, где любая x_i – переменная $(1 \le i \le n)$, отличная от t_i.

Пусть $Q = t_1/x_1, t_2/x_2, \ldots, t_n/x_n$ и $L = u_1/y_1, u_2/y_2, \ldots, u_m/y_m$ – две подстановки. Тогда композицией $Q \circ L$ двух подстановок Q и L называется подстановка, состоящая из множества $t_1 \circ L/x_1$, $t_2 \circ L/x_2$, …,

$t_n \circ L/x_n$, $u_1/y_1, u_2/y_2, \ldots, u_m/y_m$, в котором вычеркиваются $t_i \circ L/x_i$ в случае $t_i \circ L = x_i$ и u_j/y_j , если y_j находится среди x_1 , x_2 , …, x_n .

Пример 1.8.

$Q = \{g(x, y)/x, z/y\}$; $L = \{a/x, b/y, x/w, y/z\}$;

$Q \circ L = \{g(a,b)/x, y/y, a/x, b/y, c/w, y/z\} = \{g(a,b)/x, c/w, y/z\}$.

Подстановку Q будем называть **унификатором** для множества выражений $\{W_1, W_2, \ldots, W_k\}$, если $W_1Q = W_2Q = \ldots = W_kQ$. Говорят, что множество выражений $\{W_1, W_2, \ldots, W_k\}$ **унифицируемо**, если для него имеется унификатор. Унификатор σ для множества выражений называется **наиболее общим унификатором** (НОУ) тогда и только тогда, когда для каждого унификатора Q для этого множества выражений найдется подстановка L такая, что $Q = \sigma \circ L$.

Пример 1.9.

$W = \{P(x, a, f(g(a)), P(z, y, f(u))\}$.

Подстановка $\sigma = \{z/x, a/y, g(a)/y, g(a)/u\}$ есть НОУ, а $Q = \{b/x, a/y, b/z, g(a)/u\}$ есть унификатор.

Наиболее общий унификатор может быть найден с помощью следующего алгоритма [1].

Алгоритм унификации (нахождение НОУ)

1. Установить $k = 0$, $W_k = W$ и $\sigma_k = 0$. Перейти к п. 2.

2. Если W_k не является одноэлементным множеством, то перейти к п. 3. В противном случае положить $\sigma = \sigma_k$ и окончить работу.

3. Каждая из литер в W_k рассматривается как цепочка символов и выделяются первые подвыражения литер, не являющиеся одинаковыми у всех элементов W_k , т.е. образуется так называемое множество рассогласований типа x_k, t_k . Если в этом множестве x_k – переменная, а t_k – терм, отличный от x_k , то перейти к п. 4. В противном случае окончить работу: W не унифицируемо.

4. Пусть $\sigma_{k+1} = \sigma_k \circ \{t_k/x_k\}$ и $W_{k+1} = W_k\{t_k/x_k\}$.

5. Установить $k = k + 1$ и перейти к п. 2.

Пример 1.10. Найти НОУ для

$W = \{P(y, g(z), f(x)), P(a, x, f(g(y)))\}$.

1. $\sigma_0 = 0$, $W_0 = W$.

2. Т.к. W_0 не одноэлементно, то переходим к п. 3.

3. Рассогласование $\{y, a\}$, т.е. $\{a/y\}$.

4. $\sigma_1 = \sigma_0 \circ \{a/y\} = \{a/y\}$;

$W_1 = W_0\{a/y\} = \{P(a, g(z), f(x)), P(a, x, f(g(a)))\}$.

5. $k = k + 1$, $k = 1$, переходим к п. 6.

6. W_1 – не одноэлементно. Множество рассогласований $\{g(z), x\}$, т.е. $\{g(z)/x\}$.

7. $\sigma_2 = \sigma_1 \circ \{g(z)/x\} = \{a/y, g(z)/x\}$;

$W_2 = W_1\{g(z)/x\} = \{P(a, g(z), f(g(z))), P(a, g(z), f(g(a)))\}$.

W_2 – не одноэлементно, переходим к п.8.

8. Имеем рассогласование $\{z, a\}$, т.е. $\{a/z\}$.

$\sigma_3 = \sigma_2 \circ \{a/z\} = \{a/y, g(a)/x, a/z\}$;

$W_3 = W_2\{a/z\} = \{P(a, g(a), f(g(a))), P(a, g(a), f(g(a)))\}$ =

$= \{P(a, g(a), f(g, a))\}$.

W_3 – одноэлементно, поэтому $\sigma_3 = \{a/y, g(a)/x, a/z\}$ есть НОУ для W.

Пусть D_1 и D_2 – два дизъюнкта, не имеющие общих переменных. И пусть L_1 и $\overline{L_2}$ – литералы в дизъюнктах D_1 и D_2 соответственно, имеющие НОУ σ. Тогда резольвентой R для D_1 и D_2 является дизъюнкт вида:

$$R = (D_1\sigma - L_1\sigma) \cup (D_2\sigma - \overline{L_2}\sigma).$$

Пример 1.11.

Пусть $D_1 : P(f(g(a)) \cup \overline{R(b)}$; $D_2 : \overline{P(x)} \cup \overline{P(f(y))} \cup Q(y)$.

Тогда $D_2\sigma = D_2\{g(a)/y\} = D_2' = D_2\{\overline{P(f(y))} \cup Q(y)\}\{g(a)/y\} =$ $\overline{P(f(g(a)))} \cup Q(g(a))$ и резольвентой для D_1 и D_2' будет $R =$ $= \overline{R(b)} \cup Q(g(a))$.

Принцип резолюции обладает **свойством полноты**. Множество дизъюнктов S не выполнимо (противоречиво) тогда и только тогда, когда существует вывод из S пустого дизъюнкта [].

Таким образом, при решении задач с использованием логических моделей следует придерживаться следующей технологии.

1.7. Технология решения задач с использованием логических моделей

1. Условия задачи описываются формулами исчисления высказываний или исчисления предикатов. Обозначим эти формулы A_1, A_2, …, A_n. Цель решения задачи, выраженная в виде высказывания или отношения, истинность которого надо доказать, описывается логической формулой B.

2. Суть решения задачи состоит в доказательстве противоречивости формулы

$$F = A_1 \,\&\, A_2 \,\&\, \ldots \,\&\, A_n \,\&\, \overline{B}$$

что следует из теоремы "B является логическим следствием A_1, A_2, …, A_n тогда и только тогда, когда указанная формула противоречива".

3. Формула $F = A_1 \,\&\, A_2 \,\&\, \ldots \,\&\, A_n \,\&\, \overline{B}$ преобразуется к множеству S дизъюнктов в соответствии с описанным в п. 1.5 алгоритмом $F = D_1 \,\&\, D_2 \,\&\, \ldots \,\&\, D_n$, где $D_j = L_1 \cup L_2 \cup \ldots \cup L_k$, отсюда имеем $S = \{D_1, D_2, \ldots, D_n\}$.

Если исходная формула F противоречива, то и множество S также противоречиво. Противоречивость множества S доказывается в соответствии со следующим алгоритмом.

4. Просматривается множество S. Если в нем есть пустой дизъюнкт $[] = \{A \,\&\, \overline{A}\}$, то доказательство заканчивается и делается вывод, что B логически следует из A_1, A_2, …, A_n. Если пустого дизъюнкта нет, то делается попытка получить его путем вывода.

5. В множестве S находятся два дизъюнкта D_i, D_j, содержащие контрарную пару литералов L и $\overline{L}$, затем находится резольвента по правилу:

$$R = (D_i\sigma - L\sigma) \cup (D_j\sigma - \overline{L}\,\sigma),$$

где σ – наиболее общий унификатор-подстановка, делающая идентичными списки аргументов предикатов L и $\overline{L}$. Резольвента это дизъюнкт, являющийся логическим следствием дизъюнктов D_i, D_j.

6. Резольвента R добавляется к множеству S: $S = S \cup \{R\}$.

7. Если число повторений цикла по порождению новых резольвент превышает заданную границу, то процесс решения задачи заканчива-

ется и делается заключение о не выводимости B. Если не превышает, то повторяются действия, начиная с п. 5.

Пример 1.12. Существуют студенты, которые любят всех преподавателей. Ни один из студентов не любит невежд. Следовательно, ни один из преподавателей не является невеждой.

Условия задачи описываются следующими ППФ:

$A_1 : \exists x(C(x) \& \forall y(P(y) \to L(x, y)))$

$A_2 : \forall x(C(x) \to \forall y(H(y) \to \overline{L(x, y)}))$

$B : \forall x(P(y) \to \overline{H(y)})$,

где $C(x)$ – x является студентом; $P(y)$ – y является преподавателем; $L(x, y)$ – x любит y; $H(y)$ – y невежда.

После приведения к множеству дизъюнктов получаем следующее.

1. $C(a)$.
2. $\overline{P(y)} \cup L(a, y)$.
3. $\overline{C(x)} \cup \overline{H(y)} \cup \overline{L(x, y)}$.
4. $P(b)$
5. $H(b)$

} отрицание целевой ППФ

Используя принцип резолюции, получаем следующие резольвенты:

$L(a,b)$	(2,4)	$\sigma = b/y$;
$\overline{H(y)} \cup \overline{L(a, y)}$	(1,3)	$\sigma = a/x$;
$\overline{L(a,b)}$	(5,7)	$\sigma = b/y$;
[]	(6,8).	

1.8. Стратегии управления

Описанная выше процедура доказательства противоречивости множества S является недетерминированной. Способ, в соответствии с которым выбираются два предложения D_i и D_j, содержащие контрарную пару, определяют стратегию управления в методе резолюции. От этой стратегии зависят два свойства алгоритма: полнота (всегда ли будет найдено опровержение, если оно существует) и эффективность. Ниже рассматриваются некоторые из стратегий на следующем примере.

Пример 1.13. Имеются следующие утверждения.

1. Кто может читать, тот грамотный $(\forall x)[Ч(x) \to Г(x)]$.

2. Дельфины не грамотны $(\forall x)[Д(x) \to \overline{Г(x)}]$.

3. Некоторые дельфины обладают интеллектом $(\exists x)[Д(x) \,\&\, И(x)]$.

Необходимо доказать следующее утверждение.

4. Некоторые из тех, кто обладает интеллектом, не могут читать $(\exists x)[И(x) \,\&\, \overline{Ч(x)}]$.

Множество предложений, соответствующих утверждениям с первого по третье, таково:

1) $\overline{Ч(x)} \cup Г(x)$;

2) $\overline{Д(y)} \cup \overline{Г(y)}$;

3 а) $Д(a)$;

3 б) $И(a)$.

Здесь переменные разделены, a – сколемовская константа. Отрицание теоремы, которую надо доказать, преобразованное в форму предложений, имеет вид:

4) $\overline{И(z)} \cup Ч(z)$.

Стратегия поиска в глубину

За начальное состояние S_0 базы данных алгоритма принимается одно из предложений множества S. Затем среди предложений S находится первое попавшееся, составляющее контрарную пару для текущего S_0. Выполняются унификация и резолюция. Получившаяся в результате резольвента принимается за текущее состояние, и весь процесс повторяется заново. Эти действия выполняются до тех пор, пока не будет получен пустой дизъюнкт. Стратегия поиска в глубину [2] иллюстрируется деревом опровержения, приведённым на рис. 1.4.

Стратегия поиска в ширину

В стратегии поиска в ширину (полного перебора) [2] сначала вычисляются все резольвенты первого уровня, затем все резольвенты второго уровня и т.д. Резольвентами первого уровня являются резольвенты, полученные из предложений базового множества; резольвентами i-го уровня являются резольвенты, получаемые из дизъюнктов всех предыдущих (i-1) уровней. Стратегия поиска в ширину является полной, но весьма неэффективной. На рис. 1.5 изображен граф опровержения, порожденный стратегией поиска в ширину.

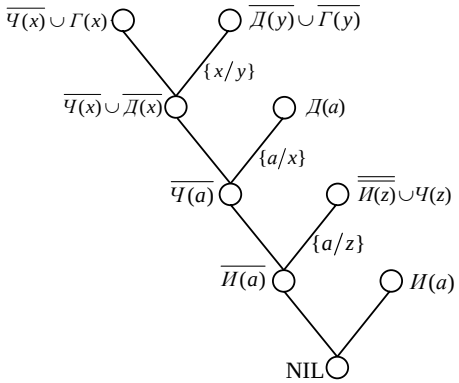


Рис. 1.4. Дерево опровержения стратегии поиска в глубину

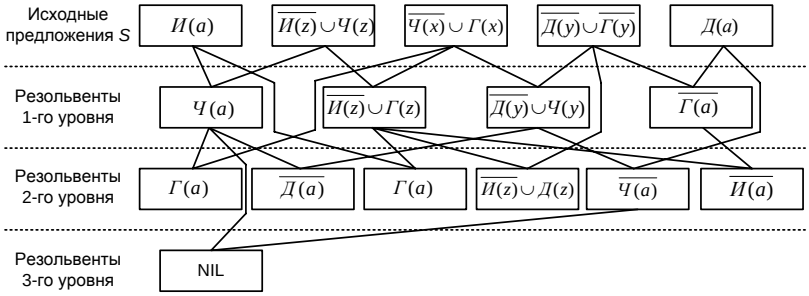


Рис. 1.5. Стратегия поиска в ширину

Стратегия опорного множества

Опровержение с использованием опорного множества [2] состоит в том, что, по крайней мере, один из родителей в каждой резольвенте выбирается из предложений, возникающих при отрицании целевой ППФ или их потомков (опорное множество). Можно показать, что если существует какое-либо опровержение, то существует и некоторое опровержение, основанное на опорном множестве, поэтому полная стратегия может быть основана на опорном множестве. Такая стратегия должна лишь обеспечивать перебор всех возможных опровержений на основе опорного множества. Обычно стратегии, основанные на опорном множестве, оказываются более эффективными по сравнению с неограниченным поиском в ширину. На рис. 1.6 показан граф опровержения, порожденный стратегией с использованием опорного множества.

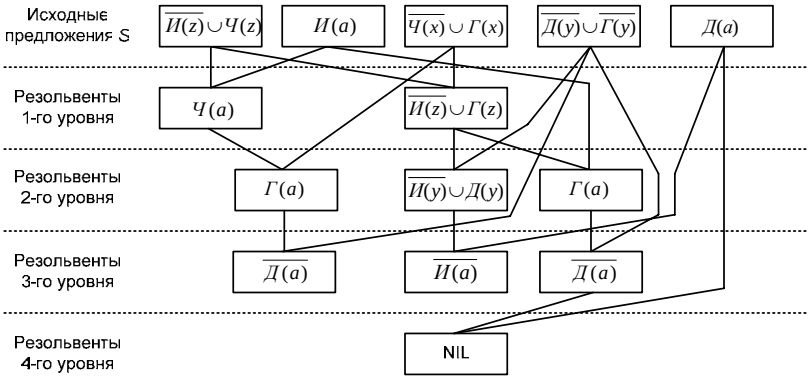


Рис. 1.6. Стратегия опорного множества

Как правило, стратегия опорного множества приводит к более медленному росту множества предложений и тем самым позволяет ослабить обычный комбинаторный взрыв.

Стратегия предпочтения одночленам

Стратегия предпочтения одночленам [2] является такой модификацией стратегии опорного множества, в которой вместо заполнения каждого уровня выбирается однолитеральное предложение (называемое одночленом) в качестве родительской вершины для резолюции. Всякий раз, когда в резолюции используется одночлен, резольвенты содержат меньшее по сравнению с другими родительскими предложениями число литералов. Этот процесс позволяет сконцентрировать поиск в направлении создания пустого предложения и поэтому, как правило, повышает эффективность. Дерево опровержения для стратегии предпочтения одночленам приведено на рис. 1.7.

Линейная по входу стратегия

Линейное по входу опровержение – это опровержение, в котором, по крайней мере, одно родительское предложение в каждой резольвенте принадлежит исходному множеству S предложений. На рис. 1.8 приведен граф опровержения для линейной по входу стратегии.

Имеются случаи, в которых опровержение существует, но не существует опровержения с линейной по входу стратегией, следовательно, линейные по входу стратегии не являются полными. Несмотря на отсутствие свойства полноты, линейные по входу стратегии [2] применяются довольно часто из-за их простоты и эффективности.

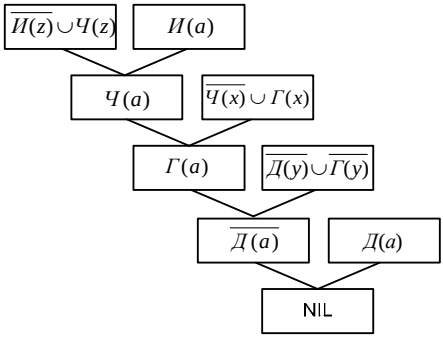


Рис. 1.7. Стратегия предпочтения одночленам

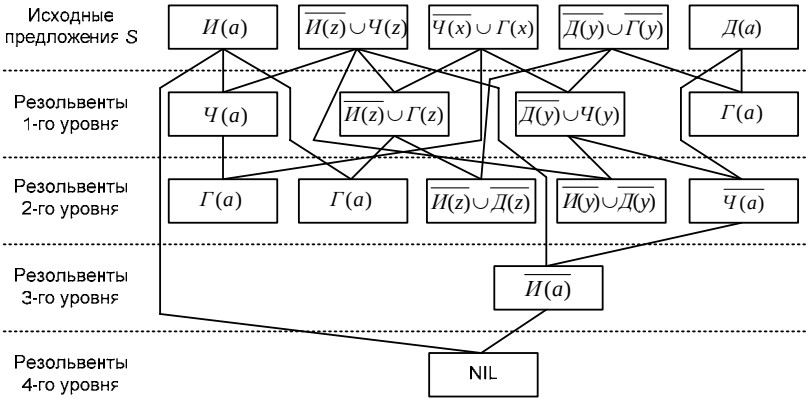


Рис. 1.8. Линейная по входу стратегия

1.9. Хорновские дизъюнкты

Алгоритм, основанный на принципе резолюции, положен в основу работы интерпретатора языка Пролог. В теории логического программирования было показано, что чтобы исключить неоднозначность алгоритма следует для описания предметной области использовать логические формулы, приводимые к Хорновским дизъюнктам [2, 10].

Хорновский дизъюнкт – это дизъюнкт, состоящий не более чем из одного литерала без отрицания, все остальные литералы должны быть с отрицанием. Например:

$$D_1 = L; \quad D_2 \equiv \overline{L}_1 \cup \overline{L}_2 \cup ... \cup \overline{L}_n; \quad D_3 \equiv L_1 \cup \overline{L}_2 \cup ... \cup \overline{L}_n$$

Напомним, что литерал – это либо предикат $P()$, либо предикат с отрицанием $\overline{P()}$.

Покажем, что Хорновские дизъюнкты позволяют избавиться от неоднозначности алгоритма. Рассмотрим произвольный дизъюнкт и его литералы расположим в следующем порядке: сначала литералы без отрицания, затем литералы с отрицанием.

$$D = G_1 \cup G_2 \cup G_3 \cup ... \cup \overline{L}_1 \cup ... \cup \overline{L}_n$$

Перейдем от записи в виде дизъюнкта к эквивалентной имликативной форме:

$$L_1 \,\&\, L_2 \,\&\, ... \,\&\, L_n \to G_1 \cup G_2 \cup ... \cup G_m ,$$

которая читается так "Если истинны $L_1, L_2, \ldots, L_n$, то истинны или G_1, или G_2, или … G_m". Такая фраза может трактоваться однозначно только в том случае, если в правой части импликации будет не более одного литерала G_i без отрицания, а это по определению возможно только для Хорновского дизъюнкта.

Рассмотрим следующие случаи.

1. $m = 1$, $n = 0$, получим формулу G_1: она соответствует факту в языке Пролог, т.е. высказыванию, которое всегда истинно.

2. $m = 0$, $n = 1$, получим формулу $L_1 \to ?$: она соответствует вопросу в языке Пролог – гипотезе, истинность которой выясняется в процессе выполнения программы.

3. $m = 1$, $n > 1$, получим формулу $L_1 \,\&\, L_2 \,\&\, L_3 \,\&\, \ldots \,\&\, L_n \to ?$: она соответствует конъюнкции запросов (предикатов) в языке Пролог, истинность которых определяется в процессе выполнения программы.

4. $m = 0$, $n > 1$, получим формулу $L_1 \,\&\, L_2 \,\&\, L_3 \,\&\, \ldots \,\&\, L_n \to G_1$: она соответствует правилу в языке Пролог, где G_1 – заголовок правила, $L_1 \,\&\, L_2 \,\&\, L_3 \,\&\, \ldots \,\&\, L_n$ – тело правила.

Таким образом, все предикаты тела правила, и все предикаты конъюнкции запросов являются литералами с отрицанием, а все заголовки правил и факты – литералами без отрицания. При доказательстве истинности текущего предиката L_n контрарная ему пара ищется среди заголовков G_m. Если имена и число аргументов L_n и G_m совпадают, начинается процесс унификации аргументов. При успешной унификации процесс доказательства текущего предиката либо заканчивается успешно, если найденное утверждение G_m факт, либо пере-

ходит к доказательству первого предиката L_1 в теле $L_1 \& L_2 \& L_3 \& \& \ldots \& L_n$ правила, если найденное утверждение G_m правило.

Подробно алгоритм работы интерпретатора Пролога рассмотрен в гл. 3.

В табл. 1.2 приведено соответствие между понятиями исчисления предикатов первого порядка и базовыми конструкциями языка Пролог.

Табл. 1.2

№ п.п	Исчисление предикатов первого порядка	Конструкции языка Пролог
1.	Предметная константа a, b, $c \in V$	Константа: число, символическое имя, строка – 34, студент, "Петров С.С."
2.	Предметная переменная $x \in V$, $y \in V$, $z \in V$	Переменная: имя переменной начинается с большой буквы или со знака подчеркивания – Дом, X, _луч
3.	Предикат $P(t_1, t_2, \ldots, t_n)$, где t_k – термы	Предикат, описывающий отношение (связь) в предметной области – *студент("Петров С.С. ", 843)*
4.	Литерал без отрицания – G_k	Факт – предикат, после которого поставлена точка (тело – true).
5.	Литерал с отрицанием L_n	Вопрос – предикат, истинность которого доказывается в процессе выполнения программы, находится в разделе *goal*
6.	Хорновский дизъюнкт $L_1 \& L_2 \& \ldots \& L_n \rightarrow G_1$	Правило <заголовок> :– <тело>

Контрольные вопросы

1. Каково назначение формальной системы?
2. Что такое интерпретация формальной системы?
3. Какие свойства характеризуют ФС?
4. Сколько интерпретаций имеет ППФ ИВ, содержащая n атомов? Что представляет собой каждая интерпретация?
5. Каковы правила образования ППФ в ИВ?
6. Какими свойствами обладает исчисление высказываний? Поясните каждое свойство.

7. Какая процедура обеспечивает свойство разрешимости для ИВ? Что она позволяет определить?
8. Чем отличается исчисление предикатов от исчисления высказываний?
9. Что такое предикат?
10. Каковы правила конструирования термов? Что такое терм?
11. Каковы правила образования ППФ в исчислении предикатов?
12. Какие правила вывода в исчислении предикатов Вы знаете? Поясните их.
13. Что понимается под интерпретацией в ИП?
14. Какие свойства ИП Вы знаете? Поясните их.
15. В чем основная идея принципа резолюции?
16. Что такое подстановка? Зачем она нужна при рассмотрении принципа резолюции в исчислении предикатов?
17. Какими свойствами обладает резольвента?
18. В чем суть стратегии опорного множества?
19. В чем суть стратегии предпочтения одночленам?
20. Что такое Хорновские дизъюнкты и как они связаны с основными конструкциями языка Пролог?

Контрольные задания

Задание: условия задачи описать логическими формулами, решить задачу, используя метод, основанный на принципе резолюции, составить программу построения таблиц истинности для формул задачи и с помощью программы показать справедливость теорем T_1 и T_2.

1. По обвинению в ограблении перед судом предстали A, B и C. Установлено следующее: 1) если A не виновен или B виновен, то C виновен; 2) если A не виновен, то C не виновен. Можно ли установить виновность для каждого из трех подсудимых?

2. Про некое лицо по имени Владимир известна следующая информация. Если Владимир интересуется логикой, то он либо запишется в следующем семестре на занятия по курсу "Логика", либо он ленив. Если Владимир самостоятельно изучил литературу по логике, то он интересуется логикой. Владимир самостоятельно изучал литературу по логике, Владимир не ленив. Вопрос: запишется ли Владимир в следующем семестре на курс "Логика"?

3. На склад, имеющий два помещения для хранения двух видов топлива – угля и кокса, каждого отдельно, поступают грузовики, каждый всякий раз с отдельным видом топлива. К механизму, открывающему шахты, предъявляется требование, чтобы он открыл шахту в по-

мещение для угля, если прибыл грузовик с углем, и шахту для кокса, если грузовик с коксом. Для обеспечения хорошей сортировки топлива было предъявлено дополнительное требование: всякий раз в помещение склада впускается только один грузовик и открывается лишь одна шахта. Спрашивается, имеет ли этот механизм также следующее свойство: если не въехал в помещение склада грузовик с углем, то шахта для угля не откроется, а если не въехал грузовик с коксом, то не откроется шахта для кокса.

4. Если команда A выигрывает в футбол, то город A' торжествует, а если выигрывает команда B, то торжествует город B'. Выигрывает или A, или B. Однако, если выигрывает A, то город B' не торжествует, а если выигрывает B, то не будет торжествовать город A'. Следовательно, город B' будет торжествовать тогда и только тогда, когда не будет торжествовать город A'.

5. Будет пасмурная погода со снегом. Если будет снег, то будет и дождь. Если будет пасмурная погода с ветром, то дождя не будет. Вывод: ветра не будет.

6. На день рождения было решено купить астры или георгины. Было также решено, что купленные цветы должны быть светлыми и красными. В магазине выяснилось, что все светлые астры не красные. Вывод: были куплены георгины.

7. Известно, что посетитель буфета взял или кефир, или молоко, или сок. Если он взял кефир или молоко, то взятый им напиток был холодным. Если же он взял не холодный напиток, то это не сок. Вывод: посетитель взял холодный напиток.

8. В вазе лежит только одно яблоко. Известно, что оно или красное, или зеленое. Если это яблоко красное и большое, то оно сладкое. Если же это яблоко зеленое и не сладкое, то оно не большое. Выяснилось также, что это яблоко сладкое. Вывод: если это яблоко не большое и не красное, то оно зеленое.

9. Либо будет снег, либо дождь. Погода будет пасмурной и ветряной. Если не будет снега, то погода будет не пасмурной. Ветреная, дождливая погода бывает только при пасмурной погоде. Вывод: будет ветреная погода с дождем или снегом.

10. Если Петя не пойдет в кино, то он будет смотреть телевизор или пойдет к своим друзьям. Если же он пойдет к друзьям, то он не пойдет в кино. А если он не пойдет в кино, то он не приготовит уроки. Если же он приготовит уроки, то будет смотреть телевизор. Вывод: если Петя выполнит уроки, но не пойдет в кино, то он не пойдет и к своим друзьям.

11. Надо купить рубашку, которая может быть или белой, или голубой, или розовой. Если будет куплена голубая или розовая рубашка, то она будет шерстяной. Если же будет куплена льняная или шерстяная рубашка, то она будет белой или голубой. Вывод: если будет куплена голубая рубашка, то она будет шерстяной.

12. Существуют студенты, которые любят всех преподавателей. Ни один из студентов не любит невежд. Следовательно, ни один из преподавателей не является невеждой.

13. Тони, Майк и Джон являются членами Альпинклуба. Каждый член Альпинклуба, не являющийся горнолыжником, является альпинистом. Альпинисты не любят дождя, и всякий, кто не любит снега, не является горнолыжником. Майк не любит то, что любит Тони, и любит то, что Тони не любит. Тони любит дождь и снег. Имеется ли такой член Альпинклуба, кто является альпинистом, но не является горнолыжником?

14. Преподаватели принимали зачеты у всех студентов, не являющихся отличниками. Некоторые аспиранты и студенты сдавали зачеты только аспирантам. Ни один из аспирантов не был отличником. Следовательно, некоторые преподаватели были аспирантами.

15. Ситуация из мира кубиков описывается следующим множеством ППФ:

на_столе (a)	свободен (e)
на_столе (c)	свободен (d)
на (b,a)	тяжелый (d)
на (d,c)	деревянный (b)
тяжелый (b)	на (e,b)

Общие знания из этого мира кубиков содержат следующие утверждения:

"Каждый большой голубой кубик находится на зеленом кубике",

"Каждый тяжелый деревянный кубик является большим",

"Все кубики со свободной верхней поверхностью являются голубыми",

"Все деревянные кубики являются голубыми ".

Какой кубик находится на зеленом кубике?

16. Все люди – животные. Следовательно, голова человека является головой животного.

17. Для всех x и y, если x является отцом y и y является отцом z, то x является дедушкой z. Каждый индивид имеет отца. Отсюда следует, что каждый индивид имеет дедушку.

Глава 2. ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ С ПОМОЩЬЮ ПРОГРАММЫ НА ПРОЛОГЕ

Пролог – это язык программирования, предназначенный для представления и использования знаний о некоторой предметной области. Под **предметной областью** будем понимать множество рассматриваемых объектов и совокупность знаний о них. Любую взаимосвязь между объектами и (или) их свойствами назовем **отношением**. Так для предметной области "Студенческая группа" характерными объектами будут студент, группа, факультет. Свойства объектов – это фамилия, имя, отчество студента, номер группы, наименование факультета. Связи между конкретным студентом и номером группы, между его оценками и размером стипендии, между номером группы и факультетом, специальностью, курсом обучения будут являться отношениями. Таким образом, ***предметная область*** – это ***объекты и отношения***.

Объекты могут быть объединены в классы, обладающие определенными свойствами. Элементы этих классов являются конкретными объектами предметной области. Для представления конкретного объекта в программе на Прологе используется константа. Константа – это число или ***символическое имя*** объекта. Например: "учебник", 345, 6.89, "Петров".

Имена конкретных объектов, отношений, свойств образуются по определенным правилам, зависящим от версии (реализации) языка Пролог, и называются ***символическими именами***. Ниже рассмотрим правила, характерные для большинства версий Пролога.

Символическое имя (атом) – это неразрывная цепочка букв (в Visual Prologe латинских и русских), цифр и символа подчеркивания, начинающаяся со строчной латинской (русской) буквы.

Для ***именования объектов и их свойств*** могут использоваться ***строки*** – последовательности любых символов, заключенные в двойные кавычки. В дальнейшем для именования конкретных объектов будем чаще всего использовать строки.

Например: *table1, стол, stock_5, r5z, студент, машина, меню, "Закон", "список аргументов", "группа 343"*.

Для описания в программе некоторого объекта, принадлежащего определенному классу, используется **переменная**. ***Переменная*** в программе представляется своим именем. ***Имя переменной*** в Прологе – это цепочка букв или цифр, начинающихся с прописной буквы или символа подчеркивания. Примеры имен переменных: Группа, Возраст, D, Leda, X, Y, _P, Lokon, День2 и т. д.

Чтобы описать ***отношение***, необходимо указать его имя и перечислить через запятую объекты, связываемые этим отношением.

<имя отношения>(<имя объекта 1>,<имя объекта 2>,…,
<имя объекта n>)

Отношение характеризуется именем и числом аргументов. Число аргументов равно числу объектов, связанных этим отношением. Для описания отношений в программе на Прологе используются предикаты. ***Предикат*** – это логическая функция от n аргументов, имеющая только два значения "истина" и "ложь". Синтаксис предиката

<имя предиката>(<аргумент 1>,<аргумент 2>,…,
<аргумент n>)

При описании отношения имя предиката совпадает с именем отношения, а аргументы предиката – это связываемые отношением объекты. Если описываемое предикатом отношение имеет место в предметной области, то предикат принимает значение "истина", если оно несправедливо для данной предметной области, то значение предиката "ложь".

В качестве имени предиката (отношения) в Visual Prolog допустимо использовать только символические имена.

Знания о предметной области выражаются на языке Пролог в виде предложений, называемых ***утверждениями (clauses)***. Каждое утверждение заканчивается точкой и описывает какое-либо отношение, свойство, объект или закономерность. Структура утверждения проста и имеет одну из форм:

<заголовок>. */*факт*/*

или

<заголовок>:- <тело>. */*правило*/*,

где ***заголовок*** является предикатом и полностью характеризует описываемое отношение.

Тело утверждения состоит либо из одного предиката, либо из списка предикатов, разделенных знаками ",", ";", "not", соответствующими логическим операциям "и", "или", "не". Таким образом, тело утверждения является логическим выражением. Каждый входящий в это выражение предикат описывает какое-либо отношение. Знак ":-" соответствует слову "если". Утверждение читается так: "Отношение, стоящее в заголовке будет истинным, если истинно логическое выражение, находящееся в теле утверждения".

Утверждения образуют программу. На языке ***Visual prolog*** для них отводится специальный раздел, называемый ***clauses.***

Прежде чем рассматривать примеры программ отметим следующее. В среде Visual Prolog 7.0 – 7.3 можно создавать приложения двух

типов: в режиме консоли и в режиме графического интерфейса. Большинство примеров, приведенных в данном пособии, выполнены в режиме консоли, и только реализация нескольких задач рассматривается с использованием графического интерфейса. В главе 10 даны правила создания приложений обоих типов.

Рассмотрим более подробно различные виды утверждений.

2.1. Факты

Все утверждения программы на Прологе делятся на ***факты, правила и вопросы***. ***Факты*** отражают текущее состояние предметной области, содержат конкретную информацию и являются истинными предикатами. Факты соответствуют простым безусловным высказываниям.

Рассмотрим пример описания меню в ресторане. Объекты предметной области – это блюда, которые можно съесть в ресторане, а одним из возможных видов отношений является классификация всех блюд на закуски, вторые мясные или рыбные блюда и десерты. Меню представляет собой небольшую ***базу знаний (БЗ)***, которая записывается в виде последовательности фактов следующим образом:

```
 /* МЕНЮ */
domains /*описание типов данных*/
        name = string.
class facts
 /*описание динамической базы данных*/
        закуска:(name).
        мясо:(name).
        рыба:(name).
        десерт:(name).
clauses
/*утверждения (факты и правила) БЗ*/
        /*Определение отношения закуска
        в виде фактов*/
        закуска("артишоки_в_белом_соусе").
        закуска("трюфели_в_шампанском").
        закуска("салат_с_яйцом").
        /*Определение отношения мясо в виде фактов*/
        мясо("говяжье_жаркое").
        мясо("цыпленок_в_липовом_цвете").
        /*Определение отношения рыба в виде фактов*/
        рыба("окунь_во_фритюре").
        рыба("фаршированный_судак").
        /*Определение отношения десерт в виде фактов*/
```

```
десерт("грушевое_мороженое").
десерт("земляника_со_взбитыми_сливками").
десерт("дыня_сюрприз").
```

Эти ***факты*** вводят одновременно ***объекты*** и их классификацию (***отношения)***. Например, факт *закуска("салат_с_яйцом")* показывает, что салат с яйцом является закуской.

Синтаксически правильно записанный факт имеет следующую структуру предиката (см. рис. 2.1), после которого поставлена точка.

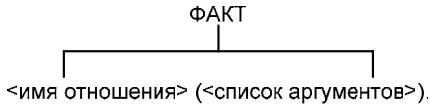


Рис. 2.1. Структура предиката ***факт***

На рис. 2.1 в списке аргументов перечисляются имена объектов (не более 255), связанных данным отношением. Аргументы в списке отделяются друг от друга запятыми, в некоторых случаях могут отсутствовать. Если аргумент представляет собой имя конкретного объекта (свойства) или число, то он является ***константой*** Пролога.

Определяя с помощью фактов отношения между объектами, необходимо учитывать порядок, в котором перечисляются их имена внутри круглых скобок. Выбрав один раз какой-либо порядок, вы должны везде следовать ему и далее. Например, факт *является_отцом("Петр", "Иван")* означает, что Петр является отцом Ивану, а факт *является_отцом("Иван", "Петр")* говорит уже совсем о другом, а именно, что Иван является отцом Петра. Одно и то же утверждение, записанное в виде факта, может по-разному интерпретироваться. Только автор программы определяет истинную интерпретацию имен объектов и порядок следования аргументов, и им он должен следовать в процессе написания всей программы, отражая в комментариях смысл записанных им высказываний. ***Комментарий*** – это текст, заключенный между символами /* ... */ или строка, начинающаяся со знака %, например:

```
/* Андрею нравится Ольга */
нравится("Андрей", "Ольга").
/* Спица является частью колеса */
часть_объекта("спица", "колесо").
```

Язык Visual prolog позволяет передать смысл аргументов предикатов с помощью специальных разделов описаний ***domains, predicates, facts, database***, которые будут рассмотрены далее в пункте "Структура программы на языке Visual prolog".

2.2. Вопросы или целевые утверждения

Чтобы выполнить простейшую программу на Прологе, откроем файл с программой "Меню" в среде Visual Prolog 7 (как создать консольное приложение описано в главе 10).

Программа начинает выполняться, если в нее ввести те вопросы, ответы на которые хочет получить пользователь. Для этого предназначен раздел ***goal*** (Цель). В нем записываются необходимые **вопросы** – третий тип утверждений в программе на Прологе.

Введите такие вопросы: "Есть ли в ресторане рыбное блюдо "окунь во фритюре"?". В среде VIP 5.2 это будет выглядеть так:

```
goal
    рыба("окунь_во_фритюре").
```

В диалоговом окне получаем ответ

```
    yes
```

или "Является ли мясным блюдом блюдо "артишоки в белом соусе"?"

```
goal
    мясо("артишоки_в_белом_соусе").
no
```

Чтобы задать вопрос в программах на Visual Prolog 7 необходимо в файле *.pro (в файле реализации main.pro) изменить предикат ***run().***

```
clauses
    run():-   закуска("артишоки_в_белом_соусе"),
              write("yes"), nl, !.
    run():-write("no").
    end implement main
goal
    mainExe::run(main::run).
```

Вопросы будут меняться при изменении тела первого утверждения предиката *run()*.

При получении ***вопроса (целевого утверждения)*** программа начинает выполняться. Суть выполнения состоит в поиске в базе знаний заголовка утверждения, соответствующего вопросу. Если такое утверждение (факт) есть, то на экране появляется ответ "yes" (да). Если нет, как в случае вопроса *закуска("салат_из_помидор")*, то на экране появляется отрицательный ответ "no"(нет).

Вопрос, или целевое утверждение, с помощью предиката описывает отношение (гипотезу), истинность которого для данной предметной области неизвестна и должна быть определена в процессе выполнения

программы, в процессе согласования вопроса с утверждениями базы знаний. Синтаксис вопроса совпадает с синтаксисом предиката.

2.3. Переменные

Продолжим работу с "Меню". Предположим, что мы хотим получить информацию обо всех закусках. В этом случае мы, естественно, задали бы вопрос: "Какие блюда являются закусками?" или более формализовано: "Существуют ли те блюда (объекты) X, которые являются закусками?". Здесь имя X обозначает не какой-то конкретный объект, а любой, принадлежащий некоторому множеству (может быть пустому) объектов, обладающих свойством быть закуской и которые нужно найти в меню. В этом случае говорят, что X есть ***переменная***. Имя переменной начинается с большой буквы или со знака подчеркивания.

На Прологе вопрос о закусках записывается так:

```
goal
    закуска(X).
```

Будет получен ответ:

```
X=артишоки_в_белом_соусе
X=трюфели_в_шампанском
X=салат_с_яйцом
3 Solutions
```

Все эти ответы содержат множество тех значений переменной X, при которых утверждение *закуска(X)* истинно.

На Visual Prolog 7 изменяем предикат *run()* следующим образом:

```
clauses
    run():-   nl, закуска(Name),
              write(Name), nl, fail.
    run().
```

2.4. Правила

С помощью ***отношений***, которые составляют начальную базу знаний, можно конструировать более сложные и более общие ***отношения***. Например, с помощью отношений *мясо* и *рыба*, выражающих то, что их аргумент является вторым мясным или рыбным блюдом, можно определить отношение *блюдо*: "Блюдо – это второе мясное или рыбное блюдо", что записывается на Прологе в виде двух правил следующим образом:

```
class predicates /* Описание предиката блюдо */
        блюдо:(name) nondeterm anyflow.
clauses
/*Правила, задающие определение понятия блюдо*/
        блюдо(Y):-мясо(Y).
        блюдо(Y):-рыба(Y).
```

и читается так: "Y является блюдом, если Y – второе мясное блюдо, или Y является блюдом, если Y – второе рыбное блюдо". Последовательность двух правил означает их ***дизъюнкцию*** (операцию "или": первое правило или второе). ***Область действия переменной*** ограничена правилом, в котором она определена. Поэтому переменная из первого правила никак не связана с переменной Y из второго. Вопрос "Что является блюдом?", выраженный в виде:

```
clauses
        run():-   nl, блюдо(Name),
                  write(Name),nl,fail.
        run().
```

вызовет следующие ответы:

```
говяжье_жаркое
цыпленок_в_липовом_цвете
окунь_во_фритюре
фаршированный_судак
4 Solutions
```

Можно построить и более сложные правила. Займемся теперь составлением обеда, в который входят закуска, второе блюдо (мясное или рыбное) и десерт. Обед является, следовательно, тройкой (X, Y, Z), где X – закуска, Y – блюдо, Z – десерт. В Прологе это выражается очень естественно в виде следующего правила:

```
class predicates /* Описание предиката обед */
        обед:(name,name, name) nondeterm anyflow.
clauses
/* Определение отношения "обед" */
        обед(X,Y,Z):-закуска(X), блюдо(Y), десерт(Z).
```

Оно читается так: "X, Y, Z удовлетворяют отношению *обед*, если X удовлетворяет отношению *закуска*, Y удовлетворяет отношению *блюдо* и Z удовлетворяет отношению *десерт*". Формально говоря, мы определили новое отношение как ***конъюнкцию*** (операцию ***И***) трех других отношений, как ***конъюнкцию предикатов***.

На вопрос "Что является обедом?":

```
clauses
        run():-   nl, обед(X,Y,Z),
                  write(X," ",Y," ",Z), nl, fail.
        run().
```

вы получите ответ:

```
артишоки_в_белом_соусе говяжье_жаркое грушевое_мороженое
артишоки_в_белом_соусе говяжье_жаркое
земляника_со_взбитыми_сливками
 .....
36 Solutions
```

(т.е. выдаст список всех 36 возможных комбинаций из трех блюд. Попробуйте это выполнить).

Правила описывают зависимость некоторого отношения от группы других отношений (зависимость предиката от группы других предикатов), называемых условиями (или целевыми утверждениями). Правило соответствует условному высказыванию (импликации) и имеет следующую синтаксическую структуру (см. рис. 2.2).

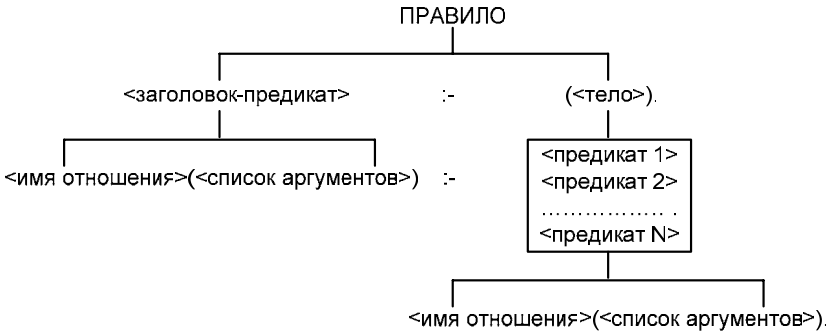


Рис. 2.2. Синтаксическая структура правила

Правило состоит из заголовка и тела правила. Заголовок и тело соединяются с помощью символа ":-", соответствующего в русском языке слову "если". Правила также заканчиваются точкой. Заголовок правила описывает отношение, для определения которого предназначено правило. Тело правила в большинстве случаев представляет собой **конъюнкцию** предикатов (**целевых утверждений**), которые должны быть последовательно согласованы с базой знаний для того, чтобы заголовок правила был истинным. Предикаты (цели) в теле правила разделяются запятыми. Правило – это некоторое общее утверждение.

Оно описывает закономерность, свойственную не какому-то конкретному объекту, а целому классу объектов. Поэтому в аргументы правил входят **переменные**. Правило можно рассматривать и как разбиение сложной задачи (цели) на более простые подзадачи (подцели).

Примеры правил:

```
любит(X, "баскетбол"):- любит(X, "бег").
/* X любит баскетбол, если X любит бегать */
можно_купить(X):- есть_в_магазине(X,V), V<3000.
/* можно купить X, если X есть в магазине
и стоит меньше 3000р. */
старше(P1,P2):-     возраст(P1,V1),
                    возраст(P2,V2),
                    V1>V2.
/*субъект P1 старше субъекта P2,если его возраст больше
возраста P2*/
```

2.5. Конъюнкция целевых утверждений

Вернемся к программе "Меню". Уточним вопрос, сохранив то же множество отношений: нас интересуют обеды с главным блюдом из рыбы. Этот вопрос имеет вид:
на языке Visual Prolog 5.2

```
goal
    обед(X,Y,Z), рыба(Y).
```

в консольном режиме среды Visual Prolog 7

```
clauses
    run():-   nl, обед(X,Y,Z) ,рыба(Y),
              write(X," ",Y," ",Z), nl, fail.
    run().
```

и на естественном языке формулируется следующим образом: "Какие блюда X, Y, Z составляют обед и в этом обеде второе блюдо – рыба?". Вопрос состоит из конъюнкции двух целевых утверждений (предикатов), истинность которых должна быть подтверждена в процессе выполнения. Сначала будут найдены такие значения переменных X, Y, Z, которые составляют обед, т.е. при которых первый предикат станет истинным. Такими первыми значениями являются:

```
X=артишоки_в_белом_соусе,Y=говяжье_жаркое,
Z=грушевое_мороженое
```

После сопоставления с базой знаний первой цели программа перейдет ко второму отношению *рыба(Y)* со значением, которое, приняла

переменная *Y=говяжье_жаркое*. При этом будет проверяться предикат *рыба("говяжье_жаркое")*. Поскольку база знаний не содержит такого утверждения, то предложенный выбор значений не удовлетворяет запросу – выбор неудачен, и нужно анализировать следующие варианты обедов.

Выполните программу и посмотрите, какие ответы будут получены на этот вопрос. Вы убедитесь, что на этот раз получено всего 18 решений, в каждом из которых второе блюдо рыбное.

2.6. Пополнение базы знаний

Пополним базу знаний новым классом отношений, введя значение калорийности для каждого блюда.

```
domains
        kol_vo = integer

class facts
        калории:(name, kol_vo).
clauses
        калории("артишоки_в_белом_соусе", 150).
        калории("трюфели_в_шампанском", 212).
        калории("салат_с яйцом", 202).
        калории("говяжье_жаркое", 532).
        калории("цыпленок_в_липовом_цвете", 400).
        калории("окунь_во_фритюре", 270).
        калории("фаршированный_судак", 254).
        калории("грушевое_мороженое", 223).
        калории("земляника_со_взбитыми_сливками", 289).
        калории("дыня_сюрприз", 122).
```

Утверждение *калории("салат_с_яйцом",202)* означает, что одна порция салата содержит 202 калории.

Тогда, чтобы узнать калорийность всех закусок, зададим вопрос:
на языке Visual Prolog 5.2

```
goal
        закуска(X),калории(X,Y).
```

в консольном режиме среды Visual Prolog 7

```
clauses
        run():-   nl, закуска(X), калории(X,Y),
                  write(X," ",Y), nl, fail.
        run().
```

Для тех, кто беспокоится о своем здоровье или о стройности своей фигуры, важно определить такой объект, как сбалансированный обед, т.е. обед, калорийность которого не превышает, например, 800 калорий:

```
class predicates
        значение:(name,name,name,kol_vo)nondeterm anyflow.
        сбалансированный_обед:(name, name,
                                name) nondeterm anyflow.
clauses
/* Определение отношения "калорийность обеда" */
        значение(X,Y,Z,V):-     калории(X,E),
                                калории(Y,P),
                                калории(Z,D),
                                V = E+P+D.
/*Определение отношения "сбалансированный обед"*/
сбалансированный_обед(X,Y,Z):-      обед(X,Y,Z),
                                    значение(X,Y,Z,V),
                                    V<800.
```

Здесь "="– встроенный предикат, при выполнении которого вычисляется значение арифметического выражения, стоящего справа от "=", и его значение присваивается переменной, стоящей слева; "+"– операция сложения; "<"– операция сравнения.

Посмотрите, какие ответы будут получены в ответ на запрос:
на языке Visual Prolog 5.2

```
goal
        сбалансированный обед(X,Y,Z).
```

в консольном режиме среды Visual Prolog 7

```
clauses
        run():-   nl, сбалансированный_обед(X,Y,Z),
                  write(X," ",Y," ",Z), nl, fail.
        run().
```

2.7. Структура программы на языке Visual Prolog

Рассмотрим еще один пример программы на языке Visual Prolog 7 (файл реализации программы "Студенты" – в среде Visual Prolog 7.3 main.pro).

```
implement main
        open core, console
constants
        className = "main".
        classVersion = "".
```

```
clauses
        classInfo(className, classVersion).
/* программа "Студенты" */
domains
        student = string.
        facultet = string.
        gruppa = integer.
class facts
        студент:(student, gruppa).
        наим_фак:(integer, facultet).
        группа:(gruppa).
clauses
/* определение отношения "группа" с помощью фактов*/
        группа(943).
        группа(843).
        группа(232).
/* определение отношения "студент" с помощью фактов*/
        студент("Орлова Л.И.", 943).
        студент("Семенова М.П.", 232).
        студент("Цуканова В.В.", 943).
/* определение отношения "наименование факультета"
с помощью фактов */
        наим_фак(1,"РТФ").
        наим_фак(2,"ФЭ").
        наим_фак(3,"ФАИТУ").
        наим_фак(4,"ФВТ").
        наим_фак(7,"ИЭФ").
class predicates
        факультет:(gruppa, facultet)nondeterm anyflow.
        вторая_цифра:(gruppa,integer)nondeterm anyflow.
clauses
/* определение отношения "факультет" с помощью правила*/
        факультет(NG,F):-       вторая_цифра(NG,G2),
                                наим_фак(G2,F).
/* определение отношения "вторая цифра" с помощью правила */
        вторая_цифра(NG,G2):-   G1 = NG mod 100,
                                G2 = G1 div 10.
clauses
        run():-     группа(Gruppa), студент(Student, Gruppa),
                    факультет(Gruppa, Facultet),
                    write(Student, " ", Gruppa," ",Facultet), nl,
                    fail.
        run().
end implement main
goal
        mainExe::run(main::run).
```

Программа на языке Visual Prolog имеет структуру, состоящую из следующих разделов:

```
domains                                    /*описание типов данных*/
class facts        /*описание фактов динамической базы данных,
                                    принадлежащих данному классу*/
class predicates                  /*описание предикатов класса*/
clauses                        /*утверждения: факты и правила */
goal                                  /* целевое утверждение */
```

В разделе ***domains*** программист описывает ***множества (типы) объектов,*** задавая их ***символические имена***, и с помощью стандартных типов данных определяя их свойства. Например, множества объектов типа студент, факультет, группа задаются следующим образом:

```
domains
    student = string.
    facultet = string.
    gruppa = integer.
```

Таким образом, в этом разделе программист создает свои типы (области – *domains*) данных.

В разделах ***predicates***, ***class predicates***, ***class facts*** описываются образцы предикатов и фактов, а именно, задаются имена предикатов, число их аргументов, смысл аргументов с помощью имен, веденных в разделе ***domains*** типов данных, режим детерминизма и шаблон потока данных. Например:

```
class predicates
    факультет:(gruppa, facultet)nondeterm anyflow.
    вторая_цифра:(gruppa, integer)nondeterm anyflow.
```

Такое описание показывает, что в предикате *факультет* первым аргументом является номер группы, а вторым – наименование факультета. Ключевые слова ***determ*** и ***nondeterm*** в описании предиката означают, что предикат может быть согласован с базой знаний соответственно один раз или много раз. Ключевое слово ***class*** показывает, что описываемый предикат принадлежит классу и доступен только в нем**.**

Различие ***predicates*** и ***facts*** в том, что в ***facts*** описываются факты динамической базы данных, которые можно добавлять, удалять и изменять в процессе выполнения программы. Раздел ***predicates*** содержит описание предикатов, однако соответствующие этим описаниям утверждения нельзя динамически изменять.

Раздел ***clauses*** содержит собственно программу, т.е. все факты и правила, описывающие предметную область и составляющие базу знаний.

В программе могут присутствовать несколько утверждений, описывающих одно и то же отношение. Все они являются возможными различными вариантами описания этого отношения. Таким образом, совокупность утверждений с одинаковыми заголовками, т.е. с одинаковым именем и тем же числом аргументов называется ***определением отношения.*** Утверждения одного определения отношения должны быть сгруппированы вместе. Все утверждения о предметной области (программа), находящиеся в области действия программы Пролога, называются ***базой знаний***. **Структура раздела (*clauses*) утверждений программы на языке Пролог** приведена на рис. 2.3.

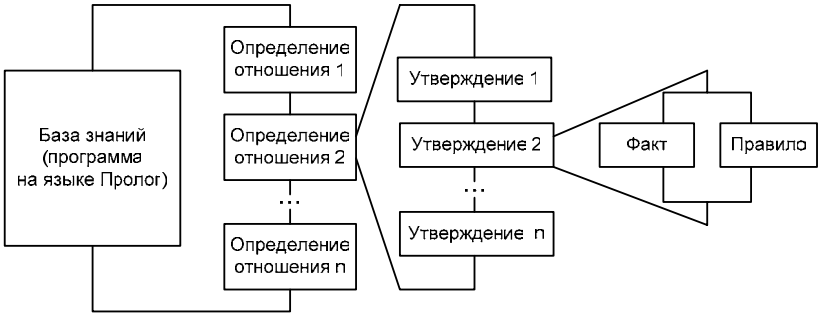


Рис. 2.3. Структура раздела утверждений программы

Раздел ***goal*** содержит предикат *mainExe::run(main::run)*, запускающий программу на выполнение. Чтобы задавать различные вопросы, необходимо изменять тело предиката *run()*, описанного в файле реализации *implement main*.

Множества(типы) объектов предметной области, введенные программистом, описываются с помощью следующих стандартных типов данных:

symbol – символическое имя (person,y,a,mas);

string – строка – любая последовательность символов в двойных кавычках ("Катрин", "Домодедово");

char – отдельный символ, заключенный в апострофы (‘R’, ‘*’);

integer – целое число в диапазоне от -32768 до 32767;

byte – целое число в диапазоне от 0 до 255;

word – целое число в диапазоне от 0 до 65535;

real – любое число, может быть представлено в экспоненциальной форме.

Данные типа *symbol* в отличие от данных типа *string* запоминаются в таблице символов. Таблица символов размещается в оперативной памяти, поэтому ее использование обеспечивает наиболее быстрый поиск. Однако для построения таблицы символов требуется дополнительное время.

2.8. Реляционный язык Пролог

Логические программы можно рассматривать как мощное расширение модели реляционной базы данных (БД). Факты программы образуют отношения БД. Правила реализуют запросы. Основные операции реляционной алгебры легко выражаются в логическом программировании. Пять основных операций определяют реляционную алгебру: объединение, симметрическая разность, декартово произведение, проекция и выборка. Покажем, как каждая из них выражается в логической программе.

Операция ***объединения*** строит одно ***n***-арное отношение из двух ***n***-арных отношений ***r*** и ***s***. Новое отношение, которое обозначим ***r_union_s***, является объединением ***r*** и ***s***. Это отношение задается логической программой, состоящей из двух правил:

```
r_union_s(X1,X2,…,Xn):- r(X1,X2,…,Xn).
r_union_s(X1,X2,…,Xn):- s(X1,X2,…,Xn).
```

Пример:

```
блюдо(Y):-мясо(Y).
блюдо(Y):-рыба(Y).
```

Из примера следует, что объединение отношений можно использовать для обобщения знаний о предметной области.

Определение ***симметрической разности*** использует понятие отрицания:

$$A \oplus B = (A \cap \overline{B}) \cup (\overline{A} \cap B).$$

```
r_diff_s(X1,X2,…,Xn) :-     r(X1,X2,…,Xn),
                            not s(X1, X 2,…, X n).
r_diff_s(X1,X2,…,Xn):-      s(X1,X2,…,Xn),
                            not r(X1,X2,…,Xn).
```

Пример:

```
футболист_либо_волейболист(X) :-   футболист(X),
                                   not волейболист(X).
футболист_либо_волейболист(X) :-   волейболист(X),
```

```
                                        not футболист (X).
```

Декартово произведение r_x_s может быть определено одним правилом:

```
r_x_s(X1, X2,…, Xm, Y1, Y2,…,Yn):-   r(X1, X2, …, Xm),
                                      s(Y1,Y2,…,Yn).
```

Пример:

```
/* Определение отношения "обед" */
обед(X,Y,Z):- закуска(X), блюдо(Y), десерт(Z).
```

Проекция состоит в построении отношения, использующего лишь некоторые аргументы исходного отношения:

```
r13(X1, X3):- r(X1, X2, X3).
```

Пример:

```
/*Телефон аптеки */
аптека(6, "Циолковского, 7", "44-67-57").
телефон_аптеки(NomApt, TelApt):- аптека(NomApt, _, TelApt).
```

Чтобы получить ***выборку***, достаточно в теле правила записать условия отбора:

```
rv(X1, X3):- r(X1, X2, X3), X2>X1.
```

Пример:

```
аптека(6, "Циолковского, 7", "44-67-57").
телефон_аптеки(NomApt, TelApt):-   аптека(NomApt, _, TelApt),
                                   NomApt < 8 .

/*Определение отношения сбалансированный обед*/
сбалансированный_обед(X,Y,Z):-     обед(X,Y,Z),
                                   значение(X,Y,Z,V),
                                   V<800.
```

Операция ***соединения*** двух или нескольких отношений определяется следующим образом:

```
/* определение отношения "факультет" с помощью правила */
факультет(NG,F):-    вторая_цифра(NG,G2),
                     наим_фак(G2,F).
```

Здесь два отношения связаны общей переменной *G2*.

Теоретический материал этого пункта поможет Вам выполнить первое контрольное задание, в котором Вы должны с помощью фактов создать базу данных и с помощью правил определить запросы к ней.

Заключение

В заключение выделим основные понятия второй главы. Итак, ***программа на Прологе*** представляет собой базу знаний о некоторой ***предметной области*** и имеет структуру, представленную на рис. 2.3. Программа состоит из ***утверждений (фактов и правил) и их описаний.*** Совокупность утверждений, имеющих одинаковые имена отношений с одним и тем же числом аргументов (т.е. одинаковые заголовки), называется ***определением отношения***. Утверждения с одинаковыми именами, но с разным числом аргументов образуют разные уникальные определения в программе.

Элементарными составляющими утверждений являются ***предикаты*** со своими списками аргументов, описывающие отношения предметной области. Предикаты представляют собой логические функции, приобретающие значение (истина или ложь) в процессе выполнения программы. Их структура имеет вид:

<имя отношения>(<список аргументов>).

Элементами списка аргументов могут быть только термы, отделяющиеся друг от друга запятой.

Терм – это синтаксически правильная конструкция Пролога. Различают следующие типы термов: константы, переменные, списки, строки, структуры. Пока вы познакомились с константами и переменными. Все остальные типы будут рассматриваться в следующих главах. Программа начинает выполняться после ввода ***вопроса*** или ***целевого утверждения***, которое записывается в форме предиката в разделе

```
goal
<имя отношения>(<список аргументов>).
```

и формулирует цель решаемой задачи. В процессе выполнения устанавливается истинность или ложность поставленной цели.

Программирование на языке Пролог включает следующие этапы:

- объявление фактов, задающих отношения между конкретными объектами или объектами и их свойствами;
- определение правил о зависимости одних отношений или объектов от других;
- формулировка цели решаемой задачи в виде вопроса (гипотезы, целевого утверждения).

Чтобы программу, написанную на базовом Прологе, адаптировать к языку Visual Prolog, необходимо выполнить следующее:

- утверждения программы записать в разделе ***clauses*** в соответствии с правилами: имена предикатов должны быть символическими именами; имена объектов предметной области – строками;

- затем в разделе ***domains*** создать и описать свои типы данных, используя названия типов объектов;
- в разделе ***predicates*** или ***class predicates*** описать предикаты программы, указав их имена и типы объектов, являющиеся аргументами предикатов;
- в разделе ***facts*** или ***class facts*** описать только те факты, которые могут в процессе выполнения программы изменяться, удаляться и добавляться;
- при описании фактов и предикатов после имени отношения необходимо ставить двоеточие;
- в тело предиката *run()* вставить вопрос пользователя.

Пример выполнения контрольного задания

Задание: описать предметную область "Расписание занятий" с помощью программы на языке Пролог.

Отношения – факты:

<группа>(<номер>, <факультет>, <число студентов>, <староста>).
<дисциплина>(<шифр>, <название>, <количество часов>).
<аудитория>(<номер>, <тип>, <число мест>).
<преподаватель>(<ФИО>, <кафедра>, <телефон>).
<расписание_занятий>(<номер_группы>, <шифр_дисциплины>,
<ФИО>, <номер_аудитории>,
<день_недели>, <время>,
<числитель/знаменатель>).

Отношения – правила: какие аудитории заняты в данное время, какие аудитории свободны в данное время, расписание занятий в заданное время, где находится преподаватель по расписанию, расписание занятий для заданной группы и т.д.

```
/* Пример выполнения задания */
implement main
    open core,console, Boolean
constants
    className = "main".
    classVersion = "".
clauses
    classInfo(className, classVersion).
/* Типы данных */
domains
    fio = string.       fucultet = string.
    discip= string.     tip = string.          kaf = string.
    tel = string.       denj = string.         c_z = string.
    rez = boolean.      ngruppa = integer.     kol_vo = integer.
```

```
        shifr = integer.     nau = integer.              vremja = integer.
/* Факты */
class facts
        группа:(ngruppa, fucultet, kol_vo,fio).
        студент:(fio, ngruppa).
        дисциплина:(shifr, discip, kol_vo).
        аудитория:(nau, tip, kol_vo).
        преподаватель:(fio, kaf, tel).
        расписание_занятий:(ngruppa, shifr,fio, nau, denj, vremja, c_z).
/* Определение отношений с помощью фактов */
clauses
        /* определение отношения <группа>(<номер>, <факультет>,
                                    <число студентов>, <староста>)*/
        группа(243,"ФВТ",23,"Карасев").
        группа(244,"ФВТ",25,"Карев").
        /*определение отношения <студент>(<ФИО>, <группа>)*/
        студент("Иванов",243).
        студент("Петров",243).
        студент("Ивагин",243).
        студент("Лунева",243).
        студент("Волков",244).
        студент("Мосин",244).
        /* определение отношения <дисциплина>(<шифр>,
                                    <название>, <количество часов>) */
        дисциплина(1,"Математика",34).
        дисциплина(2,"Физика",48).
        дисциплина(3,"Информатика",56).
        дисциплина(4,"Базы данных",48).
        /* определение отношения <аудитория>(<номер>, <тип>,
                                                <число мест>) */
        аудитория(132,"ауд",30).
        аудитория(206,"лаб",100).
        аудитория(403,"ауд",30).
        аудитория(411,"ауд",10).
        аудитория(324,"ауд",200).
        /* определение отношения <преподаватель>(<ФИО>,
                                            <кафедра>, <телефон>) */
        преподаватель("Шевяков","ВПМ","723723").
        преподаватель("Новичков","ВПМ","323723").
        преподаватель("Швечкова","ВПМ","764589").
        /* определение отношения
        <расписание занятий> (<номер группы>, <шифр дисциплины>,
                        <ФИО преподавателя>,<номер аудитории>,
                        <день недели>, <время>,
                        <числитель/знаменатель>) */
        расписание_занятий(243,1,"Новиков",324, "понедельник",8,"ч").
```

```
        расписание_занятий(243,2,"Шерстов",132,
                                                "понедельник",10,"ч").
        расписание_занятий(243,3,"Новичков",132,
                                                "понедельник",12,"ч").
        расписание_занятий(243,4,"Шевяков",206, "вторник",8,"ч").
        расписание_занятий(243,5,"Парфилова",411,
                                                "вторник",10,"ч").
        расписание_занятий(243,4,"Шевяков",403, "среда",12,"з").
        расписание_занятий(243,5,"Парфилова",206, "среда",14,"з").
class predicates
        занята:(denj, vremja, c_z, nau) nondeterm anyflow.
        свободна:(denj, vremja, c_z, nau, rez)nondeterm anyflow.
        расписание_занятий_группы:(ngruppa, discip, fio, nau, denj,
                                        vremja, c_z) nondeterm anyflow.
        где_преподаватель:(fio,denj,vremja,c_z,nau)nondeterm anyflow.
clauses
/* Определение отношений с помощью правил */
        /* определение отношения
        <занята_аудитория>(<день_недели>, <время>,
                        <числитель/знаменатель>, <аудитория>)*/
        занята(D,V,T,NA):- расписание_занятий(_, _, _, NA, D, V, T).
        /* определение отношения
        <свободна_аудитория>(<день_недели>, <время>,
                         <числитель/знаменатель>, <аудитория>)*/
        свободна(D,V,T,NA,R):- занята(D,V,T,NA), R = false,!.
        свободна(D,V,T,NA,R):- R = true.
        /* определение отношения
        <расписание занятий группы>(<номер группы>,
                        <Дисциплина>, <ФИО преподавателя>,
                         <аудитория>, <день недели>, <время>,
                         <числитель/знаменатель>)*/
        расписание_занятий_группы(NG,D,Pr,Aud,De,Vr,Z):-
                        расписание_занятий(NG, Kd, Pr, Aud,De, Vr, Z),
                        дисциплина(Kd,D,_).
        /*определение отношения <где найти преподавателя>*/
        где_преподаватель(Pr,D,V,T,Na):-
                                        расписание_занятий(_,_,Pr,Na,D,V,T).
clauses
        run():-  console::init(),
                 write("Введите номер группы "),
                 Ng = read(), H = readLine(),
                 nl, nl, write("Расписание занятий группы ", Ng), nl, nl
                 расписание_занятий_группы(Ng,D,Pr,Aud,De,Vr,Z),
                 write(D," ", Pr," ", Aud," ", De," ", Vr, " ",Z),nl,fail.
        run():-  console::init(),
                 write("Введите фамилию преподавателя "),
```

```
                Pr = readLine(), nl, nl,
                write("Где находится преподаватель ",Pr),
                nl, nl, где_преподаватель(Pr,D,V,T,Na),
                write(D," ", Pr," ", Na," ", V, " ",T), nl, fail.
        run():- console::init(),
                write("Введите день,время,", "числитель или
                                        знаменатель "), D = readLine(),
                V = read(), H = readLine(),
                T = readLine(),
                nl, nl, write("Свободные аудитории", "в это время"),
                аудитория(Na,_,_), nl, свободна(D,V,T,Na,R),
                R = true, write(Na," ", D," ", V, " ",T),nl,fail.
                run().
end implement main
goal
    mainExe::run(main::run).
```

Контрольные вопросы

1. Что такое предикат и для описания каких элементов предметной области он используется?
2. Что следует понимать под предметной областью?
3. Как описываются конкретные объекты предметной области в программе на Прологе?
4. Как описываются отношения предметной области в программе на Прологе?
5. Объясните, что такое факт?
6. Объясните, что такое правило?
7. Объясните, что такое база знаний?
8. Как задать вопрос, позволяющий извлечь нужную информацию из базы знаний?
9. Какова структура программы на Прологе?
10. Что такое определение отношения?

Контрольные задания

По аналогии с программой "Меню" создайте небольшую базу знаний на языке Пролог для указанной в вашем варианте предметной области (ПО). Ваша программа должна содержать не менее 10 фактов, описывающих указанные в задании отношения, 2 – 3 правил, выражающих типичные запросы пользователей, Вами должно быть сформулировано не менее 3 целевых утверждений, ответы на которые необходимо отразить в отчете. Отчет должен содержать цель работы,

задание, собранные вами сведения о ПО, программу на Прологе, вопросы к базе знаний и полученные на них ответы.

1. Предметная область "Столицы стран разных частей света".

Отношения – факты:

<столица>(<название столицы>, <название государства>,
<число жителей столицы>),
<находится>(<название государства>, <название части света>,
<площадь государства>).

Отношения – правила: в какой части света находится столица, столица государства с площадью, меньшей заданной величины, какие столицы с миллионным числом жителей находятся в данной части света и т.п.

2. Предметная область "Семья".

Отношения – факты:

<родитель>(<имя родителя>,<имя ребенка>),
<мужчина>(<имя мужчины>),
<женщина>(<имя женщины>).

Отношения – правила: X отец Y, X мать Y, Z сестра V, U брат Z, X дед Y и т.д.

3. Предметная область "Мои любимые книги".

Отношения – факты:

<книга>(<имя автора>, <название>, <жанр>, <год издания>,
<издательство>, <число страниц>).

Отношения – правила: роман X, пьеса Y, писатель Z, книги издательства, книги автора, книги определенного года издания и т.д.

4. Предметная область "Студенты группы".

Отношения – факты:

<студент>(<номер зачетной книжки>, <ФИО>, <группа>,
<год рождения>, <адрес места жительства>, <язык>).

Отношения – правила: X старше Y, студенты одной группы NG, студенты, изучающие английский язык и т.д.

5. Предметная область "Студенты группы".

Отношения – факты:

<студент>(<ФИО>, <группа>).
<успеваемость>(<ФИО>, <группа>, <дисциплина>, <семестр>,
<оценка>).

Отношения – правила: студенты – "отличники", студенты – "хорошисты", успеваемость в группе, успеваемость по каждому предмету, изучаемые дисциплины.

6. Предметная область "Аптеки города".

Отношения – факты:

<аптека>(<номер>, <адрес>,<телефон>).
<лекарство>(<шифр>, <наименование>,<группа>).

<аптека_имеет_лекарство>(<номер_аптеки>, <шифр_лекарства>,
<количество>, <цена>, <срок годности>).

Отношения – правила: телефон аптеки, имеющей нужное лекарство; в какой аптеке цена на заданное лекарство меньше определенной величины; у каких лекарств превышен срок годности и т.д.

7. Предметная область "Кинотеатры города".

Отношения – факты:

<кинотеатр>(<код кт>, <название>, <адрес>, <телефон>,
<количество мест>).
<кинофильм>(<код кф>, <название>, <год выпуска>, <режиссер>,
<число серий>).
<показывает>(<код кт>, <код кф>, <дата>, <время>, <выручка>).

Отношения – правила: телефон кинотеатра, показывающего нужный фильм; в каких кинотеатрах идут фильмы заданного режиссера т.д.

8. Предметная область "Подписка на газеты и журналы в почтовом отделении".

Отношения – факты:

<издание>(<шифр>, <название>, <тип>, <цена 1 экз>,
<число экз в год>).
<подписчик>(<ФИО>, <профессия>, <возраст>, <адрес>).
<подписался>(<шифр_издания>, <ФИО>, <дата_начала>,
<длительность>, <стоимость>).

Отношения – правила: кто на какое издание подписан, стоимость издания, что за издание - журнал или газета и т.д.

9. Предметная область "Друзья".

Отношения – факты:

<личность>(<ФИО>, <адрес>, <телефон>, <хобби>, <возраст>,
<средний заработок>).
<дружат>(<ФИО>,<ФИО>).

Отношения – правила: кто с кем дружит, телефон друга, адрес друга, общие интересы у друзей, друзья, у которых можно попросить в долг и т.д.

10. Предметная область "Прием больных в поликлинике".

Отношения – факты:

<Карта пациента>(<Номер>, <ФИО>, <адрес>, <телефон>,
<возраст>, <место работы>).
<Врач>(<код врача>, <ФИО>, <специализация>).
<Прием>(<код_врача>, <номер_карты_пациента>, <дата_приема>,
<диагноз>, <назначения>, <больничный(да/нет)>).

Отношения – правила: каких врачей прошел пациент за указанный период времени, какие диагнозы были поставлены пациенту за все

время наблюдения, каких пациентов принял врач в течение заданного времени, кому были выданы больничные и т.д.

11. Предметная область "Библиотека".

Отношения – факты:

<книга>(<шифр>, <автор>, <название>, <издательство>,
<год издания>, <количество страниц>).
<читатель>(<ФИО>, <адрес>, <телефон>, <возраст>,
<место работы>).
<учетная карточка>(<ФИО>, <шифр книги>, <дата взятия>,
<дата возврата>, <факт возврата(да/нет)>).

Отношения – правила: кто взял заданную книгу, читатели – должники, какие книги читают молодые люди в возрасте от 18 до 25 лет, каких авторов предпочитает данный читатель и т.д.

12. Предметная область "Преподаватели кафедры ВПМ".

Отношения – факты:

<сотрудник>(<код>, <ФИО>, <кафедра>, <должность>, <оклад>,
<степень>, <звание>).
<дисциплина>(<шифр>, <название>, <количество часов>).
<читает>(<код сотрудника>, <шифр дисциплины>,<семестр>,
<число студентов - слушателей>).

Отношения – правила: степень и звание преподавателя, читающего данную дисциплину, телефон преподавателя кафедры, адрес преподавателя кафедры, какие дисциплины читает преподаватель в данном семестре и т.д.

13. Предметная область "Расписание занятий".

Отношения – факты:

<группа>(<номер>, <факультет>, <число студентов>, <староста>).
<дисциплина>(<шифр>, <название>, <количество часов>).
<аудитория>(<номер>, <тип>, <число мест>).
<преподаватель>(<ФИО>, <кафедра>, <телефон>).
<расписание_занятий>(<номер_группы>, <шифр_дисциплины>,
<ФИО>, <номер_аудитории>,
<день_недели>, <время>
<числитель/знаменатель>).

Отношения – правила: какие аудитории заняты в данное время, какие аудитории свободны в данное время, расписание занятий в заданное время, где находится преподаватель по расписанию, расписание занятий для заданной группы и т.д.

14. Предметная область "Мои любимые фильмы".

Отношения – факты:

<фильм>(<название>, <режиссер>, <год выпуска>).
<артист>(<ФИО>, <звание>, <возраст>, <место работы>).
<снимался>(<название фильма>, <ФИО>, <герой>).

Отношения – правила: в каких фильмах снимался данный артист, возраст артистов, занятых на съемках данного фильма, кто играл в данном фильме указанного героя и какое у него звание, каких артистов предпочитает снимать заданный режиссер и т.д.

15. Предметная область "Контроль знаний по теме "Основные понятия языка Пролог".

Отношения – факты:

<контрольные вопросы>(<номер>, <текст>, <уровень сложности>).
<правильные ответы>(<номер вопроса>, <номер ответа>,
<текст ответа>, <оценка ответа>).
<неправильные ответы>(<номер вопроса>, <номер ответа>,
<текст>, <оценка>).

Отношения – правила должны описать контролирующую программу, в которой задаются вопросы и предлагается меню правильных и неправильных ответов, затем оценивается сделанный обучаемым выбор.

16. Предметная область "Стипендия студентов группы".

Отношения – факты:

<студент>(<номер зачетной книжки>, <ФИО>,<группа>).
<успеваемость>(<номер з/ч книжки>, <семестр>,
<список из пяти оценок за сессию>).

Отношения – правила: кому предоставляется стипендия, размер стипендий в зависимости от успеваемости и т.д.

17. Предметная область "Кулинария ".

Отношения – факты:

<блюдо>(<номер>, <название>, <рецепт приготовления>).
<продукт>(<номер продукта>, <название>, <единица измерения>,
<количество калорий в единице измерения>).
<состав блюда>(<номер блюда>, <номер продукта>,<количество>).

Отношения – правила: какие продукты входят в данное блюдо, в каких блюдах используется данный продукт, какова калорийность каждого продукта, входящего в блюдо, и т.д.

18. Предметная область "Студенты группы".

Отношения – факты:

<студент>(<номер зачетной книжки>, <ФИО>,<группа>).
<возраст>(<номер зачетной книжки>, <возраст>).
<рост>(<номер зачетной книжки>, <рост>).
<вес>(<номер зачетной книжки>, <вес>).

Отношения – правила: какими характеристиками (рост, вес, возраст) обладает студент определенной группы, кто в группе кого старше, выше, тяжелее и т.д.

19. Предметная область "Поставщики продуктов".

Отношения – факты:

<поставщик>(<номер>, <фирма>, <адрес>, <телефон>,<ФИО>).
<продукт>(<номер продукта>, <название>, <единица измерения>,
<количество калорий в единице измерения>).
<поставляет>(<номер поставщика>, <номер продукта>,
<количество>, <цена>, <дата поставки>).

Отношения – правила: какие продукты поставляет заданный поставщик и по какой цене, телефон поставщика, цены на заданный продукт у разных поставщиков.

20. Предметная область "Оценка деятельности преподавателя".

Отношения – факты:

<читает>(<ФИО преподавателя>, <название дисциплины>, <год>,
<семестр>).
<статья>(<ФИО преподавателя>, <название статьи>, <год издания>,
<число страниц>).
<доклад>(<ФИО преподавателя>, <название доклада>,
<дата заслушивания>).
< методическое пособие > (<ФИО преподавателя>, <название >,
<год издания>, <число страниц>, <число соавторов>).
<работа_со_студентами>(<ФИО_преподавателя>, <ФИО_студента>,
<группа>, <название доклада или работы>).

Отношения – правила: методическая работа преподавателей, научная работа преподавателей, учебная работа преподавателей, преподаватель – "отличник" должен иметь все виды работ и т.д.

21. Предметная область "Шахматы".

Отношения – факты:

<шахматист>(<ФИО_шахматиста>, <команда>, <возраст>,
<место_работы>, <телефон>).
<партия>(<номер_игры>, <ФИО_шахматиста_белыми>,
<ФИО_шахматиста_черными>, <дата>, <очки-б>, <очки-ч>,
<количество_ходов>,<итог>).
<ходы>(<номер_игры>, <номер_хода>, <ход-б>, <время-б>, <ход-ч>
<время-ч>).

Отношения – правила: все ходы одной партии, все партии шахматистов одной команды, кто с кем играл и с каким результатом в заданный день, какие результаты у определенного шахматиста при игре белыми, при игре черными и т.д.

22. Предметная область "Управление троллейбусами".

Отношения – факты:

<маршрут>(<номер_маршрута>,<протяженность>,
<время_движения>, <число_остановок>).
<остановка>(<номер_остановки>, <название>).
<имеет>(<номер_маршрута>, <номер_остановки>).
<машина>(<номер_машины>, <состояние>, <срок_службы>).
<обслуживает>(<номер_маршрута>, <номер_машины>, <дата>,

<начало_движения>, <конец_движения>).

Отношения – правила: остановки заданного маршрута, маршруты, число остановок которых больше заданного числа, какие машины обслуживали заданный маршрут на определенную дату, каким маршрутам принадлежит данная остановка, на какой остановке можно пересесть с одного маршрута на другой и т.д.

23. Предметная область "Междугородные переговоры".

Отношения – факты:

<телефон>(<номер_телефона>, <владелец>, <адрес>, <льготы>).

<междугородные_переговоры>(<номер_переговоров>,
<телефон_заказчика>, <город>, <вызываемый_телефон>,
<дата>, <продолжительность>, <стоимость>).

Отношения – правила: с какими городами осуществлялись переговоры с данного телефона, кто владелец телефона, продолжительность переговоров с которого больше заданной величины, с какими городами осуществлялись переговоры за последний месяц и т.д.

Глава 3. ОБЩАЯ СХЕМА ВЫПОЛНЕНИЯ ПРОГРАММЫ НА ЯЗЫКЕ ПРОЛОГ

Прежде чем приступить к чтению теоретического материала, попытайтесь отдать себе отчет, хорошо ли вы понимаете, что такое факт, правило, утверждение, предикат, определение отношения, константа и переменная, какова структура программы на языке Пролог. Без четкого представления сути этих понятий дальнейшее чтение бесполезно. В качестве примера далее рассматривается усеченный вариант программы "Меню", приведенный в первой главе. В дальнейшем в примерах чаще всего будем рассматривать только раздел утверждений ***clauses,*** ваша задача добавить описания.

```
/* Усеченный вариант программы МЕНЮ */
clauses
        /* определение отношения закуска */
        закуска("артишоки").
        закуска("трюфели").
        закуска("салат ").
        /* определение отношения мясо */
        мясо("жаркое").
        мясо("цыпленок").
        /* определение отношения рыба */
        рыба("окунь").
        рыба("судак").
        /* определение отношения десерт */
        десерт("мороженое").
        десерт("земляника").
        десерт("дыня").
        /* определение отношения блюдо */
        блюдо(Y):-мясо(Y).
        блюдо(Y):-рыба(Y).
        /* определение отношения обед */
        обед(X,Y,Z):- закуска(X), блюдо(Y), десерт(Z).
```

3.1. Общие сведения

Программа на языке Пролог представляет собой базу знаний и имеет структуру, представленную на рис. 2.3 в гл. 2. Выполнение программы начинается с момента введения запроса к базе знаний, представляющего собой либо одиночное целевое утверждение, либо конъюнкцию целей.

Например:

в программе на Vip5.2:

```
goal
    закуска(Zac);
    обед(X,Y,Z), рыба(Y), Z="мороженое".
```

в программе на языке Visual Prolog 7

```
clauses
    run():-  nl, сбалансированный_обед(X,Y,Z),
             write(X," ",Y," ",Z), nl, fail.
    run().
```

В процессе выполнения решается задача доказательства целевых утверждений запроса на основе сопоставления с известными утверждениями базы знаний. Сопоставление с фактом может привести к немедленному доказательству целевого утверждения, т.к. факт всегда истинен. Сопоставление с правилом только сводит данную задачу (цель) к совокупности более простых подзадач, описываемых конъюнкцией предикатов-подцелей. Встретившись с конъюнкцией предикатов-подцелей (в запросе или в правиле), Пролог пытается согласовать с базой знаний входящие в конъюнкцию целевые утверждения в том порядке, в каком они записаны (слева направо). Это означает, что не будет обрабатываться некоторое целевое утверждение пока не будет доказана истинность его соседа слева. А сосед справа будет рассматриваться только после доказательства данного целевого утверждения. В запросе могут отсутствовать [*закуска("салат")*] либо присутствовать переменные [*закуска(X)*]. В первом случае решается задача доказательства (проверки) истинности заданного утверждения с использованием фактов и правил базы знаний. Во втором – ищутся те значения переменных (т.е. объекты), для которых целевое утверждение является истинным.

3.2. Модель в виде И-ИЛИ дерева процесса доказательства целевого запроса

В программировании принято для отображения хода выполнения программы использовать графические модели в виде схем алгоритмов. Для программ на Прологе такой графической моделью является модель в виде И-ИЛИ дерева. Эта модель описывает схему выполнения программы при заданном запросе к базе знаний.

И-ИЛИ дерево представляет собой множество вершин и связывающих их ребер (рис. 3.1). Вершины делятся на два вида: незакрашенные кружочки – это вершины И, они соответствуют целевым утверждениям или предикатам (подцелям) тела правила. Закрашенные кружочки – это вершины ИЛИ, они соответствуют разным утвержде-

ниям (вариантам) определения одного и того же отношения. Вершины расположены ярусами (уровнями), каждый ярус (уровень) состоит либо из вершин И, либо из вершин ИЛИ. Совокупность вершин И, выходящих из одной вершины ИЛИ, описывает разбиение сложной задачи на более простые подзадачи, а совокупность вершин ИЛИ, выходящих из одной вершины И, задает всевозможные варианты или пути решения каждой подзадачи.

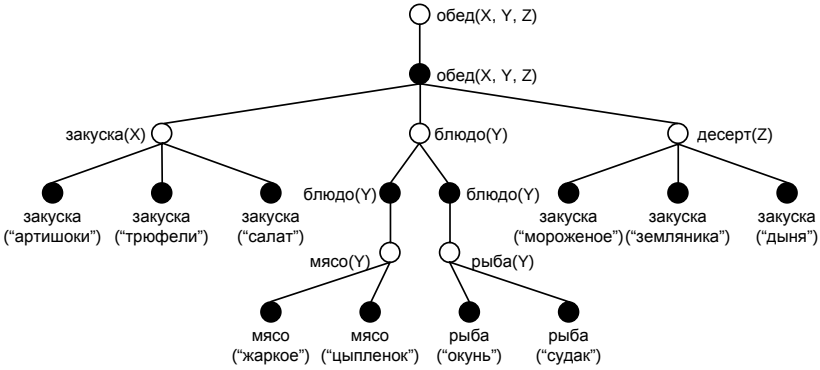


Рис. 3.1. Схема базы знаний "Меню"

Построение дерева идет сверху вниз. Целевому утверждению ставится в соответствие вершина И с именем цели, затем отыскивается в базе знаний определение с этим именем, и к вершине И подсоединяется столько вершин ИЛИ, сколько утверждений в определении соответствующего отношения. Вершинам приписываются заголовки утверждений. Порядок расположения вершин ИЛИ слева направо соответствует порядку следования утверждений в определении (в программе).

Теперь просматриваются вершины ИЛИ. К тем из них, которые соответствуют правилам, подсоединяется столько вершин И, сколько предикатов в теле правила. Вершинам И приписываются соответствующие предикаты. Вершины И располагаются слева направо, в том же порядке, что условия в правиле, и описывают разбиение сложной цели на более простые подцели. Далее процесс повторяется, к каждой вершине И подсоединяются ее вершины ИЛИ (варианты доказательства) и так до тех пор, пока все листовые вершины не будут соответствовать фактам.

Пример. На рис. 3.1 приведено дерево программы "Меню" для целевого утверждения

```
goal
```

```
обед(X,Y,Z).
```

3.3. Алгоритм работы интерпретатора

Выполнение программы можно представить как пошаговый процесс обработки целевых утверждений. Каждый шаг этого процесса делится на 4 фазы:

фаза 1 – сопоставление с образцом;

фаза 2 – унификация аргументов;

фаза 3 – проверка: факт или правило;

фаза 4 – процесс возврата.

Рассмотрим каждую фазу подробнее, иллюстрируя ее выполнение с помощью модели И-ИЛИ дерева.

Фаза 1 – сопоставление с образцом. Обрабатываемое целевое утверждение полностью определяется именем предиката и списком (числом) аргументов. Имя предиката (отношения) и число аргументов являются тем образцом, с которым происходит сравнение при дальнейшем поиске.

В базе знаний по этому образцу ищется ***определение отношения***. Если в базе знаний запрашиваемого отношения нет, то выдается сообщение "no" (нет) и выполнение заканчивается. Если обнаруживается заголовок утверждения с заданным именем и требуемым числом аргументов, то осуществляется переход ко второй фазе выполнения – фазе унификации аргументов.

Фаза 2 – унификация аргументов. На этой фазе проявляется огромная важность переменных в языке, поэтому расширим свои познания о переменных. В процессе выполнения программы переменной может быть присвоено (приписано) значение конкретного объекта. До момента присвоения переменной конкретного значения она имеет так называемое неконкретизированное значение и называется ***неконкретизированной переменной***. Неконкретизированное значение – это символ подчеркивания, за которым следует уникальный номер для данной переменной. Например, _401, _33. Каждая переменная имеет свой уникальный номер.

При унификации последовательно рассматриваются утверждения найденного на первой фазе ***определения*** в том порядке, в каком они расположены в базе знаний. При этом происходит сопоставление аргументов цели (предиката) с аргументами заголовка анализируемого утверждения в порядке расположения аргументов в списке. Аргументами могут быть любые термы.

Аргумент X предиката цели и аргумент Y утверждения базы знаний унифицируются по следующим правилам. Если X и Y – константы или

конкретизированные переменные, то они успешно унифицируются, только если они одинаковы. Если X является константой или конкретизированной переменной, а Y - неконкретизированной переменной, то X и Y успешно унифицируются и Y принимает значение X (и наоборот). Если X и Y неконкретизированные (свободные) переменные, то они успешно унифицируются, при этом они связываются и приобретают одно и то же неконкретизированное значение. Если в процессе доказательства одна из переменных примет значение, то то же значение примет и связанная с ней переменная. Ниже приведены примеры унификации аргументов:

Сопоставление и унификация	*Результат*
джек("личность") ↔ джек("человек")	нет
человек("Джек") ↔ человек("Джек")	да
человек(X) ↔ человек("Джек")	да: X = Джек
размер(X,X) ↔ размер(23,23)	да: X=23
размер(X,23) ↔ размер(12,Y)	да: X=12, Y=23
размер(X,X) ↔ размер(12,23)	нет

При успешном согласовании аргументов найденное в определении предиката утверждение базы знаний помечается маркером и начинает выполняться следующая третья фаза. Каждая цель имеет свой маркер. Если согласование аргументов цели и рассматриваемого утверждения заканчивается неудачей, то осуществляется переход к следующему по порядку утверждению в определении, и так до тех пор, пока не будет достигнут успех, или в базе знаний не окажется больше ни одного утверждения с заданным именем. В последнем случае целевое утверждение ложно, включается механизм возврата, т.е. начинает выполняться 4-я фаза.

Фаза 3 – проверка: факт или правило. В третьей фазе определяется, является ли обрабатываемое утверждение фактом или правилом.

Обрабатываемое утверждение – факт. Если утверждение не содержит символов ":-", то оно является фактом. Т.к. факт всегда истинен, то доказательство текущей цели завершается успешно. Берется следующая цель, и для нее весь процесс повторяется сначала с фазы 1. В И-ИЛИ-дереве найденный факт отмечается маркером (M_1), и осуществляется продвижение слева направо по вершинам И текущего яруса.

Обрабатываемое утверждение – правило. Если сопоставление с образцом и унификация выполнялись не с фактом, а с заголовком правила, то начинает "раскрываться" правило, т.е. доказывается каждое

целевое утверждение в теле правила последовательно слева направо, поэтому текущей целью становится первый предикат в теле правила.

Фаза 4 – процесс возврата (бектрекинг). Процесс возврата инициируется в следующих случаях:

- когда текущая цель оказалась ложной: унификация аргументов закончилась неудачей, т.к. не нашлось ни одного утверждения в определении, аргументы которого были бы сопоставимы с аргументами цели;
- когда программист намеренно создает процесс возврата, воспользовавшись встроенным предикатом fail в качестве условия в правиле.

Этот предикат всегда ложен, он не имеет аргументов. Синтаксис предиката прост: fail. Например, все варианты обедов будут рассмотрены, если ввести следующие утверждения в программу:

```
clauses
        все_обеды(X,Y,Z):-     обед(X,Y,Z),
                               write(X," ",Y," ",Z), nl,
                               fail.
        все_обеды(X,Y,Z).
```

и затем обратиться к ней с вопросом:
в программе на Vip5.2:

```
goal
        все_обеды(X,Y,Z).
```

в программе на Visual Prolog 7

```
class predicates
        все_обеды:() procedure.
clauses
        все_обеды():-   обед(X,Y,Z),
                        write(X," ",Y," ",Z), nl,
                        fail.
        все_обеды().
clauses
        run():- nl, write("Меню-все варианты обедов "), nl, все_обеды().
```

Процесс возврата заключается в пересмотре проделанной работы и попытках передоказать (вновь согласовать) целевые утверждения путем поиска альтернативных путей доказательства. Пусть, например, доказательство текущей цели закончилось неудачей. В результате осуществляется возврат к предыдущей цели, стоящей слева от потерпевшей неудачу, и она становится текущей. При этом все переменные, которые на предыдущем шаге доказательства приняли значения кон-

кретных объектов, расконкретизируются, т.е. приобретают неконкретизированное значение. Обработка предшествующей цели осуществляется не сначала, а как бы продолжается от достигнутого ранее: от утверждения, помеченного маркером цели. При этом берется следующее после отмеченного маркером утверждение в определении отношения (следующий вариант) и начинается процесс унификации аргументов его заголовка и аргументов цели. Если будет найдено новое утверждение в определении, соответствующее целевому, это место помечается, текущая цель является доказанной и осуществляется переход к следующей цели. Если нет другого подходящего утверждения в определении, то данная цель считается недоказуемой и возврат к передоказательству пройденных целей продолжается. При прямом ходе доказательства рассмотрение утверждений в определении отношения всегда начинается с самого начала, при возврате – от утверждения, помеченного маркером.

Пример. Рассмотрим обработку следующего запроса:

```
goal
    блюдо(X), рыба(X).
```

На рис. 3.2 приведена модель процесса выполнения программы в виде И-ИЛИ-дерева.

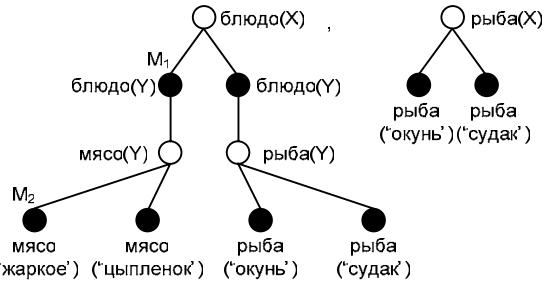


Рис. 3.2. Модель процесса выполнения программы в виде И-ИЛИ-дерева

Первой обрабатывается цель *блюдо(X)*. В базе знаний имеется определение, состоящее из двух утверждений с предикатом *блюдо(Y)* и одним аргументом. Унификация аргументов начинается с первого утверждения *блюдо(Y):-мясо(Y)*.

Аргумент цели и аргумент утверждения Y – переменные, поэтому их унификация заканчивается успешно, в результате переменная цели X оказывается связанной с переменной Y. Обе они имеют одно и то же

неконкретизированное значение X=Y=_64, которое присваивается им автоматически:

блюдо(X)→ блюдо(Y) → блюдо(_64).

Маркером M_1 отмечается найденное правило *блюдо*. Так как утверждение *блюдо* является правилом, то следующей начинает обрабатываться первая в этом правиле подцель *мясо(Y)→ мясо(_64).* В базе знаний есть два утверждения с предикатом *мясо*. Анализ начинается с первого по порядку утверждения *мясо("жаркое").* Так как аргумент цели – неконкретизированная переменная, а аргумент факта – константа, то унификация проходит успешно, в результате переменная Y принимает конкретное значение *Y="жаркое"*.

Маркером M_2 отмечается найденное утверждение. Так как это утверждение – факт, то доказательство цели *мясо(Y),* следовательно, цели *блюдо(X)→ блюдо("жаркое")* заканчивается успешно, и осуществляется переход к цели *рыба(X),* расположенной справа в запросе. При этом переменная X принимает значение *X=”жаркое”*. Область действия переменной – все правило, следовательно, переменная X в цели *рыба(X)* имеет то же самое значение *рыба("жаркое").* При согласовании этой цели с базой знаний процесс унификации аргументов заканчивается неудачей, т.к. нет константы *жаркое* в фактах *рыба(.).* Инициируется процесс возврата к цели *блюдо*, расположенной слева в запросе. Переменная X расконкретизируется, теперь ее значение равно _64.

Процесс обработки цели *блюдо* возобновляется с того момента, на котором он был завершен в предыдущий раз. Таким образом, возврат происходит к подцели *мясо(Y).* Для нее возобновляется процесс унификации аргументов с утверждения, следующего за маркером M_2. В результате переменная Y принимает значение *Y=”цыпленок”*, маркер M_2 устанавливается рядом с найденным утверждением *мясо("цыпленок").* Цель *мясо(Y),* следовательно, цель *блюдо(Y)* доказаны, и осуществляется переход к цели *рыба*, которая имеет теперь следующий вид: *рыба("цыпленок").*

Процесс ее доказательства начинается сначала: находится определение с предикатом *рыба,* рассматривается сначала первое утверждение *рыба("окунь"),* затем – *рыба("судак").* Константы *цыпленок* в этих фактах нет, поэтому доказательство второй цели опять терпит неудачу. На этот раз при возврате первое правило *блюдо* не дает положительного результата, в рассмотрение включается второе правило *блюдо(Y):-рыба(Y),* маркер M_1 передвигается вправо ко второму правилу блюдо. Унификация аргумента цели *рыба(Y)* проходит успешно, переменная Y принимает значение *Y=окунь*. Это же значение приобретает пере-

менная X цели *блюдо(X)→ блюдо("окунь")*, и в силу успешного доказательства этой цели осуществляется переход к цели *рыба(X)* в запросе *рыба("окунь")*. Доказательство этой цели опять начинается сначала. Теперь унификация аргументов проходит успешно, т.е. константы цели и *факта окунь* совпадают. На экран выдается результат:

```
X=окунь во фритюре
X=фаршированный судак
2 Solutions
```

Контрольные вопросы

1. С какого момента начинается выполнение программы?
2. В чем разница в выполнении программы: при наличии переменных в списке аргументов целевого утверждения и при их отсутствии?
3. Что такое И-ИЛИ-дерево?
4. На какие фазы делится процесс обработки целевого утверждения?
5. Какие действия выполняются на фазе "сопоставление с образцом"? Что служит образцом на этом этапе?
6. Опишите фазу унификации аргументов.
7. Как происходит процесс возврата?
8. Когда происходит расконкретизация и каких переменных?
9. Будет ли успешна унификация двух конкретизированных переменных?
10. Будет ли успешна унификация двух неконкретизированных переменных?

Контрольные задания

Для своей программы, полученной в процессе выполнения задания № 2:

- построить И-ИЛИ-дерево для двух запросов;
- использовать fail для вызова возврата;
- проследить на И-ИЛИ-дереве за неудачей и возвратом.

Глава 4. АРИФМЕТИЧЕСКИЕ ВЫРАЖЕНИЯ. ПРЕДИКАТЫ ВВОДА И ВЫВОДА ТЕРМОВ

4.1. Термы

Терм – это синтаксически правильно построенная конструкция Пролога, используемая для представления данных. Терм – это аргумент предиката. Терм может быть константой, переменной или составным термом (структурой, списком, строкой, оператором). В языке Visual Prolog все термы предиката должны быть описаны, т.е. для них должны быть указаны типы – стандартные или определенные в разделе ***domains***. Описание предиката и его термов осуществляется в разделе ***predicates*** или ***class predicates***.

4.2. Константы

Константа в программе на Прологе представляет конкретный объект. Константа может быть задана числом, символическим именем (symbol), строкой (string) и другими составными типами данных.

Числа в Прологе представляются точно так же, как и в процедурных языках программирования, например, 0, –1, 123.4, 0.23, –5. Большинство реализаций Пролога поддерживают целые и действительные числа. В языке Visual Prolog существуют следующие типы чисел: *integer*, *word*, *byte*, *long*, *real* и т.п.

4.3. Переменные

Переменная используется для обозначения одного из представителей некоторого множества(типа) объектов, конкретное значение которого определяется в процессе выполнения программы. В программе переменная задается своим именем (X, U1, _t), правила образования которого рассмотрены в первой главе.

Частным случаем переменной является анонимная переменная, для обозначения которой используется один знак подчеркивания '_'. В естественном языке понятие анонимной переменной равнозначно словам "некто", "нечто". Следовательно, анонимная переменная обозначает объект программы, значение которого никак не влияет на решение задачи и безразлично пользователю. Если в одно утверждение входит несколько анонимных переменных, то они между собой не связаны и интерпретируются неоднозначно. Например:

```
/* Некто имеет книгу */
имеет(_,"книга").
```

```
/* У кого-то есть роман Толстого Л.Н. */
имеет( _, роман( _,"Толстой Л.Н.")).
```

При использовании переменных следует руководствоваться следующими правилами.

В процессе выполнения программы переменная может быть конкретизирована, т.е. ей будет приписано (присвоено) значение конкретного объекта. До момента приобретения переменной конкретного значения она имеет так называемое неконкретизированное значение и называется неконкретизированной переменной. Неконкретизированное значение – это символ подчеркивания, за которым следует уникальный номер для данной переменной. Каждая переменная с новым именем имеет другой уникальный номер.

Переменная может приобрести значение только в процессе унификации, если она в настоящее время неконкретизирована. Если переменная имеет конкретное значение, то нового значения ей присвоить нельзя. Только после расконкретизации, которая происходит в процессе возврата (при бектрекинге), переменная вновь приобретает возможность получать новые значения.

Например, запрос:

```
goal
        X=7, write(X), X = X+2, write(X),nl.
7 No Solutions
```

приведет к отрицательному ответу, а при выполнении следующей программы

```
clauses
        студент("Иванов",943).
        студент("Петров",623).
        студент("Андреева",943).
goal
        студент(X, 943), write(X),nl, fail.
```

на экране будет выведен список всех студентов группы 943, при этом переменная X принимает различные значения фамилий студентов.

```
Иванов
Андреева
No Solutions
```

Область действия переменной – все утверждение от заголовка до точки. Если переменная приобрела значение, то это же значение будут иметь все вхождения этой переменной в списки аргументов предикатов утверждения.

4.4. Арифметические выражения

Язык Пролог не предназначен для программирования задач с большим количеством арифметических операций. Для этого используются процедурные языки программирования. Однако в любую Пролог-систему включаются все обычные арифметические операторы: сложение (+), вычитание (–), умножение (*), деление (/) , целочисленное деление div, остаток от деления целых чисел (mod). В языке Visual Prolog присутствует более широкий набор встроенных арифметических операторов и стандартных функций. С ними Вы можете познакомиться, прочитав соответствующий раздел справочной службы.

Арифметическим выражением является выражение, составленное из целых (и вещественных) чисел с использованием арифметических операторов и встроенных функций. Арифметическое выражение не должно содержать неконкретизированных переменных.

Арифметические выражения вычисляются с помощью системного предиката "=", например: SUM = 2+4. Предикат "=" определен как инфиксный оператор. Его левый аргумент – или число, или неконкретизированная переменная, или выражение, а правый аргумент – арифметическое выражение. Попытка доказательства целевого утверждения X = Y заканчивается успехом в одном из следующих случаев:

- X – неконкретизированная переменная, а результат вычисления Y есть число;
- X – число, которое равно результату вычисления выражения Y;
- X и Y – выражения, значения которых одинаковы.

Цель X = Y не имеет побочных эффектов и не может быть согласована вновь.

Например:

```
D = 10–5      % успех D = 5
4 = 2*4–4     % успех
2*4–4 = 4     % удача
2 = 4–X       % неудача
```

4.5. Предикаты сравнения значений арифметических выражений

К элементам языка относятся и встроенные предикаты сравнения, позволяющие сравнивать значения арифметических выражений. Для некоторого предиката сравнения @ доказательство целевого утверждения X @ Y заканчивается успехом, если результаты вычисления арифметических выражений X, Y находятся в таком отношении друг к

другу, которое задается предикатом @. Такое целевое утверждение не имеет побочных эффектов и не может быть согласовано вновь. Если X, Y не арифметические выражения, возникает ошибка.

С помощью предикатов описываются следующие отношения:

X = Y	X унифицируется с Y
X <> Y	X не унифицируется с Y
X >< Y	X не унифицируется с Y
X > Y	X больше Y
X >= Y	X больше или равно Y
X < Y	X меньше Y
X =< Y	X меньше или равно Y

4.6. Ввод и вывод

С помощью встроенных предикатов ввода-вывода программа, взаимодействуя с пользователем, может принимать от него данные и печатать результаты. Рассмотрим простейшие встроенные предикаты ввода/вывода термов.

Ввод термов. В языке Visual Prolog в консольном режиме для ввода термов предназначено несколько встроенных предикатов, каждый для ввода определенного типа данных: *readLine* : () –> *string Value* – для ввода строки, *read* : () –> _ *Term*– для ввода термов, *readChar* : () –> *char Value* – для ввода символов. По умолчанию данные вводятся с клавиатуры терминала, ввод завершается нажатием клавиши "Enter".

Например, при выполнении цели

```
clauses
    run():-  write("Введите строку "), Line = readLine(),
             write(Line), nl,
             write("Введите символ "), Ch = readChar(),
             write(Ch),nl,
             write("Введите целое число "), X=read(), X=3,
             write(X), nl,
             write("Введите действительное число"),
             Y = read(), Y = 3.5, write(Y),nl,fail.
    run().
```

получим ответ:

```
Введите строку mama
mama
Введите символ r
Введите целое число 3
3
```

```
Введите действительное число 3.5
3.5
Line = mama, Ch = r, X = 3, Y = 3.5
1 Solution
```

Вывод термов. Для вывода термов используется предикат write(T1, T2, …, Tn). Он выводит значения термов T1, T2, …, Tn на текущее устройство вывода, по умолчанию, на экран. Предикат write(…) не допускает повторного согласования и выполняется лишь один раз. Переход на новую строку при печати данных обеспечивается встроенным предикатом nl, название которого образовано аббревиатурой (начальными буквами) слов "newline" (новая строка). Как и write, предикат nl выполняется только один раз. В предыдущем примере значения всех переменных можно было бы вывести с помощью следующего предиката

```
write("Line=",Line,'\n', "Ch=",Ch,'\t', "X=",X,'\t',"Y=",Y,'\n').
```

Чтобы можно было использовать вышеописанные предикаты ввода/вывода, необходимо в файле реализации **.pro* в предложении *open core* добавить подключение класса *console open core, console.*

Контрольные вопросы

1. Что такое константа? Каковы ее виды и формы записи?
2. Что такое переменная? Каковы правила записи имен переменных?
3. Что такое анонимная переменная?
4. Что такое конкретизированное значение переменной?
5. С помощью каких предикатов осуществляется ввод и вывод термов?
6. Какие предикаты арифметических операций сравнения вы знаете? Поясните их.

Глава 5. УПРАВЛЕНИЕ ВЫПОЛНЕНИЕМ ПРОГРАММЫ

В теории программирования доказано, что для описания алгоритма решения задачи достаточно трех управляющих структур: цепочки (последовательное выполнение действий), ветвления (выполнение действий в зависимости от истинности некоторых условий) и цикла (многократное повторение одних и тех же действий, но с разными данными). Язык Пролог относится к декларативным языкам – языкам, описывающим предметную область. Процесс выполнения программы определяется работой интерпретатора. Однако этим процессом можно управлять, используя базовые управляющие структуры в программе. Ниже рассматривается, как реализуются базовые управляющие структуры на языке Пролог.

5.1. Цепочка

Конъюнкция предикатов в теле правила может рассматриваться как последовательность (цепочка) вызовов логических функций с параметрами – аргументами предикатов. Выполняются эти вызовы в порядке их следования в теле правила, если ни одна из них не окажется ложной.

5.2. Выбор среди альтернатив

Несколько утверждений в определении отношения (предиката) рассматриваются как альтернативные варианты одного и того же отношения. Поэтому для реализации ветвления следует создать новое отношение, в описании которого должно быть столько утверждений, сколько веточек в ветвлении. В теле каждого утверждения должны быть описаны условия, при которых выполняется эта веточка. Условия следует формулировать так, чтобы они взаимно исключали друг друга. Тогда всегда будет выполняться одна веточка и при возникновении возврата остальные ветви окажутся ложными и рассматриваться не будут.

Рассмотрим функцию, имеющую два альтернативных определения:

$$Z = \begin{cases} [L/K], & \text{если } L > 20 \\ L \bmod K, & \text{если } L \le 20 \end{cases}$$

где [ ] – знак выделения целой части числа.

На Прологе такая функция может быть описана с помощью двух правил:

```
функz(L,K,Z):- L>20, Z = L div K.
функz(L,K,Z):- L=<20, Z = L mod K.
```

Если теперь к этим двум правилам добавить третье, получим на языке Visual Prolog 7 следующий фрагмент программы:

```
class predicates
    функz:(integer, integer, integer) nondeterm(i, i, o).
clauses
    функz(L,K,Z):- L>20, Z = L div K.
    функz(L,K,Z):- L<=20, Z = L mod K.
class predicates
    вычислить:() procedure.
clauses
    вычислить():-   write("введите L="), L = read(),
                    nl, write("введите K="), K =read(),nl,
                    функz(L,K,Z), write(" Z=",Z), nl, fail.
    вычислить().
```

добавим его в программу (в файл реализации), а затем исправим целевой предикат *run()* как показано ниже.

```
clauses
    run():-   console::init(), вычислить().
```

В результате будет организован следующий диалог с ЭВМ:

```
введите L=32.
введите K=7.
z=4
yes
```

При организации выбора из альтернатив надо предусматривать, чтобы один из вариантов давал решение поставленной задачи.

Рассмотрим часто встречающуюся задачу выбора наибольшего *Z* из двух чисел *X* и *Y*. Предикат *наибольшее(X, Y, Z)* будет иметь следующее определение:

```
class predicates
    наибольшее:(integer, integer, integer) nondeterm (i, i, o).
    наибольшее:(string, string, string) nondeterm (i, i, o).
clauses
    наибольшее(X, Y, X):- X>Y.
    наибольшее(X, Y, Y):- X<=Y.
```

При таком описании предиката *наибольшее(X, Y, Z)* он может использоваться и для сравнения целых чисел и для сравнения строк.

Далее приведем определение предиката "успеваемость", с помощью которого можно классифицировать студентов по среднему баллу Sr:

```
clauses
    успеваемость("отличник",Sr):- Sr > 4.6,Sr <= 5.0.
    успеваемость("хорошист",Sr):- Sr > 3.6,Sr <= 4.6.
    успеваемость("троечник",Sr):- Sr > 2.8, Sr <= 3.6.
```

Рассмотрим еще одну программу, осуществляющую выбор из альтернатив:

```
/*выбор из двух альтернатив
успешна или неуспешна регистрация */
class predicates
    регистрация:()nondeterm.
clauses
    регистрация():-  ввод_шифр_пароль(ID,N),
                     write("регистрация успешна"), nl.
    регистрация() :- write("шифр и пароль не сопоставимы"), nl.
class predicates
    ввод_шифр_пароль:(string,string)nondeterm anyflow.
clauses
    ввод_шифр_пароль(ID, N):-      write("введите шифр:"),
                                   ID = readLine(), nl,
                                   write(" пароль: "),
                                   N = readLine(),
                                   контроль(ID,N).
class facts
    контроль:(string, string).
clauses
/* выбор из множества допустимых шифров и паролей */
    контроль("Г843", "Маркина").
    контроль("Г843", "Семенова").
    контроль("Г251", "Петров").
clauses
    run():-  console::init(),
             регистрация(),!.
    run().
```

Программа будет выполняться следующим образом:

```
введите шифр: Г843.  % ввод шифра
пароль: Маркина.     % ввод пароля
регистрация успешна  % первое правило успешно

введите шифр: Г526.  % ввод шифра
```

```
пароль: Петров.        % ввод пароля
шифр и пароль не сопоставимы
% использован второй вариант
```

В первом случае сопоставление с фактом базы знаний *контроль("Г843", "Маркина")* происходит успешно, отсюда правило *ввод_шифр_пароль* завершается удачно и первое из правил *регистрация* применимо.

Во втором случае факта *контроль("Г526", "Петров")* в базе данных нет, первое правило *регистрация* заканчивается неудачей и выполнение переходит ко второму правилу. Программа выводит на экран сообщение о неудаче.

5.3. Использование fail для организации повторяющегося процесса (цикла)

В Прологе имеется встроенный предикат *fail* (фэйл), который всегда имеет значение "ложь". Предикат не имеет аргументов, поэтому его синтаксис очень прост: *fail*.

Этот предикат нужен потому, что бывают моменты, когда полезно в процессе выполнения сгенерировать неудачу. Одно из применений предиката *fail* – создать процесс возврата для получения всех ответов на запрос или всевозможных вариантов решения некоторой задачи.

Рассмотрим следующий вариант программы "Предоставление помощи":

```
clauses
    помощь(X):-    сотрудник_РГРТУ(X).
                   сотрудник_РГРТУ("Киселев").
                   сотрудник_РГРТУ("Петров").
                   сотрудник_РГРТУ("Иванов").
```

Если вы введете запрос *помощь(X)*, то получите ответы в непонятной форме: *X = Киселев, X = Петров, X = Иванов*.

Как получить в форме отчета сразу весь список людей, имеющих право на получение материальной помощи? Как самому организовать вывод в нужной форме, а не использовать стандартный вывод? Один из путей заключается в использовании предиката *fail* для организации возврата. Проанализируем следующую программу:

```
class predicates
    помощь:()procedure.
clauses
    помощь():-     сотрудник_РГРТУ(X),
```

```
                    write(X," может претендовать",
                    "на материальную помощь"), nl, fail.
% fail добавлен к правилу
    помощь():-      write(" список закончен "), nl.
% второе правило является завершающим условием
class facts
    сотрудник_РГРТУ:(string).
clauses
    сотрудник_РГРТУ("Киселев").
    сотрудник_РГРТУ("Петров").
    сотрудник_РГРТУ("Иванов").
clauses
    run():-  console::init(),
             помощь().
```

Заметьте, что в первое правило *помощь* включен предикат *fail*, который констатирует неудачу и порождает возврат к получению нового решения. Кроме того, добавлено второе правило *помощь*, так называемое завершающее условие, которое обеспечивает успешное завершение выполнения всего правила.

При выполнении этой программы на экране появятся имена тех людей, которым может быть предоставлена помощь:

```
Киселев может претендовать на материальную помощь
Петров может претендовать на материальную помощь
Иванов может претендовать на материальную помощь
Список закончен
```

Общие принципы проектирования повторяющегося процесса при организации возврата с помощью *fail* следующие:

- первое (первые) правило в определении рабочее и включает *fail* для инициирования возврата;
- последнее в определении утверждение всегда имеет значение "истина" и обеспечивает успешное завершение возвратов.

Такой способ организации повторяющегося процесса возможен только, если в теле рабочего правила есть хотя бы один предикат, для которого в базе знаний имеется несколько (>1) вариантов согласований. Для рассмотренной программы таким предикатом является *сотрудник_РГРТУ(X)*. Переменная X при возвратах будет принимать последовательно значения *"Киселев"*, *"Петров"*, *"Иванов"*. Структура повторяющегося процесса зависит от количества предикатов в теле рабочего правила, имеющих несколько вариантов успешного согласования с целью. Если таких предикатов два или более, то получаются

вложенные циклы. Ниже приведен пример организации трех вложенных циклов.

```
/* Программа генерации предложений русского языка */
class facts
        дополнение:(string).
        подлежащее:(string).
        сказуемое:(string) .
clauses
        подлежащее("Он").
        подлежащее("Катя").
        подлежащее("Щенок").

        сказуемое("бежит").
        сказуемое("поет").
        сказуемое("сидит").

        дополнение("по земле").
        дополнение("на мосту").
        дополнение("в лесу").
class predicates
        предложение:(string, string, string) nondeterm anyflow.
        все_предложения:()procedure.
clauses
        предложение(X,Y,Z):-    подлежащее(X),
                                сказуемое(Y),
                                дополнение(Z).
        все_предложения():-     предложение(X,Y,Z),
                                write(" ", X, " ", Y, " ",Z),
                                nl, fail.
        все_предложения().
clauses
        run():-   console::init(),
                  все_предложения().
```

В результате выполнения этой программы будут получены 27 предложений типа "Он поет на мосту", "Щенок бежит по земле" и т.д. Попробуйте.

5.4. Преобразование базы знаний

В программе на языке ***Visual Prolog*** все утверждения находятся в разделе ***clauses***. Предикаты, соответствующие ***правилам***, должны быть описаны в разделе ***predicates.*** А вот факты могут быть описаны в разделах ***predicates, database, facts.*** Если факты описаны в разделах ***database, facts***, то они образуют внутреннюю динамическую базу данных

(фактов), которую можно модифицировать в процессе выполнения программы. Эти факты сохраняются в таблицах, благодаря чему они доступны для изменений. Предикаты, описанные в разделе ***predicates,*** компилируются в двоичный код для максимального быстродействия и изменениям не подлежат.

Над фактами динамической базы данных можно выполнять следующие операции: добавлять факты, удалять, модифицировать, загружать их из файла, записывать в файл, а также обращаться к фактам, находящимся в файле на диске.

Факты динамической базы данных описываются одним из следующих способов.

1.
```
database
        <имя предиката>:(<список аргументов>).
facts
        <имя предиката>:(<список аргументов>).
```

2.
```
database – <имя базы данных>
        <имя предиката>:(<список аргументов>).
facts – <имя базы данных>
        <имя предиката>:(<список аргументов>).
```

После предиката могут быть указаны ключевые слова: *nondeterm* – означает, что в базе данных может быть любое число образцов факта, *determ* – не больше чем один образец факта (т.е. 0 или 1), *single* – один и только один образец факта должен всегда присутствовать в базе данных.

Пусть в базе знаний хранится информация о стипендии студентов в виде фактов *стипендия(<ФИО>, <группа>, <размер_стипендии>)*, например, *стипендия("Иванов", 943, 400).* Вышел приказ о повышении стипендии на 500 рублей, следовательно, требуется скорректировать факты базы знаний. В Прологе есть встроенные предикаты для работы с базой знаний. Познакомимся с тремя из них: *asserta(X), assertz(X), retract(X).*

Предикаты *asserta, assertz*. Два встроенных предиката *asserta, assertz* позволяют добавлять новые факты в динамическую базу данных. Оба предиката действуют одинаковым образом за тем исключением, что *asserta* добавляет факт в **начало** базы, в то время как *assertz* добавляет факт в ее конец. Это отличие можно легко запомнить, учитывая, что "a" является первой буквой английского алфавита, а "z" – его последняя буква. При выполнении целевого утверждения *asserta(X) X* должно иметь значение утверждения-факта, причем достаточно кон-

кретизированного. Необходимо подчеркнуть, что результат добавления в базу данных утверждения не устраняется при выполнении возврата, следовательно, если мы использовали предикат *asserta* или *assertz* для того чтобы добавить новое утверждение, то это утверждение может быть удалено только с помощью предиката *retract.*

Предикат *retract*. Встроенный предикат *retract* позволяет удалять факт из ***динамической базы данных***. Этот предикат имеет один аргумент, с которым должно быть сопоставлено удаляемое утверждение. Указанный аргумент должен быть достаточно конкретизирован, чтобы можно было определить предикат удаляемого факта. При попытке выполнить целевое утверждение *retract(X)* находится первое утверждение в базе знаний, с которым может быть унифицирован X, и это утверждение удаляется. При возникновении процесса возврата просматривается база знаний, начиная с места удаленного утверждения, чтобы найти другое сопоставимое утверждение. Если такое утверждение находится, то оно также удаляется и т.д. Заметим, что если утверждение было удалено, то оно ни при каких условиях не будет восстановлено. Если в некоторый момент в базе знаний не находится утверждения X, являющегося аргументом цели *retract*(X), то согласование этой цели с базой знаний заканчивается неудачей.

Унификация аргументов при согласовании цели *retract*(X) приводит к тому, что в дальнейшем X дает возможность получить точное представление об удаляемом утверждении и получаемая при этом информация может быть использована в следующих целевых утверждениях.

Вернемся к нашему примеру. Перед нами стоит задача факты *стипендия( )* изменить так, чтобы размер стипендии увеличился на 500 единиц. Программа для решения этой задачи приведена ниже:

```
/* Увеличение стипендии */
class facts
        стипендия:(string, integer, integer).
        стипендия1:(string, integer, integer).
class predicates
        увеличение_стипендии:(integer)procedure anyflow.
        вывод_стипендии:()procedure.
clauses
/* Изменить размер стипендии, создав новые факты
стипендия1 */
        увеличение_стипендии(H):-
                retract(стипендия(Fio, Ngr, St)),
                St1 = St+H,
                asserta(стипендия1(Fio, Ngr, St1)),
```

```
                fail.
/* Факты стипендия1 переписать в факты стипендия */
    увеличение_стипендии(H):-
                retract(стипендия1(Fio, Ngr, St)),
                asserta(стипендия(Fio, Ngr, St)),
                fail.
/* Успешно завершить процесс */
    увеличение_стипендии(H):-       write("Процесс закончен"),nl.

    вывод_стипендии():-     стипендия(Fio, Ngr, St),
                            write(Fio, " ", St), nl,
                            fail.
    вывод_стипендии():-     write("Вывод закончен"),nl.

    стипендия("Иванов", 943, 800).
    стипендия("Петров", 943, 700).
    стипендия("Серебряный", 143, 800).
clauses
    run():-         console::init(),
                    вывод_стипендии(),
                    увеличение_стипендии(500),
                    вывод_стипендии().
```

При доказательстве предиката *retract(стипендия(Fio, Ngr, St))* в базе знаний отыскивается первый факт *стипендия("Иванов", 943, 800).* В результате унификации аргументов переменная *St* принимает значение *800,* затем факт *стипендия("Иванов", 943, 800)* удаляется из базы знаний. Вычисляется значение *St1 = St + 500 = 1300,* и при выполнении целевого утверждения *asserta(стипендия1(Fio, Ngr, St1))* в начало базы знаний заносится новый факт *стипендия1("Иванов", 943, 500).* Затем предикат *fail* инициирует возврат, и вышеописанный процесс повторяется.

5.5. Накопление суммы

Предположим, необходимо создать программу, которая обрабатывает ваши расходы во время командировки. В табл. 5.1представлены расходы расходы по дням.

Один из способов представления этой информации на Прологе – записать факты следующего формата: *расходы(<месяц>, <день>, <проезд>, <еда>, <гостиница>).* Например:

```
расходы(май, 1, 100, 300, 150).
расходы(май, 2, 250, 350, 200).
```

Табл. 5.1

Дата	Проезд	Еда	Гостиница	Всего
май 1	100	300	150	550
май 2	250	350	200	800
май 3	150	400	180	730
май 4	200	450	300	950
май 5	150	300	150	600
май 6	260	420	150	830
май 7	1110	2220	1130	4460

Задача заключается в том, чтобы с помощью программы можно было бы найти суммарные расходы за весь период, отдельно вычислить расходы за проезд, еду и гостиницу.

Перед нами стоит задача – накопление суммы расходов за все дни. В процедурных языках накопление суммы осуществляется организацией цикла, в тело которого входит следующий оператор присваивания: S := S + <слагаемое>; в соответствии с которым к предыдущему значению суммы прибавляется очередное слагаемое, и получается новое значение суммы. В Прологе подобная операция недопустима, т.к. если переменная имеет конкретное значение, то ей уже другое значение присвоить нельзя.

Одним из методов решения таких задач является использование вышеописанных предикатов, модифицирующих базу знаний. Опишем в разделе ***facts*** факт *summy(проезд, питание, гостиница) determ.* Затем с помощью предиката *fail* организуем просмотр всех фактов о расходах и будем накапливать новые суммы, удаляя старый факт *retract(summy(SPr1, SPi1, SGo1))* из базы знаний и записывая новый *asserta(summy(SPr2, SPi2, SGo2))* с вновь полученным значением суммы в начало базы знаний.

```
/* Программа "Командировочные расходы" */
domains
        месяц = string.
        день = integer.
        проезд = real.
        питание = real.
        гостиница = real.
/*описание фактов динамической базы данных */
class facts
        summy:(проезд, питание, гостиница)determ.
class predicates
        расходы:(месяц, день, проезд, питание, гостиница)
                        nondeterm anyflow.
```

```
        расчет_итого:(месяц, проезд, питание, гостиница, real)
                        nondeterm (i, o, o, o, o).
    clauses
        расходы("май", 1, 100.0, 300.0, 150.0).
        расходы("май", 2, 250.0, 350.0, 200.0).
        расходы("май", 3, 150.0, 400.0, 180.0).
        расходы("май", 4, 200.0, 450.0, 300.0).

        расчет_итого(M, _ , _ , _ , _ ) :-
                assert(summy(0.0,0.0,0.0)),
                /* начальные значения */
                расходы(M, _ , Pr, Pi, Go),
                /* расходы за текущий день*/
                retract(summy(SPr1, SPi1, SGo1)),
                /*расходы за предыд. дни */
                SPr2 = SPr1 + Pr,
                SPi2 = SPi1 + Pi,
                SGo2 = SGo1 + Go,
                /*к итоговым суммам добавляем расходы за день */
                asserta(summy(SPr2, SPi2, SGo2)),
                /*сохраняем новые итоговые суммы*/
                fail.
        /* порождаем процесс возврата */
        расчет_итого(M, SPr, SPi, SGo, S):-
                retract(summy(SPr, SPi, SGo)),
                /* удаляем факт и извлекаем итоговые суммы */
                S = SPr + SPi + SGo.
                /*находим общую сумму расходов */
    clauses
        run():-   console::init(),
                  расчет_итого("май", SPr, SPi, SGo, S),
                  write(SPr," ", SPi," ", SGo," ", S), nl,fail.
        run().
```

Аналогично можно организовать вычисление средних значений, поиск максимального и минимального значений некоторого аргумента факта.

5.6. Создание бесконечных альтернатив при помощи repeat

Другим методом управления процессом выполнения программы является использование предиката *repeat*. Он предназначен для организации повторяющихся процессов в том случае, когда в теле правила

нет многократно согласующихся с базой знаний предикатов. Предикат *repeat* должен быть описан в программе следующим образом:

```
clauses
        repeat().
        repeat():- repeat().
```

Рекурсивное определение выглядит как петля (цикл) и таковой является. Что произойдет, если мы поместим предикат *repeat* в тело одного из наших правил?

Во-первых, это целевое утверждение всегда согласуется с базой знаний, так как имеется факт repeat (первое утверждение в определении). Во-вторых, в процессе возврата целевое утверждение repeat может быть согласовано бесконечное число раз благодаря наличию рекурсивно определенного правила. Выполнение правила, в котором находится repeat, закончится успешно только в том случае, если все целевые утверждения, стоящие после repeat, в процессе согласования с базой знаний окажутся истинными, в противном случае возникает бесконечный цикл (произойдет зацикливание). При организации повторяющихся процессов предикат repeat можно использовать и в том случае, когда в теле правила нет многократно согласующихся с базой знаний предикатов.

Приведем примеры использования предиката *repeat*. В первой программе описан датчик случайных чисел. С использованием repeat организованы два цикла: во внутреннем – создается последовательность случайных чисел, оканчивающаяся числом 5, внешний цикл выполняется 5 раз и предназначен для генерации 5-ти случайных последовательностей.

```
/* Программа генерации случайных последовательностей
по запросу p */
class facts
        seed:(integer)determ.
        n:(integer)determ.
class predicates
        rand:(integer, integer) determ anyflow.
        случайные_числа:()nondeterm.
        p:()nondeterm.
        repeat:() multi.
clauses
        repeat().
        repeat():- repeat().
        seed(13).
        /* Датчик случайных чисел */
```

```
        rand(N,R):-     retract(seed(S)),
                        S1 = (125*S+1) mod 4096,
                        asserta(seed(S1)),
                        N = S1 mod R +1.
        /* Генерация случайной последовательности,
        заканчивающейся числом 5 */
        случайные_числа():-     repeat(),
                                rand(N,10),
                                write(N," "),
                                N=5, nl, !.
        /* Внешний цикл – генерирует пять случайных
        последовательностей */
        p():-   assert(n(0)),
                repeat(),
                случайные_числа(), nl,
                retract(n(K)),
                K1 = K + 1,
                asserta (n(K1)),
        K1 = 5, retract(n(_)),!.
    clauses
        run():-  console::init(),
                 p(), fail.
        run().
```

5.7. Ввод и вывод фактов динамической базы данных

Заполнение динамической базы фактов утверждениями, добавление новых фактов осуществляются с помощью встроенного предиката ***file::consult(X).*** Здесь X – это строка, представляющая собой имя файла, из которого берутся вводимые в базу знаний утверждения. Целевое утверждение ***file::consult(F)*** добавляет факты из файла F в конец существующей базы данных.

Сохранить факты динамической базы данных в файле позволяет встроенный предикат ***file::save(X).*** Строка **X** также задает имя файла или путь к нему. Следует учитывать, что при указании пути обратную косую черту надо повторить дважды.

Пример использования этих предикатов приведен ниже. В разделе цели сначала с помощью предиката ***consult*** из текстового файла *"Styd.dba"* читаются факты о студентах, затем с помощью предиката ***dobavit*** организуется процесс ввода новой информации о студентах, затем с помощью предиката ***save*** все факты записываются в файл *"Styd.dba"*. Таким образом, происходит обновление базы данных.

В этой программе (консольный режим) в цикле осуществляется ввод информации о студентах и запись полученных данных в базу зна-

ний в виде отношения *студент(<ФИО>, <группа>, <рост>, <вес>)*. Признаком окончания ввода данных служит слово "конец", введенное вместо фамилии. Предикат *вывод_студент* позволяет просмотреть введенные в базу знаний отношения.

```
implement main
    open core, console
constants
    className = "main".
    classVersion = "$JustDate: $$Revision: $".
clauses
    classInfo(className, classVersion).
/* Формирование базы данных о студентах */
domains
    fio=string.gruppa=integer.rost=real.ves = real.
class facts – studentDB
    студент:(fio, gruppa, rost, ves).
class predicates
    % Вывод всех фактов БД
    вывод_студент:()procedure.
     % Запись факта в БД
    запись_в_бд:(fio, gruppa, rost, ves)procedure.
clauses
     /*Если есть уже такой факт в БД, то не записывать */
    запись_в_бд(Fam, Ngr, Rost, Ves):-
                    студент(Fam, Ngr, Rost, Ves),!.
    /* Если нет, то записать */
    запись_в_бд(Fam, Ngr, Rost, Ves):-
                    assert(студент(Fam, Ngr, Rost, Ves)).

    % Вывод всех фактов БД
    вывод_студент():-
            stdio::nl(),
            stdio::write("Студенты и их характеристики","\n"),
            stdio::write("Фамилия ", "Группа ", "Рост ", "Вес "),
            stdio::nl,
            студент(Fio, Gruppa, Rost, Ves),
            stdio::write(Fio," ",Gruppa, " ", Rost," ", Ves),
            stdio::nl, fail.
    вывод_студент():-
            stdio::nl,
            stdio::write("Вывод окончен"), stdio::nl.
class predicates
    dobavit:()nondeterm.
    контроль:(fio)nondeterm .
    repeat:() multi.
```

```
clauses
        repeat().
        repeat():- repeat().
        dobavit():-
                write("Введите данные "),nl,
                repeat(),/* Повторяем */
                write("Фамилия: "),Fam = readLine(),
                /* Вводим фамилию */
                контроль(Fam).
                /*Контроль записи */

        контроль("конец"). /* Конец процесса ввода */
        контроль(Fam):-
                 /* Продолжение ввода */
                write("Группа "), Ngr = read(),
                /* Вводим номер группы */
                write("Рост "), Rost = read(),
                /* Вводим рост */
                write("Вес "), Ves = read(),
                /* Вводим вес */
                H = readLine(),
                запись_в_бд(Fam, Ngr, Rost, Ves),
                fail.
                /* инициируем возврат для повторения ввода данных
                о следующем студенте */
clauses
        run():-  console::init(),
                file::consult("D:\\Visual Prolog Projects\\
                        VvodBd\\Styd.dba",
                        studentDB),
                вывод_студент(),
                dobavit(),
                вывод_студент(),
                file::save("D: \\Visual Prolog Projects\\
                        VvodBd\\Styd.dba",
                        studentDB),!.
        run().
end implement main
```

Пример текстового файла "Stud.dba":

```
clauses
        студент("Перепелкин И.В.",343, 180, 78).
        студент("Родионова И.П.",344, 170, 62).
……..
```

Пример выполнения контрольного задания

Задание: получить для каждой группы список студентов и определить число студентов в группе. Найти аудитории с максимальным и минимальным числом мест.

Рассмотрим решение этой задачи (Проект ***Gui_Raspisanie***) в режиме работы с графическим интерфейсом. Правила создания приложения с графическим интерфейсом рассмотрены в гл. 11, и прежде чем выполнять задание прочитайте нужные пункты этой главы.

Выполнение программы (рис. 5.1) происходит следующим образом. Открывается главное окно задач GUI_Raspisanie, его меню содержит пункты: "Операции с базой фактов", "Запросы". Могут быть выполнены две операции с БД: "ЗагрузитьБД" из файла и "СохранитьБД" в файле. Пункт меню "Запросы" включает подпункты: "Открыть форму запросов", "Вывести списки групп".

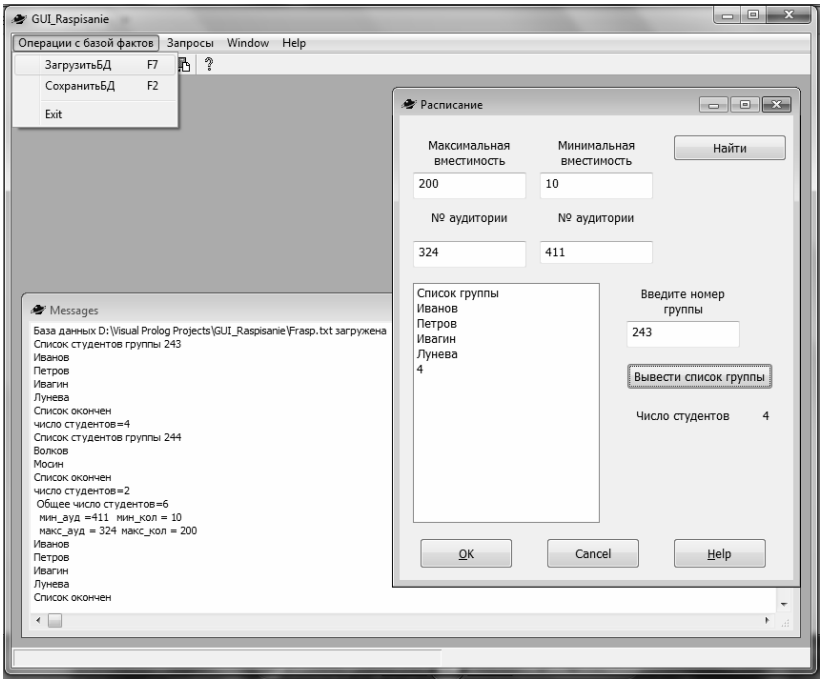


Рис. 5.1. Окно задач программы ***Gui_Raspisanie***

Форма запросов приведена на рис. 5.1. В ней при нажатии кнопки "Найти" выводится информация об аудиториях с минимальной и мак-

симальной вместимостью, при нажатии кнопки "Вывести список группы" в окне вывода появляется список группы и количество студентов в группе, номер которой задан пользователем в текстовом поле "Введите номер группы".

При работе в режиме графического интерфейса необходимо создать пакет *Prasp*, в нем класс *Crasp* и форму *FrmRasp*. Форму наполнить нужными управляющими элементами, изменить пункты меню главного окна задач. Далее к управляющим элементам меню и формы подключить соответствующие обработчики. Ниже приведены листинги модулей соответственно: файл реализации класса *Crasp.pro*, объявление класса *Crasp.cl*, фрагменты файла реализации формы *frmRasp.pro* и фрагменты файла реализации главного окна *TaskWindow.pro*, в которых описаны все обработчики и сама база знаний.

```
/* Пример выполнения задания  в режиме работы с графическим
интерфейсом */
/* файл реализации класса Crasp.pro */
implement crasp
        open core,stdio
constants
        className = "Prasp/crasp".
        classVersion = "$JustDate: $$Revision: $".
clauses
        classInfo(className, classVersion).
        /* Пример выполнения задания 4 */
class facts – raspisanieDB /* Описание фактов БД */
        kol:(kol_vo) determ.
        студент:(fio, ngruppa).
        sum:(kol_vo) determ.
        группа:(ngruppa, fucultet, kol_vo, fio).
        дисциплина:(shifr, discip, kol_vo).
        преподаватель:(fio, kaf, tel).
        аудитория:(nau, tip, kol_vo).
        расписание_занятий:(ngruppa, shifr, fio, nau, denj, vremja, c_z).
        max_min:(kol_vo, nau, kol_vo, nau)determ.
clauses
        студент_п(Фио, НомерГР):-студент(Фио, НомерГр).
class predicates
        /* Предикат перезаписи БД фактов из файла FileName */
        reconsult : (string FileName).
clauses
        reconsult(Filename) :-
                retractFactDB(raspisanieDB),
                file::consult(Filename,raspisanieDB).
```

```
/* Предикат выполняется при выборе пункта меню
"ЗагрузитьБД"*/
class facts
        currentDirectory: string:="".
/* Описание факта currentDirectory */
clauses
        загрузитьБД():-
                currentDirectory:=directory::getCurrentDirectory(),
                directory::setCurrentDirectory("..\\"),
                Filename = vpiCommonDialogs::getFileName
                ( "", ["Text files (*.txt)", "*.*"],
                "Загрузка БД о студентах", [], ".", _),
                directory::setCurrentDirectory(currentDirectory),! ,
                reconsult(Filename),
                stdIO::writef("База данных % загружена\n", Filename).
        загрузитьБД().
/* Предикат выполняется при выборе пункта меню
"СохранитьБД" */
clauses
        сохранитьБД():-
                currentDirectory:=directory::getCurrentDirectory(),
                directory::setCurrentDirectory("..\\"),
                Filename =vpiCommonDialogs::getFileName( "",
                ["Text files (*.txt)", "*.*"],
                "Загрузка БД о студентах", [], ".", _),
                directory::setCurrentDirectory(currentDirectory), ! ,
                file::save(Filename,raspisanieDB),
                stdIO::writef("База данных % сохранена\n", Filename).
        сохранитьБД().
clauses
/* Вывод списка фамилий студентов группы
и определение числа студентов в группе */
        список_группы(NG, _):-   assert(kol(0)),
                                 студент(F, NG), write(F), nl,
                                 retract(kol(N)),
                                 N1 = N + 1,
                                 asserta(kol(N1)), fail.
        список_группы(NG,Nst):-  retract(kol(Nst)),
                                 write("Список окончен"),nl.
/* Для каждой группы выводится список студентов и определяет-
ся число студентов в группе, а также общее число студентов */
        списки():-       assert(sum(0)),
                         группа(Ng, _, _, _),
                         write("Список студентов группы ", NG), nl,
                         список_группы(Ng,Nst),
                         write("число студентов="), write(Nst), nl,
```

```
                        retract(sum(S)),
                        S1 = S + Nst,
                        asserta(sum(S1)),
-                       fail.
        списки():-      retract(sum(S)),
                        write(" Общее число студентов="),
                        write(S),nl.
/* Определяются аудитории с максимальным и минимальным чис-
лом мест */
        аудитории_макс_мин(Aumax, Nmax, Aumin, Nmin):-
                         assert(max_min(0, 0, 1000, 1)),
                        аудитория(Au, _, Ns),
                        retract(max_min(Max, Amax, Min, Amin)),
                        наибольшее(Au, Ns, Amax, Max, Amax1, Max1),
                        наименьшее(Au, Ns, Amin, Min, Amin1, Min1),
                        asserta(max_min(Max1, Amax1, Min1, Amin1)),
                        fail.
        аудитории_макс_мин(Aumax, Nmax, Aumin, Nmin):-
                        retract(max_min(Nmax, Aumax, Nmin, Aumin)).
                        /*Предикаты определения наибольшего
                        и наименьшего из двух элементов*/
                        наибольшее(Au,Ns,Amax,Max,Amax,Max):- Max>Ns.
                        наибольшее(Au,Ns,Amax,Max,Au,Ns):- Ns>=Max.

        наименьшее(Au,Ns,Amin,Min,Amin,Min):- Min<Ns.
        наименьшее(Au,Ns,Amin,Min,Au,Ns):- Ns<=Min.
end implement crasp

/*Описание класса Crasp - файл crasp.cl*/
class crasp : crasp
        open core
predicates
        classInfo : core::classInfo.
domains /*Типы данных */
        fio=string. fucultet=string. discip=string. tip=string.
        kaf=string. tel=string. denj=string. c_z = string.
        ngruppa=integer. kol_vo=integer. shifr=integer.
         nau=integer. vremja = integer.
predicates /*Предикаты, доступные другим классам */
        список_группы:(ngruppa, kol_vo)determ (i,o).
        списки:() determ.
        аудитории_макс_мин:(nau, kol_vo, nau, kol_vo)
                        nondeterm (o,o,o,o).
        наибольшее:(nau, kol_vo, nau, kol_vo, nau, kol_vo)
                        nondeterm (i,i,i,i,o,o).
        наименьшее:(nau, kol_vo, nau, kol_vo, nau, kol_vo)
```

```
            nondeterm(i,i,i,i,o,o).
    загрузитьБД:().
    сохранитьБД:().
    студент_п:(fio, ngruppa)nondeterm (i,o) (i,i) (o,o) (o,i).
end class crasp

/*Файл реализации формы – файл frmRasp.pro (фрагменты) */
implement frmRasp
    inherits formWindow
    open core, vpiDomains

/*Здесь пропущен текст……..*/
clauses % Создание формы и ее открытие
    display(Parent) = Form :-
            Form = new(Parent),
            Form:show().

clauses % Инициализация парам. формы
    new(Parent):-   formWindow::new(Parent),
                    generatedInitialize().

/* Обработчик кнопки "Найти" аудитории с максимальной
и минимальной вместимостью*/
predicates
    onMax_min_a_puchClick : button::clickResponder.
clauses
    onMax_min_a_puchClick(_Source) = button::defaultAction:-
            crasp::аудитории_макс_мин(Na1,Ka1,Na2,Ka2),
            max_ctl:setText(tostring(Ka1)),
            min_ctl:setText(tostring(Ka2)),
            aumax_ctl:setText(tostring(Na1)),
            aumin_ctl:setText(tostring(Na2)),
            stdio::write(" мин_ауд =",Na2," мин_кол = ",Ka2,
            "\n макс_ауд = ",Na1," макс_кол = ",Ka1), stdio::nl,!.
    onMax_min_a_puchClick(_Source) =button::defaultAction().

/* Вывод списка группы в listbox:listсписок_ctl */
predicates
    списокГр:(crasp::ngruppa)procedure.
clauses
    списокГр(Ngruppa):-     listсписок_ctl:add("Список группы"),
                    crasp::студент_п(Фио, Ngruppa),
                    listсписок_ctl:add(Фио), fail.
    списокГр(Ngruppa).
/*Обработчик кнопки "Вывести список группы"*/
predicates
    onPushButtonClick : button::clickResponder.
```

```
clauses
        onPushButtonClick(_Source) = button::defaultAction:-
                Ngruppa = ngr_ctl:getText(), списокГр(toterm(Ngruppa)),
                crasp::список_группы(toterm(Ngruppa), Kl_vo),
                listсписок_ctl:add(tostring(Kl_vo)),
                kol_ctl:setText(tostring(Kl_vo)),!.
        onPushButtonClick(_Source) = button::defaultAction.
% This code is maintained automatically,
% do not update it manually. 22:52:06-28.7.2010
%.....................................
% end of automatic code
end implement frmRasp

/* Фрагменты файла реализации главного окна задач –
TaskWindow.pro*/
implement taskWindow
        inherits applicationWindow
        open core, vpiDomains
/* Здесь пропущен текст ……*/

/* Обработчик события "Выбор пункта меню "Загрузить БД""*/
predicates
        onFileNew : window::menuItemListener.
clauses
        onFileNew(_Source, _MenuTag):-
                crasp::загрузитьБД(),crasp::списки(),!.
        onFileNew(_Source, _MenuTag).
predicates
/* Обработчик события "Выбор пункта меню "Edit/Открыть фор-
му запросов""*/
        onEditUndo : window::menuItemListener.
clauses
        onEditUndo(W, _MenuTag):- S = frmRasp::new(W), S:show().

end implement taskWindow
```

Контрольные вопросы

1. Как описывается выбор из альтернатив на Прологе?
2. Как выполняется программа при наличии нескольких утверждений в определении?
3. Для каких целей используется встроенный предикат fail?
4. Каковы правила организации повторяющегося процесса с помощью fail?
5. Что такое динамическая база фактов и как ее создать?

6. Какие предикаты преобразований динамической базы данных Вы знаете? Поясните, как они выполняются?
7. Как на Прологе организовать вычисление суммы, произведения?
8. В каких случаях необходимо использовать предикат repeat? Как его определить?
9. Какие встроенные предикаты позволяют сохранить динамическую базу фактов в файле и восстановить ее из файла?
10. Что описывают ключевые слова nondeterm и determ?

Контрольные задания

Написать для своего варианта из задания № 2 следующие отчеты.

1. Получить список всех столиц для каждой части света, подсчитать для них среднее число жителей и найти столицу с максимальным числом жителей.

2. Получить списки семей с одним ребенком, с двумя детьми, сколько детей приходится на одного родителя.

3. Получить списки романов, пьес, повестей и подсчитать для каждой категории среднее число страниц.

4. Получить списки студентов для каждой группы, подсчитать для каждой группы число студентов в ней, число изучающих английский язык.

5. Получить ведомость экзамена по некоторой дисциплине для заданной группы, подсчитать средний балл для оценок выбранного экзамена. Подсчитать число отличников, хорошистов в группе.

6. Получить список лекарств в данной аптеке по группам. Подсчитать среднюю цену заданного лекарства в аптеках города.

7. Получить список фильмов, идущих в каждом кинотеатре. По каждому фильму в заданном кинотеатре подсчитать общую сумму выручки.

8. Получить список изданий для каждого подписчика и подсчитать, на какую сумму он подписался.

9. Для каждой личности получить список всех друзей и подсчитать их средний возраст.

10. Получить список пациентов, получавших больничный в текущем году, определить их количество. Определить самого старого и самого молодого пациента поликлиники.

11. Получить список читателей. Определить средний возраст читателей и количество должников. Для заданного читателя определить количество взятых за определенный период книг.

12. Получить для каждого преподавателя список читаемых им дисциплин. Определить максимальный, минимальный и средний оклады преподавателей.

13. Получить для каждой группы список студентов и определить число студентов в группе. Осуществить проверку расписания: во всех ли назначенных по расписанию аудиториях хватает мест для студентов группы. Определить аудитории с максимальным и минимальным числом мест.

14. Получить список артистов, занятых в данном фильме, и определить их средний возраст. Для каждого режиссера подсчитать число его фильмов.

15. Для каждого вопроса получить список правильных и неправильных ответов, подсчитать их количество. Найти вопросы с максимальным и минимальным уровнем сложности.

16. Для каждой группы получить список студентов с указанием размера его стипендии, подсчитать общую сумму стипендий в группе, а также средний размер стипендии.

17. Для каждого блюда получить список необходимых продуктов и подсчитать общую калорийность блюда. Найти продукты с максимальной и минимальной калорийностью

18. Для каждой группы получить список студентов с указанием их возраста, веса и роста, определить средний возраст, вес и рост в группе. Найти самого высокого студента.

19. Для каждого продукта получить список поставщиков, определить их количество, максимальную и минимальную цену продукта.

20. Для каждого преподавателя получить список его статей, определить их количество и общее число страниц. Найти статью с наибольшим и наименьшим числом страниц.

21. Получить список шахматистов с указанием числа выигранных и проигранных партий, для каждой партии получить список ходов и подсчитать общее время белых и черных.

22. Для каждого маршрута получить список остановок и подсчитать их число, подсчитать число обслуживающих маршрут машин на текущую дату. Найти маршруты максимальной и минимальной протяженности.

23. Для каждого телефона получить список его междугородних переговоров и подсчитать общую продолжительность и стоимость. Найти владельца с наибольшей продолжительностью междугородных переговоров.

Глава 6. РЕКУРСИЯ

Итерационные процессы. Процесс, состояние которого на текущем шаге зависит от состояния этого же процесса на предыдущих r шагах, называется многошаговым итерационным процессом и описывается следующими рекуррентными соотношениями:

(6.1а)(начальные условия)

$$y_0 = a_0,\ y_1 = a_1,\ldots,\ y_{r-1} = a_{r-1};$$

(6.1б)(текущий момент)

$$y_k = f(y_k, y_{k-1}, y_{k-2},\ldots, y_{k-r}, X); \qquad (6.1)$$

(6.1в)(условие окончания)

$$k = r,\ r + 1,\ \ldots,\ \ldots,\ k = N,$$

где f – произвольная функция, которая может быть задана аналитически, словесно, в виде алгоритма и т. д.; y_i – состояние процесса на i-м шаге; X – параметры процесса.

Частным случаем (6.1) является одношаговый итерационный процесс $y_n=f(y_{n-1})$, в котором текущее состояние зависит только от предыдущего. Примерами итерационных процессов являются:

- вычисление суммы: $S_0=0$, $S_n=S_{n-1}+u_n$, $n=1, 2,\ldots$;
- вычисление произведения: $P_0=1$, $P_n=P_{n-1}\cdot m_n$, $n=1, 2,\ldots$;
- вычисление факториала: $0! = 1$, $k! = (k – 1)!\cdot k$;
- вычисление чисел Фиббоначи: $F_1=1$, $F_2=2$, $F_n=F_{n-1}+F_{n-2}$;
- многие другие.

Программирование итерационных процессов по рекуррентным формулам (6.1) может осуществляться двумя способами: методом итерации (повторений) и методом рекурсии. Метод итераций рассмотрен в четвертой главе. Для его реализации организуется цикл, на каждом шаге которого текущее состояние процесса получается путем извлечения и стирания из динамической базы данных информации о его предшествующих состояниях, вычисления нового значения по формуле (6.1б) и записи вновь полученных значений на место соответствующих предшествующих состояний. Так, например, рекуррентным соотношениям, описывающим вычисление $F = N!$, может быть поставлена в соответствие следующая программа на языке *Visual Prolog*, в которой факториал вычисляется методом итерации.

```
/*Фрагмент программы вычисления факториала
методом итерации */
class facts
        фак:(integer, real)determ.
class predicates
```

```
        факториал:(integer, real) determ (i,o).
        repeat:() multi.
clauses
        repeat().
        repeat():- repeat().

        факториал(N, F):-
                asserta(фак(0, 1)),
                /* запись в БЗ начальных условий */
                repeat, /* заголовок цикла */
                retract(фак(K, P)),
                /* стирание из БЗ предыдущих значений */
                K1 = K + 1,
                P1 = P * K1,
                asserta(фак(K1, P1)),
                /* запись в БЗ новых значений */
                K1 = N, !, /* условие окончания цикла */
                retract(фак (N, F)). /* вывод результатов */
clauses
        run():-  console::init(),
                write("Введите число, факториал которого Вы
                        хотите получить "),
                K = read(), факториал(K, F),
                write("факториал(", K, ")=", F), nl, !.
        run().
```

Обратите внимание на знак отсечения ***!*** в конце тела правила *факториал(N, F)*, без этого знака процесс вычислений может зациклиться.

6.1. Введение в рекурсию

В математике **рекурсия** определяется как способ описания объектов, данных, процессов или функций через **самих себя**. Рекуррентные соотношения (6.1) представляют собой как раз такое описание. В них процесс на *n*-ом шаге определяется через тот же самый процесс на (*n*–1)-м шаге. Чтобы перейти к рекурсивному определению в программе, необходимо для описания процесса определить предикат, который обращался бы к самому себе (т.е. к описанию того же процесса) при других значениях аргументов (например, при другом номере шага *KI = K – 1*). При этом как бы допускается, что уже имеется описание процесса, которое правильно выполняется для аргумента *K – 1* и, следовательно, позволяет получить состояние y_{k-1} на предшествующем шаге. Например, рекуррентным соотношениям, описывающим процесс вычисления факториала *F = N!* и вычисления чисел Фибоначчи, можно поставить в соответствие следующую программу на Прологе.

Пример 6.1:

```
/*Вычисление факториала и чисел Фибоначчи методом нисходящей рекурсии*/
class predicates
        факт:(integer, real)nondeterm (i,o).
        фибоначчи:(integer, real)nondeterm(i,o) .
clauses
/* Нисходящая рекурсия – вычисление факториала */
        факт(0, 1):- !. /* начальные условия */
        факт(K, F):-
                /* текущее состояние на шаге K */
                K1 = K - 1, /*K1 - номер предыдущего шага */
                факт(K1, F1),
                /* F1 – предшествующее состояние */
                F = F1 * K.
                 /* определение текущего состояния */

/* Нисходящая рекурсия – вычисление чисел Фибоначчи */
        фибоначчи(0, 0):- !. /* начальные */
        фибоначчи(1, 1):-! . /* условия */
        фибоначчи(K, F):-
                K1 = K - 1,
                фибоначчи(K1, F1), /* F(K - 1) */
                K2 = K - 2,
                фибоначчи(K2, F2), /* F(K - 2) */
                F = F1 + F2. /* F(K) = F(K - 1) +F(K-2) */
clauses
        run():-
                console::init(),
                write("Введите число, факториал которого Вы хотите
                        получить"),
                K = read(), факториал(K, F),
                write("факториал(",K,")=",F), nl,
                факт(K, F1), write("факт(",K,")=",F1),nl,
                фибоначчи(10, Z), write("фибоначчи(", 10, ")=", Z), !.
        run().
```

Из примеров следует, что если процесс описан рекуррентными соотношениями типа (6.1), то формально переход к рекурсивному описанию на Прологе осуществляется достаточно просто.

6.2. Как писать рекурсивные определения

В первую очередь необходимо постараться описать процесс решения в виде рекуррентных соотношений (6.1). В них функция f может быть выражена в словесной форме. Затем разделить это описание на 2

части: собственно рекуррентные соотношения и соотношения, задающие граничные (начальные и конечные) условия процесса. Например, в задаче вычисления факториала соотношения (6.1) разбиваются на следующие 2 части:

Граничные условия	Рекуррентные соотношения
0! = 1 – начальные условия	*k! = (k – 1)!·k;*
k = N – условия окончания	*k=1, 2, 3,...*

Рекурсивное определение в программе также делится на две части. Одна из них соответствует рекуррентным формулам и состоит из правил, в теле которых присутствует целевое утверждение с тем же предикатом, что и заголовок правила, т.е. из правил, рекурсивно вызывающих самих себя. Другая часть содержит утверждения, описывающие **терминальную ситуацию**, т.е. ситуацию, в которой рекурсивное обращение предиката к самому себе прекращается. В качестве **терминальной** ситуации выбирается одно из граничных условий. **Обычно терминальная ситуация является первым утверждением** в определении. В этом случае вот что происходит при выполнении (см. пример 6.1):

- оценивается первое утверждение в рекурсивном определении;
- если первое утверждение не выполняется, осуществляется переход к следующему в определении утверждению, и оно оценивается. Обычно это правило, которое содержит условие, начинающее рекурсию;
- после прохождения первого уровня рекурсии выполнение возвращается к первому утверждению в определении и опять оценивается его истинность;
- если оценивание первого утверждения заканчивается неудачей, выполнение переходит ко второму в определении утверждению и входит во второй уровень рекурсии. Этот процесс продолжается до тех пор, пока первое утверждение (содержащее терминальную ситуацию) не выполнится и, таким образом, сделает определение успешным и остановит рекурсию.

В зависимости от того, как осуществляется переход от соотношений (6.1) к программе, различают два разных стиля рекурсивных определений: нисходящая рекурсия и восходящая рекурсия.

6.3. Нисходящая рекурсия

При нисходящей рекурсии описание процесса (6.1) начинается с конца, с момента N и номер шага постепенно уменьшается на 1.

```
факт(N, F):-  N1 = N-1,
              факт(N1, F1),
```

```
F = F1 * N.
```

В качестве терминальной ситуации выбираются начальные условия. Условия окончания используются как значения аргументов целевого утверждения-запроса: *факт(4, F).*

Все это хорошо прослеживается в примере 6.1, где использована техника нисходящей рекурсии. Термин "нисходящая" происходит от термина "нисходящее проектирование" и обозначает разбиение основной задачи на более простые подзадачи. Особенностью разбиения в рекурсивном определении является то, что подзадачи являются подобными (т.е. копиями) исходной задачи, а более простыми они оказываются за счет других исходных данных.

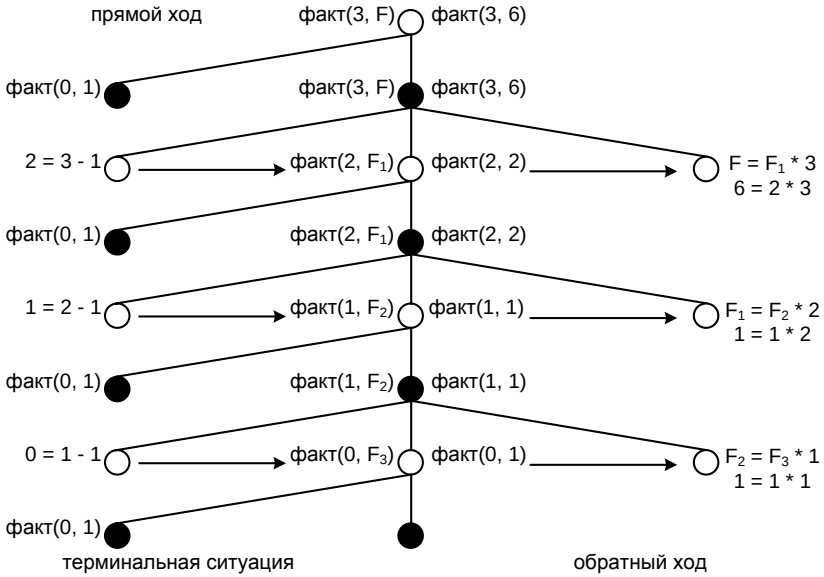


Рис. 6.1. И-ИЛИ-дерево для нисходящей рекурсии

Рассмотрим процесс нисходящей рекурсии для примера 6.1 (*F=N!*). Опишем его с помощью И-ИЛИ-дерева, представленного на рис. 6.1, для запроса *факт(3, F).*

Из анализа схемы следует, что процесс выполнения рекурсивного определения можно разбить на три части:

- **прямой ход** – с момента активизации целевого утверждения до момента достижения терминальной ситуации;

- **терминальная ситуация**, соответствующая одному из граничных условий;
- **обратный ход** – с момента наступления терминальной ситуации до момента достижения вершины доказательства.

При нисходящей рекурсии прямой ход характеризуется тем, что выполняемая цель при согласовании с БЗ порождает и вызывает к выполнению новые, точно такие же цели, но с другими аргументами. При этом не вычисляются никакие требуемые в задаче данные. В момент достижения терминальной ситуации результат решения задачи отсутствует. Он начинает строиться постепенно только при выполнении обратного хода (*F = F1 * N*), в процессе которого осуществляется передача значений аргументов доказанной цели в тело вызвавшего ее правила.

Окончательный результат оказывается построенным только при достижении вершины. В этот момент его значение приобретает один из аргументов целевого утверждения-запроса, что позволяет использовать результат в дальнейшей части программы.

К достоинствам нисходящей рекурсии следует отнести простоту перехода от рекуррентных соотношений к программе и минимальное число аргументов в определяемом предикате. Однако нисходящая рекурсия требует большого объема памяти для хранения всех копий вызываемого рекурсивно предиката и является неэффективной по времени.

6.4. Восходящая рекурсия

При восходящей рекурсии моделирование процесса начинается сначала, с момента *k = 0*. Номер шага постоянно возрастает: *k = k + 1*. В качестве терминальной ситуации выбирается момент достижения условий окончания процесса *k = N*, начальные условия задаются в качестве значений аргументов целевого утверждения-запроса. В восходящей рекурсии параметры, характеризующие состояние процесса, вычисляются на каждой стадии рекурсии в процессе выполнения прямого хода. Ниже приведено определение процесса вычисления факториала *F = 3!*, написанное с использованием техники **восходящей рекурсии**.

```
/* неправильное определение восходящей рекурсии */
факт1(3, P):- write(P), nl.
/* терминальная ситуация описывает условие
                        окончания вычислений*/
факт1(K, P):- K1 = K+1,
```

```
                P1 = P*K1,
                факт1(K1, P1).
/* P1 – текущее состояние процесса*/
run():-         факт1(0, 1),!. /* начальное состояние процесса */
run().
```

В этом определении *K* и *K1 = K+1* – номера соответственно текущего и следующего шагов; *P* – промежуточное значение факториала, вычисленное к моменту *K*; следующее значение *P1* определяется как *P1 = P * K1*.

Как видим, огромным недостатком определения *факт1* является зависимость описания терминальной ситуации от значений исходных данных. Например, для *F = 4!* первое утверждение должно иметь вид *факт1(4, F)*. Проанализируем вычисление восходящей рекурсии с помощью И-ИЛИ-дерева, представленного на рис. 6.2.

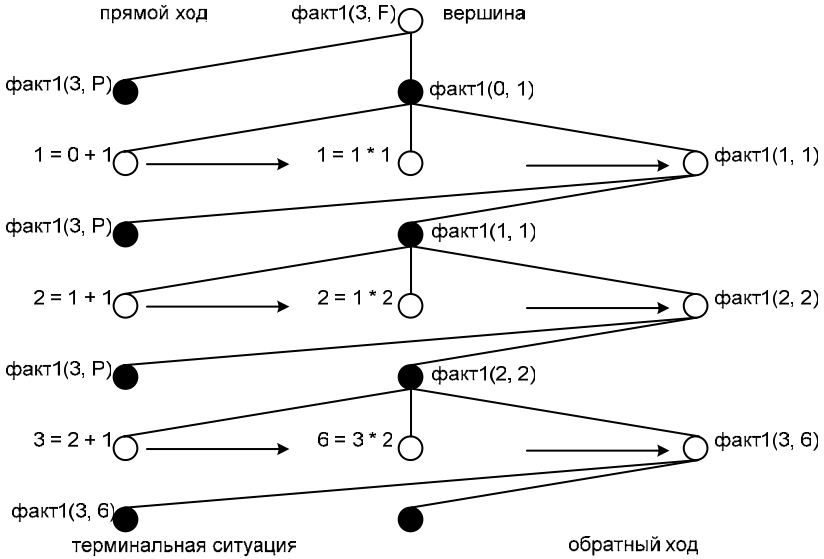


Рис. 6.2. И-ИЛИ-дерево для восходящей рекурсии

Из анализа схемы следует, что в процессе выполнения прямого хода результат строится постепенно (*P* хранит промежуточные значения) и окончательное значение приобретается в момент достижения **терминальной ситуации**. При обратном ходе результат теряется, так как восстанавливаются конкретизированные значения аргументов цели *факт1*. Таким образом, в вершине доказательства результат отсутст-

вует. Чтобы его не потерять, необходимо в момент достижения терминальной ситуации передать его значение переменной-результату, которую дополнительно необходимо включить в список аргументов. Избавиться от конкретики терминальной ситуации также можно введением еще одной переменной *N* в список аргументов, представляющей собой номер последнего шага процесса. С ее значением постоянно сравнивается номер текущего шага *K*, и при выполнении условия окончания процесса *K* = *N* достигается терминальная ситуация.

Таким образом, список аргументов предиката, проектируемого по схеме восходящей рекурсии, должен содержать две группы параметров: одна группа – это исходные данные и результат, вторая – промежуточные значения переменных процесса. Например:

факт1 (N, F, K, P),

где *N* – исходное данное, *F* – результат, *K* – текущий номер шага, *P* – текущее состояние.

Чтобы сохранить число аргументов таким же, как и при нисходящей рекурсии, следует определить два разных предиката, один из которых (основной) обращается к предикату с дополнительно введенными аргументами и построенному по схеме восходящей рекурсии. Например:

```
/* Правильное определение восходящей рекурсии */
class predicates
        факт1:(integer, real) nondeterm (i,o).
        факт1:(integer, real, integer, real) nondeterm (i,o,i,i).
clauses
        факт1(N, F):-       /* основной предикат */
                            факт1(N, F, 0, 1).
                            /* рекурсивно описанный предикат */
        факт1(N, F, N, F):-         !.
        /* терминальная ситуация */
        факт1(N, F, K, P):-         /* восходящая рекурсия */
                                    K1 = K + 1,
                                    P1 = P * K1,
        факт1(N, F, K1, P1).
clauses
        run():-   console::init(),
                  write("Введите число, факториал которого Вы
                          хотите получить "),
                  K = read(), факт1(K, F2),
                  write("факт1(",K,")=",F2),nl.
        run().
```

Достоинством восходящей рекурсии (рекурсивный вызов в конце тела правила) является экономия памяти. Необходимо хранить только параметры рекурсивно определенных целей *факт1*, а не все тело правила. Следствием этого является также эффективность восходящей рекурсии по быстродействию. Рекомендуется во всех случаях, когда возможно, переходить от нисходящей рекурсии к восходящей. Ниже приведен пример восходящей рекурсии для многошагового итерационного процесса на примере получения чисел Фибоначчи.

```
/*Получение чисел Фибоначчи методом восходящей рекурсии */
class predicates
        фибоначчи1:(integer, real) nondeterm(i,o).
        фибоначчи1:(integer, real, integer, real, real) nondeterm(i,o,i,i,i).
clauses
        фибоначчи1(N, F):-      фибоначчи1(N,F,1,1,0).
        /*начальные условия */
        фибоначчи1(N, F, N, F, _):- !.
        /* условия окончания - терминальная ситуация*/
        фибоначчи1(N, F, K, F1, F0):-
                K1 = K + 1,
                F2 = F0+F1, /* F(K) = F(K-2) + F(K-1) */
                фибоначчи1(N, F, K1, F2, F1).
clauses
        run():- write("Введите число, факториал которого Вы хотите
                        получить"),
                K = read(),факториал(K, F),
                фибоначчи1(10, Z1),
                write("фибоначчи1(",10,")=",Z1),nl,!.
        run().
```

Здесь N – порядковый номер элемента последовательности чисел Фибоначчи; F – число Фибоначчи.

Контрольные вопросы

1. Что такое рекурсивное определение?
2. Что такое итерационный процесс?
3. Как описывать рекурсивные определения?
4. Что такое терминальная ситуация?
5. Чем отличаются нисходящая и восходящая рекурсии?

Контрольные задания

Выполнить свой вариант задания тремя способами: методом итерации (гл. 5), с помощью техники нисходящей и восходящей рекурсий.

1. Даны a, n. Вычислить $y=a^n$.

2. Даны a, n. Вычислить $S=a+a(a+1)+...+a(a+1)...(a+n)$.

3. Даны a, n. Вычислить $S=a+a^2+a^3+...+a^n$.

4. Даны x, a, n. Вычислить $Z=((...((x+a)^2+a)^2+...+a^2)+a^2)+a$.

5. Пусть $x_0=2$, $q=3$, $r=2$, $x_k=q\,x_{k-1}+r$, $k=1, 2, ...$. Вычислить x_7.

6. Пусть $V_1=1$, $V_2=2$, $V_i=2\cdot V_{i-1}+3\cdot V_{i-2}$, $i=3,\ldots$. Дано n ($n/3$). Получить V_n.

7. Пусть $a_0=1$, $a_k=k\cdot a_{k-1}+1/k$, $k=1, 2,\ldots$. Дано n. Получить a_n.

8. Пусть $v_1=v_2=0$; $v_3=1.5$; $v_i=(i+1)\,v_{i-1}-v_{i-2}\cdot v_{i-3}$; $i=4,5,...$. Дано $n/4$. Получить v_n.

9. Пусть $x_0=c$; $x_1=d$; $x_k=q\,x_{k-1}+r\,x_{k-2}+b$; $k=2,3,....$. Даны q, r, b, c, d, $n/2$. Получить x_n.

10. Пусть $a_0=a_1=1$; $a_i=a_{i-2}+a_{i-1}\cdot 2^{i-1}$, $i=2,3,\ldots$. Найти произведение $P=\prod_{i=1}^{5} a_i$.

11. Пусть $x_1=y_1=1$; $x_i=3\cdot x_{i-1}$; $y_i=x_{i-1}+y_{i-1}$, $i=2,3,\ldots$. Дано n. Найти $S=\sum_{i=1}^{n}(x_i-y_i)^2$.

12. Пусть $a_1=b_1=1$; $a_k=(b_{k-1}+a_{k-1})$; $b_k=2\cdot a^2_{k-1}+b_{k-1}$, $k=2, 3, ...$. Дано n. Найти $S=\sum_{k=1}^{n} a_k b_k$.

13. Пусть $a_1=b_1=1$; $a_k=3\cdot b_{k-1}+2\cdot a_{k-1}$; $b_k=2\cdot a_{k-1}+b_{k-1}$, $k=2, 3,...$. Дано n. Найти $S=\sum_{k=1}^{n} a_k b_k$.

14. Пусть имеется следующая база данных:

ПЕТРОВ(ПРОГР(1), 10, 25, 45).
ПЕТРОВ ПРОГР(2), 15, 30, 20).
ПЕТРОВ(ПРОГР(3), 12, 27, 35).
ПЕТРОВ(ПРОГР(4), 20, 30, 30).

Эти факты отражают количество часов, затраченных программистом Петровым на составление алгоритма, программирование и отладку четырех разных программ: *ФАМИЛИЯ (<ПРОГР (НОМЕР)>, <АЛГОРИТМ>, <ПРОГРАММИРОВАНИЕ>, <ОТЛАДКА>)*. Определить суммарные затраты времени на работу с программами: с 1-й по 3-ю, со 2-й по 4-ю.

15. Для базы данных из предыдущего варианта определить затраты времени на отладку всех программ.

16. Вычислить $\begin{cases} x^2+2x, x>3 \\ \dfrac{x^2+3x+1}{2x}, x<3 \end{cases}$, x изменяется с шагом 1 от 0 до 10.

17. Написать датчик целых случайных чисел N, равномерно распределенных в интервале от 1 до R, и напечатать при R=10 последовательность первых случайных чисел до момента появления числа 5. Для описания датчика использовать следующие формулы $S_n=(125\cdot S_{n-1}+1) \bmod 4096$; $N=S_n \bmod R+1$; $S_0=13$; $R=10$; N – генерируемое случайное число.

18. Найти наименьший общий делитель НОД двух чисел по алгоритму Евклида. Если M является точным делителем N, то НОД=M, в противном случае нужно брать функцию НОД от остатка от деления N на M.

19. Даны функции Бесселя 1-го и 2-го порядков J_1=0, J_2=2. Вычислить J_q по формуле $J_{n+1}=2\,n\,J_n-J_{n-1}$.

20. Последовательность многочленов $H_0(x)=1$, $H_1(x)=x$, определяется следующим образом: $H_k(x)=x\,H_{k-1}(x)+(k-1)\cdot H_{k-2}(x)$, k=2, 3, ... Получить $H_5(5)$.

21. У прилавка в магазине выстроилась очередь из n покупателей. Время обслуживания продавцом i-го покупателя равно t_1 (мин) (i=1,n). Пусть даны натуральное n и целые t_1, t_2, ..., t_n. Получить C_1, C_2, ... C_n, где C_i – время пребывания i-го покупателя в очереди (i=1,…,n). Указать номер покупателя, для обслуживания которого продавцу потребовалось самое малое время.

22. В некоторых видах спортивных состязаний выступление каждого спортсмена независимо оценивается несколькими судьями, затем из всей совокупности оценок удаляются наиболее высокая и наиболее низкая, а для оставшихся оценок вычисляется среднее арифметическое, которое и идет в зачет спортсмену. Считая, что числа a_1, a_2, ..., a_n – это оценки, выставленные судьями одному из участников соревнований, определить оценку, которая пойдет в зачет этому спортсмену.

23. Имеется горсть из N монет C_1, C_2, ..., C_n различного достоинства. Определить, можно ли на эти деньги купить товар стоимостью в C копеек и сколько монет каждого достоинства присутствует в горсти.

24. Даны натуральное число n, целые числа a_1, a_2, ..., a_n. Получить: $\max(a_1,a_2,...,a_n)$; $\min(a_1,a_2,...,a_n)$.

Глава 7. ОТСЕЧЕНИЕ

7.1. Введение в отсечение

Изучая алгоритм работы интерпретатора Пролога, вы познакомились с тем, как работает механизм возврата (бектрекинг) в Прологе. Процессом возврата можно управлять. Для этого предназначен системный предикат отсечение (!). Он позволяет сокращать пространство поиска решений при возврате. Синтаксически использование отсечения в правиле выглядит как вхождение целевого утверждения с предикатом "!", не имеющим аргументов. Например:

```
значение(X, Y, Z, V):-
        калории(X, E),
        калории(Y, P),
        калории(Z, D),
        !, /* Знак отсечения */
        V = E + P + D.
```

Как целевое утверждение этот предикат "!" всегда согласуется с базой знаний и не может быть вновь согласован.

7.2. Воздействие отсечения на процесс выполнения

При прямом ходе доказательства некоторой цели отсечение выступает как истинное условие в теле правила и не оказывает никакого воздействия на процесс выполнения. Его назначение реализуется только при инициировании процесса возврата вследствие ложности одного из предикатов, расположенных после (справа или ниже) отсечения в теле правила. В этом случае, т.е. при прохождении отсечения в обратном направлении (справа налево), оно как бы объявляет, что **цель**, вызвавшая выполнение отсечения (**родительская цель**), не имеет других вариантов согласования, а потому процесс возврата осуществляется к предшествующей (расположенной левее от родительской) цели.

Другими словами, действие отсечения при возврате сводится к следующим моментам:

- отсечение выбрасывает из рассмотрения все утверждения, расположенные после предложения, в котором находится отсечение;
- отсечение отбрасывает все альтернативные решения конъюнкции целей, расположенных в утверждении левее отсечения, т.е. конъюнкция целей, стоящих перед отсечением, приводит не более чем к одному решению;

- отсечение не влияет на цели, расположенные правее его. В случае возврата они могут порождать более одного решения.

Таким образом, отсечение при возврате сокращает пространство поиска вариантов доказательства.

7.3. Использование отсечения

Отсечение применяется для устранения бесконечных циклов, при программировании взаимоисключающих утверждений и при необходимости неудачного завершения доказательства цели. Рассмотрим все три случая на примерах.

Пример 7.1. Устранение бесконечных циклов. Вновь обратимся к утверждениям, определяющим вычисление факториала (см. гл. 6 "Рекурсия").

```
факт(0, 1).
факт(K, F): -  K1 = K - 1,
               факт(K1, F1),
               F = F1 * K.
```

Введем запрос:

```
факт(0, X), write("X=", X), nl, fail.
```

Получим *X = 1*, и процесс зациклится.

При возникновении возврата Пролог сделает попытку сопоставить *факт(0, X)* со вторым утверждением *факт(K,F):-….* Сопоставление успешно, и теперь делается попытка доказать цель *факт(-1, F1)*, что в свою очередь приводит к цели *факт(-2, F2)* и так далее, т.е. образуется бесконечный цикл.

К счастью, можно устранить такие ситуации, используя отсечение, при этом, указывая Прологу, что не существует других решений в случае успешного согласования граничного условия.

```
факт(0, 1):- !.
факт(K, F):-   K1 = K - 1,
               факт(K1, F1),
               F = F1 * K.
```

Поставив в первое утверждение знак отсечения и задав вопрос в версии 5.2:

```
goal
      факт(0, X), write("X=", X), nl, fail.
```

или в версии 7.0-7.3:

```
run():-   факт(0, F),
          write("факт(",0,")=",F), nl,!.
run().
```

получим единственное решение:

```
X = 1.
1 Solution
```

Пример 7.2. **Программирование взаимоисключающих утверждений**. При оценке роста человека чаще всего пользуются не числовыми данными, а качественными определениями, например "Человек высокого роста" или "Он был ниже среднего роста". Можно связать эти качественные оценки с числовыми данными (Н – высота человека, см), например, следующим образом:

$$\text{РОСТ} = \begin{cases} \text{ОЧЕНЬ ВЫСОКИЙ, если } Н > 200 \\ \text{ВЫСОКИЙ, если } 170 < Н \le 200 \\ \text{ВЫШЕ СРЕДНЕГО, если } 160 < Н \le 170 \\ \text{СРЕДНИЙ, если } 150 < Н \le 160 \\ \text{НИЖЕ СРЕДНЕГО, если } 140 < Н \le 150 \\ \text{МАЛЕНЬКИЙ, если } Н < 140 \end{cases}$$

Тогда описать классификацию людей по росту можно с помощью следующих утверждений языка Пролог:

```
/*Классификация людей по росту */
domains
    rost = integer.
    harakter = string.
class predicates
    рост:(rost, harakter) nondeterm (i,o) (i,i).
    рост1:(rost, harakter) nondeterm (i,o) (i,i).
clauses
/* первый вариант программы */
    рост(Н, "ОЧЕНЬ ВЫСОКИЙ"):- Н > 200.
    рост(Н, "ВЫСОКИЙ"):- Н > 170, Н <= 200.
    рост(Н, "ВЫШЕ СРЕДНЕГО"):- Н > 160, Н <= 170.
    рост(Н, "СРЕДНИЙ"):- Н > 150, Н <= 160.
    рост(Н, "НИЖЕ СРЕДНЕГО"):- Н > 140, Н <= 150.
    рост(Н, "МАЛЕНЬКИЙ"):- Н < 140.
/* второй вариант программы */
    рост1(Н, "ОЧЕНЬ ВЫСОКИЙ"):- Н > 200, !.
    рост1(Н, "ВЫСОКИЙ"):- Н > 170, !.
    рост1(Н, "ВЫШЕ СРЕДНЕГО"):- Н > 160, !.
```

```
    рост1(Н, "СРЕДНИЙ"):- Н > 150, !.
    рост1(Н, "НИЖЕ СРЕДНЕГО"):- Н > 140, !.
    рост1(Н, "МАЛЕНЬКИЙ").

clauses
    run():-   console::init(),
              рост(150, Н),
              рост1(150, Н1),
              write(Н," ",Н1),nl,fail.
    run().
```

В первом варианте программы тело каждого правила содержит все условия, описывающие человека данной категории, во втором варианте все правила, кроме последнего, содержат знаки отсечения "!".

В первом варианте программы при возврате независимо от уже найденного единственно верного решения вновь будут безуспешно просматриваться все остальные, лежащие ниже решения. На это затрачивается много времени. Во второй программе благодаря отсечению просмотр остальных решений производиться не будет. Рассмотрим, что произойдет, если знаки отсечений удалить из утверждений. Тогда в ответ на запрос *рост1(205, Н1)*будет получено несколько решений, из которых только первое верно, а остальные ошибочны.

```
X = ОЧЕНЬ ВЫСОКИЙ
X = ВЫСОКИЙ
X = ВЫШЕ СРЕДНЕГО
X = СРЕДНИЙ
X = НИЖЕ СРЕДНЕГО
X = МАЛЕНЬКИЙ
6 Solution
```

Таким образом, в данной программе отсечение несет двойную функцию: благодаря сокращению пространства поиска уменьшает время решения и, **главное**, отсекает неправильные варианты ответов на запрос, позволяя получить единственно правильное решение.

7.4. Ловушки отсечения

При использовании отсечения возможно возникновение двух видов ловушек:

- отсечение исключает необходимые альтернативы; программа лишается свойства быть генератором новых знаний;
- отсечение разрушает декларативное восприятие программы.

В качестве примера ловушки первого типа рассмотрим применение предиката *между(X, Y, Z)* в двух различных вариантах: для генерации

всех целых чисел *X*, принадлежащих интервалу [*Y*, *Z*], и для проверки, находится ли целое число *X* в интервале [*Y*, *Z*], т.е. $Y \le X \le Z$.

```
clauses
    между(X, X, Z):-  X <= Z.
    между(X, Y, Z):-  Y1 = Y + 1,
                      Y1 <= Z,
                      между(X, Y1, Z).
```

При запросе:

```
между(X, 5, 8).
```

будут получены следующие решения

```
X = 5 X = 6 X = 7 X = 8
```

что показывает, что предикат *между* в этом случае выполняет роль генератора целых чисел в заданном интервале.

Введем в целях повышения эффективности программы отсечение в граничное условие предиката *между*:

```
clauses
    между1(X, X, Z):-       !, X =< Z.
    между1(X, Y, Z):-       Y1 = Y + 1 ,Y1 <= Z,
                            между1(X,Y1,Z).
```

и попытаемся его использовать для контроля, принадлежит ли *X* заданному интервалу [*Y*, *Z*]:

```
X = 3, между1(X, 2, 4).
yes
```

Как только будет достигнут момент согласования граничного условия *между(3, 3, 4)*, бесцельный поиск других решений при любом способе инициирования возврата благодаря отсечению производиться не будет, время доказательства цели сократится. Однако в таком виде этот предикат уже не сможет выступать в роли генератора. Сравнение двух вариантов программы в примере 7.2 иллюстрирует ту ситуацию, когда введение отсечения приводит к потере декларативности определения отношения.

Ниже приводится пример, как нужно использовать отсечение, чтобы сохранилась способность предиката быть генератором, и в тоже время сократить бессмысленный поиск заведомо не существующих решений.

```
/* Программа позволяет определять возраст студента и находить студентов по заданному возрасту*/
class predicates
```

```
        возраст:(string, integer)nondeterm anyflow.
        возраст_студента:(string, integer)nondeterm anyflow.
    clauses
        возраст("Иванов", 18).
        возраст("Петров", 17).
        возраст("Сухов", 18).
        возраст_студента(Fio, Voz):-    bound(Fio),
                                        возраст(Fio, Voz), !.
        возраст_студента(Fio, Voz):-    free(Fio),
                                        возраст(Fio, Voz).
    clauses
        run():-  console::init(),
                 возраст("Петров", V), write("Петрову ",V, " лет"), nl,
                 возраст(Fio, 18),write(Fio, " 18 лет"), nl, fail.
        run().
```

В этой программе встроенные предикаты *bound(Fio), free(Fio)* используются для того, чтобы выделить две ситуации, первая – когда переменная *Fio* имеет конкретное значение (*bound)*, в этом случае нужно найти один возраст и прекратить поиск, вторая – когда переменная *Fio* свободная, т.е. неконкретизированная, тогда выдаются все решения, согласующиеся с базой фактов.

Применять отсечение следует осторожно, соразмеряя выгоду от повышения эффективности программы с опасностью возникновения нежелательных эффектов.

Контрольные вопросы

1. Для чего предназначен системный предикат отсечение ("!")?
2. Как ведет себя предикат отсечение при прямом ходе доказательства некоторой цели?
3. Как ведет себя предикат отсечение при инициировании возврата?
4. Как используется отсечение для устранения бесконечных циклов?
5. Как программируются взаимоисключающие утверждения?
6. Что необходимо учитывать при использовании отсечения?
7. Какие "ловушки" отсечения Вы знаете?
8. Приведите примеры использования отсечения.

Контрольные задания

Пересмотреть решения задач, полученные в гл. 2, 4, 5, 6, и там, где это необходимо, ввести знак отсечения. Исследовать поведение программы при наличии и отсутствии отсечения, объяснить необходимость его использования в каждом случае.

Глава 8. СПИСКИ

8.1. Введение в списки

Список – это упорядоченная совокупность произвольного числа элементов. Порядок расположения элементов в последовательности является существенным. Элементами списка могут быть любые термы – константы, переменные, списки, структуры, которые включают, конечно, и другие списки. Списки позволяют представить практически любой тип структуры данных, который может потребоваться при обработке символьной информации. Списки широко используются для представления деревьев и синтаксического разбора, грамматик, карт городов, программ для ЭВМ и математических объектов, таких как графы, формулы и функции.

Одна из форм представления списков на Прологе – это скобочная форма записи списка. Она представляет собой заключенную в квадратные скобки последовательность элементов списка, разделенных запятыми. Например:

[*B*] – список, состоящий из одного элемента *B*;

[*A*, *B*, *F*] – список, состоящий из 3-х элементов *A*, *B*, *F*;

["этот", "человек", "любит", "ловить", "рыбу"] – здесь элементы списка образуют предложение.

[*a*, *V*1, *b*, *X*, *Y*] –элементами списка могут быть и переменные.

Преимущество использования списков – в более естественном и компактном представлении информации. Например, записать факт, что студент Иванов во втором семестре изучает дисциплины: математику, физику, программирование, историю, иностранный язык, можно следующим образом:

```
изучает("Иванов", ["математика", "физика", "программирование",
      "история", "ин_яз"]).
```

Поведение переменных, входящих в списки, ничем не отличается от поведения переменных как аргументов предиката. Первоначально они имеют неконкретизированное значение, но в любой момент могут быть конкретизированы.

8.2. Описание списков в языке Visual Prolog

В языке Visual Prolog все данные и предикаты должны быть описаны. Это требование накладывает определенные ограничения на использование списков с элементами разной природы, разного типа. Если список включает элементы разного типа, в том числе подсписки, то

они рассматриваются как структуры, что значительно усложняет описание операций и представление элементов списка. Поэтому в языке Visual Prolog ***списки*** чаще всего рассматриваются как ***совокупности однотипных элементов данных***. Вначале именно такие списки мы и будем рассматривать, затем в п. "Составные списки" будут изложены правила работы со списками с произвольным типом элементов.

Списки описываются в разделе ***domains*** следующим образом:

```
domains
        elementlist = elements*
        elements = <тип элемента>
domains
        integerlist = integer* /*Список целых чисел*/
        список_строк = string* /* Список строк*/
domains
        фио = string
        группа = integer
        студент = студент(фио, группа)
        список_студентов = студент*
        /* Список структур типа студент */
```

8.3. Голова и хвост списка

Работа со списками основана на расщеплении их на голову и хвост списка. Головой списка является первый его элемент. Хвост списка представляет собой **список**, состоящий из всех элементов исходного списка, за исключением первого его элемента (рис. 8.1).

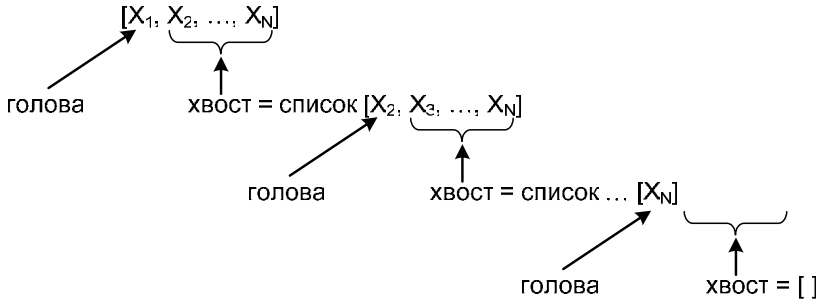


Рис. 8.1. Расщепление списка на голову и хвост

Список имеет рекурсивную структуру, т.е. определяется через самого себя, а именно: **список** – это любой **пустой** список [ ], не содержащий ни одного элемента, либо структура, имеющая два компонента [*X*|*T*] – голову *X* и хвост *T* списка. Голова *X* – это первый элемент спи-

ска, а хвост *T* – список из оставшихся элементов. Конец списка обычно представляют как хвост, который является пустым списком. Пустой список не имеет ни головы, ни хвоста. Следующие примеры демонстрируют расщепление списков на голову и хвост:

Список	Голова	Хвост
[a, b, c]	a	[b, c]
[a]	a	[ ]
[]	нет головы	нет хвоста

Так как операция расщепления списка на голову и хвост очень широко используется, то в Прологе введена специальная форма для представления непустого списка с головой *X* и хвостом *Y*. Это записывается как [*X*|*Y*], где для разделения *X* и *Y* используется вертикальная черта. При конкретизации такой структуры *X* сопоставляется с головой списка, а *Y* – с хвостом списка.

8.4. Унификация списков как аргументов предикатов

Одной из фаз процесса доказательства некоторого целевого предиката является унификация его аргументов с аргументами утверждений базы знаний. Как осуществляется унификация в том случае, когда аргументами цели являются списки? Чтобы пояснить этот процесс, введем понятие шаблона (образца) списка. Шаблон (образец) списка – это форма описания множества (семейства) списков, обладающих вполне определенными свойствами. Так, например, шаблон списка [*X*|*Y*] описывает любой произвольный список, состоящий не менее чем из одного элемента; шаблон [*X1*, *X2*|*Y*] – список, состоящий не менее чем из двух элементов; аналогично [*X1*, *X2*, *X3*|*Y*] – список, содержащий не менее трех элементов; а шаблон в виде переменной *Z* – любой список, в том числе и пустой. Шаблон может содержать как переменные, так и константы. Например, шаблон [*b*|*Z*] задает любой список, первым элементом которого является элемент *b*.

При унификации происходит сопоставление шаблонов. Если шаблоны целевого утверждения и утверждения базы знаний представляют списки с несовместимыми различными свойствами (разные классы списков), то унификация заканчивается неудачей. Так, например, нельзя сопоставить списки:

[*X1*, *X2*|*T*] [*a*],
[*X1*, *X2*, *X3*|*Z*] [1, 2]
и т.д.

Если шаблоны не противоречат друг другу, то осуществляется конкретизация отдельных переменных шаблона, т.е. присвоение им значений соответствующих констант, или сцепление с соответствующими

переменными другого шаблона. В результате оба шаблона должны стать идентичными и породить общее решение – новый шаблон. Примеры: при сопоставлении шаблона [*X*, *Y*|*Z*] со списком [*a*, *b*, *c*] унификация проходит успешно, и переменные принимают следующие значения: *X* = *a*, *Y* = *b*, *Z*=[*c*].

Если в процессе сопоставления и присвоения значений шаблоны не могут стать идентичными, то унификация заканчивается неудачей, как в следующем случае:

["бал", *Y*, *Y*] [*X*, *X*, "цех"],

где сопоставление показывает, что элементы обоих списков должны быть одинаковы, однако "бал" не равно "цех".

8.5. Принадлежность элементов списку

Предположим, что имеется некоторый список, например, список студентов:

["Иванов", "Петров", "Агеев", "Левина", "Лукашов", "Ленских"]

и мы хотим определить, имеется ли некоторый студент, например *Агеев*, в этом списке. В Прологе это можно сделать, определив отношение принадлежности объекта некоторому списку с помощью предиката *принадлежит(X, L)*. Целевое утверждение *принадлежит(X, L)* является истинным, если терм, связанный с *X*, является элементом списка *L*. Чтобы описать этот предикат, рассмотрим понятие "является элементом списка", суть которого раскрывается с помощью следующего определения: "Некоторый объект является элементом списка, если он:

- либо совпадает с головой списка;
- либо является элементом хвоста списка".

Из определения следует, что необходимы два правила для описания предиката *принадлежит*. Первое говорит о том, что объект *X* будет элементом списка *L*, если *X* совпадает с головой списка *L*. На Прологе этот факт записывается следующим образом: *принадлежит(X, [X|_])*. Здесь использована анонимная переменная "_" для обозначения хвоста списка, т.к. хвост списка в этом частном факте никак не используется, следовательно, его содержание безразлично.

Второе правило говорит о том, что X принадлежит списку также при условии, что он является элементом хвоста списка *T*, т.е. принадлежит хвосту списка *T*. Эта информация может быть выражена с помощью следующего рекурсивного правила:

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

Анонимная переменная "_", обозначающая голову списка, свидетельствует о том, что информация о голове списка не имеет никакого

значения для выбранного пути решения. Два этих правила в совокупности определяют предикат для отношения принадлежности и указывают, каким образом **просматривать список от начала до конца** при поиске некоторого элемента в списке.

Наиболее важный момент, о котором следует помнить, встретившись с рекурсивно определенным предикатом, заключается в том, что прежде всего надо найти граничные условия и способ использования рекурсии. Для предиката *принадлежит* имеются два типа граничных условий. Либо объект, который требуется найти, содержится в списке, либо не содержится. Первое граничное условие распознается первым утверждением, которое приведет к прекращению поиска в списке. Второе граничное условие встречается, когда второй аргумент предиката *принадлежит* является пустым списком.

Каждый раз, когда при поиске соответствия для целевого предиката *принадлежит* происходит рекурсивное обращение к тому же предикату, новая цель формируется для более короткого списка (см. рис. 8.1).

Очевидно, что рано или поздно произойдет одно из двух событий: либо произойдет сопоставление с первым правилом для *принадлежит*, либо в качестве второго аргумента *принадлежит* будет задан пустой список. Как только возникнет одна из этих ситуаций, прекратится рекуррентное порождение новых подцелей. Второе граничное условие не распознается ни одним из утверждений для *принадлежит*, так что процесс поиска сопоставимого элемента списка для целевого утверждения *принадлежит* закончится неудачей. Это демонстрирует следующий пример на Прологе:

```
domains
        integerList = integer*.
        stringList = string*.
class predicates
        принадлежит:(integer, integerList)nondeterm anyflow.
        принадлежит:(string, stringList)nondeterm anyflow.
clauses
        принадлежит(X, [X|_]).
        принадлежит(X, [_|Y]):- принадлежит(X, Y).
clauses
        run():-    console::init(),
                   принадлежит(7, [1, 2, 5]),
                   write("YES"),nl, !.
        run():-write("NO"),nl.
```

Достоинством предиката *принадлежит* является то, что он показывает, как с помощью рекурсивного определения получить доступ к

каждому элементу списка. Предикат *принадлежит* может быть использован в следующих интерпретациях:

- найти элемент *X* в заданном непустом списке *Y*: *принадлежит(2, [1, 2, 5])*;
- проверить, есть ли элемент *X* в заданном непустом списке *Y*: *принадлежит(a5, [a1, a2, a5])*;
- просмотреть последовательно и выяснить, какие элементы входят в список. В третьем случае для выдачи на экран всех элементов списка можно использовать следующую программу:

```
элементы_списка(L):- принадлежит(X, L),
                     write(X), nl, fail.
элементы_списка(_).
```

Для ввода и вывода списков могут быть использованы следующие два способа.

8.6. Ввод, вывод списка как терма.

При первом способе ввод/вывод списка осуществляется операторами *readterm(<тип списка>, X), write(X)*, при этом список *X* рассматривается как один терм. Так, например, программа

```
domains
    integerlist = integer*
clauses
    run():-  console::init(),
             write("Введите список "),
             hasdomain(integerList, X),
             /*Элемент X имеет тип integerList*/
             X = read(),
             write("Список L= "), write(X), nl ,!.
    run().
```

будет выполняться следующим образом:

```
Введите список [1, 4, 72, 0]
Список L= [1, 4, 72, 0]
```

Второй способ – поэлементный ввод/вывод списка может быть организован с помощью рекурсивно определенных предикатов:

```
/*Предикаты поэлементного ввода/вывода списков*/
domains
    integerlist = integer*.
    stringlist = string*.
```

```
class predicates
        ввод_списка:(integerlist) determ (o).
        ввод_спискаS:(stringlist) determ (o).
        вывод_списка:(integerlist)determ.
        вывод_списка:(stringlist)determ.
clauses
        ввод_списка([X|T]):-        write(" Введите элемент "),
                                    X = read(), X>=0, ! ,
        ввод_списка(T).
        ввод_списка([]).

        ввод_спискаS([X|T]):-       write(" Введите элемент "),
                                    X = readLine(), X <> "kon", ! ,
        ввод_спискаS(T).
        ввод_спискаS([]).

        вывод_списка([]):-          nl.
        вывод_списка([H|T]):-       write(H," "),
        вывод_списка(T).
clauses
        run():-   console::init(),
                  write("Введите список "),
                  ввод_спискаS(L), nl, write(" Список L" ),nl,
                  вывод_списка(L) ,!.
run().
```

Здесь *"kon"* – это терм, свидетельствующий о достижении конца списка.

8.7. Использование предиката присоединить

Рассмотрим простой и очень полезный предикат *присоединить*. Его основное назначение – соединить два списка в один. Согласование с базой знаний целевого утверждения *присоединить(X, Y, Z)* должно заканчиваться удачей в том случае, когда *Z* представляет собой соединение двух списков *X* и *Y*. Например, следующее целевое утверждение является истинным:

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

а целевое утверждение

```
присоединить([5, 4, 3], [3, 2, 1], Z), write(Z).
```

позволяет получить новый список *Z* = [5, 4, 3, 3, 2, 1] в результате объединения двух исходных списков [5, 4, 3] и [3, 2, 1].

Кроме основного назначения существует еще множество других применений этого предиката, например, таких как разделение списка всеми возможными способами, удаление начального или остаточного сегмента списка, разделение списка по заданному элементу.

Чтобы разработать предикат для некоторого отношения, нужно составить утверждения базы знаний, совокупность которых полностью описывает это отношение. На практике при определении отношения над списками нужно поступать следующим образом: выбрав один из аргументов этого отношения, составить высказывания относительно различных вариантов этого аргумента. Эти варианты должны покрывать все различные виды списков, которые могут появляться в качестве данного аргумента этого отношения.

Для отношения *присоединить(X, Y, Z)* выберем первый аргумент *X*. Полностью определим это отношение, задав одно утверждение для случая, когда *X* пустой список [ ], и второе – для всех случаев, когда X непустой список, представленный шаблоном [*X1*|*L1*].

Когда *X* = [ ], *Y* и *Z* всегда совпадают. Любой список, присоединенный к пустому списку, дает тот же самый список. Это можно представить с помощью следующего факта.

```
присоединить([], Y, Y).                                    (8.1)
```

Смысл второго утверждения:

```
присоединить([X1|L1], Y, [X1|L3]):-
    присоединить(L1, Y, L3).                               (8.2)
```

можно описать словами следующим образом.

Первый элемент *X1* первого списка *X* всегда будет и первым элементом третьего списка, т.е. если *X* = [*X1*|*L1*], то *Z* = [*X1*|*L3*].

Хвост *L3* третьего аргумента *Z* всегда будет представлять результат присоединения второго аргумента *Y* к хвосту *L1* первого списка *X*. Для выполнения этой операции присоединения *Y* к *L1* необходимо использовать предикат *присоединить*.

Так как при каждом обращении к правилу удаляется голова списка, являющегося первым аргументом, то постепенно этот список будет исчерпан и станет пустым, так что произойдет выход на первое граничное условие.

Таким образом, программа

```
class predicates
    присоединить:(integerList, integerList, integerList)
            nondeterm anyflow.
```

```
        присоединить:(stringList, stringList, stringList) nondeterm anyflow.
clauses
        присоединить([], Y, Y).
        присоединить([X1|L1], Y, [X1|L3]):- присоединить(L1, Y, L3).
clauses
        run():-   console::init(),
                  присоединить(["r", "y", "t"], ["u", "w"], L),
                  write(" Список L" ),nl,
                  вывод_списка(L) ,!.
        run().
```

полностью определяет отношение *присоединить*.

8.8. Использование предиката присоединить для разделения списка

Предикат *присоединить* можно использовать для разделения списка на два подсписка, если первые два аргумента объявить как переменные, а третий аргумент конкретизировать заданным списком:

На вопрос

```
присоединить(X, Y, [2, 3, 4]).
```

получим ответы:

```
X=[] Y=[2, 3, 4]
X=[2] Y=[3, 4]
X=[2, 3] Y=[4]
X=[2, 3, 4] Y=[]
4 Solution
```

Если требуется получить только непустые списки *X, Y*, то надо написать целевое утверждение *присоединить* в следующем виде: *присоединить([X1|T], [Y1|F], [2, 3, 4])*.

Аналогично с помощью одного и того же предиката *присоединить*, но с разными шаблонами аргументов можно задавать способы разделения списка на подсписки. Рассмотрим шуточную программу *мутанты*. С помощью этой программы образуются названия гибридов различных животных. Животные задаются их названиями в форме списка букв. Два животных производят на свет мутанта, если окончание названия первого из них совпадает с началом названия второго. В этой программе отношение *присоединить* применяется двумя способами: в одном случае для разделения списка, в другом – для объединения двух списков.

```
/* МУТАНТЫ */
domains
        charList = char*.
class predicates
        присоединить:(charList, charList, charList) nondeterm anyflow.
        мутант:(charList) nondeterm anyflow.
        красивый_мутант:() nondeterm.
        животное:(charList) nondeterm anyflow.
        вывод_сп:(charList) nondeterm anyflow.
clauses
        присоединить([], Y, Y).
        присоединить([X1|L1], Y, [X1|L3]):-
                присоединить(L1, Y, L3).
        мутант(Z):-     животное(X), животное(Y),
                        присоединить(A, [B|T], X),
                        присоединить([B|T], C, Y),
                        присоединить(X, C, Z).
        красивый_мутант:-       мутант(Z), вывод_сп(Z), fail.
        красивый_мутант:-       nl.
        вывод_сп([]):-          nl, !.
        вывод_сп([X|T]):-       write(X), вывод_сп(T).
        животное(['к','р','о','к','о','д','и','л']).
        животное(['ч','е','р','е','п','а','х','а']).
        животное(['к','а','р','и','б','у']).
        животное(['л','о','ш','а','д','ь']).
        животное(['х','а','м','е','л','е','о','н']).
        животное(['б','у','й','в','о','л']).
        животное(['в','о','л','к']).
        run():- console::init(),
                красивый_мутант(),!.
        run().
```

Получим следующий результат:

```
крокодил
крокодилошадь
черепаха
черепахамелеон
карибу
карибуйвол
лошадь
хамелеон
буйволошадь
буйвол
буйволк
волкрокодил
```

```
волкарибу
волк
```

8.9. Операции со списками

Так как список – это рекурсивная структура данных, то все определения операций со списками имеют рекурсивную природу. Операции по сложности можно разделить на три группы. Самая простая группа – это когда список задан и надо найти его характеристики (сумму элементов, максимальный или минимальный элемент и т.п.), вторая по сложности группа – это преобразование списка по заданному правилу, третья – записать в список элементы, получаемые либо от генератора, либо по определенному закону. Рассмотрим примеры для каждой группы.

Пример 8.1. Дан список оценок студента за все время обучения в ВУЗе, найти средний балл студента.

```
/* Программа решения примера 8.1 по спискам*/
domains
    ocenka = integer.
    list_ocenka = ocenka*.
class predicates
    средний_балл:(list_ocenka, real) determ (i,o).
    сумма_кол:(list_ocenka,ocenka, integer) determ (i,o,o).
clauses
    /* Нисходящая рекурсия */
    средний_балл(OcList, Sred_ball):- сумма_кол(OcList, S, N),
                                      Sred_ball = S / N.
    сумма_кол([], 0, 0).
    сумма_кол([Oc|List],S, N):-       сумма_кол(List, S1, N1),
                                      S = S1 + Oc, N = N1 +1.
    run():-  console::init(),
             L = [5,4,5,3,4,4,5,5,3,5],
             средний_балл(L,Sred_ball),
             write("Список оценок ", L, "Средний балл=" , Sred_ball),
             nl,!.
    run().
```

В результате выполнения программы получим:

```
Список оценок [5,4,5,3,4,4,5,5,3,5] Средний балл 4.3
```

Пример 8.2. Дан список студентов группы. Исключить из него студентов, не сдавших сессию.

```
/* Программа решения примера 8.2 */
```

```
domains
        студент = string.
        список_студентов = студент*.
class predicates
        исключить:(список_студентов, список_студентов,
                список_студентов) nondeterm anyflow.
        исключить1:(студент, список_студентов,
                список_студентов) nondeterm anyflow.
clauses
        исключить(RezSp, [], RezSp).
        исключить(SpGr,[Stud|Udal],RezSp):-
                исключить1(Stud,SpGr,RezSp1),
                исключить(RezSp1, Udal, RezSp).
        исключить1(Stud, [Stud|Gr], Gr):-!.
        исключить1(Stud, [Y|Gr], [Y|Gr1]):-
                исключить1(Stud, Gr, Gr).
        run():-
                console::init(),
                Gr =["Иванов", "Петров", "Лукьяненко"
                        "Самсонов", "Проценко"],
                Ot = ["Лукьяненко", "Проценко"],
                исключить(Gr,Ot, RezSp),
                write("Исходный список \n ",Gr,
                        "\n Список отчисленных\n ",Ot,
                        "\nСписок оставшихся ", RezSp),nl,!.
        run().
```

Пример 8.3. Записать в список все целые числа от 1 до *N*.

```
/* Программа решения примера 8.3 */
domains
        integerList = integer*.
class predicates
        список_целых:(integer, integerList)nondeterm anyflow.
        список_целых:(integer, integerList, integer, integerList)
                nondeterm anyflow.
        список_целых1:(integer, integerList) nondeterm anyflow.
        присоединить:(integerList, integerList, integerList)
                nondeterm anyflow.
clauses
        присоединить([], Y, Y).
        присоединить([X1|L1], Y, [X1|L3]):-
                присоединить(L1, Y, L3).
        список_целых(N, IntList):- список_целых(N, IntList, 1, [1]).
        /* Восходящая рекурсия - прямой порядок */
        список_целых(N, IntList, N, IntList):- !.
```

```
список_целых(N, IntList, N1, List1):-
        N2 = N1+1,
        присоединить(List1, [N2], List2),
        список_целых(N, IntList, N2, List2).
/* Нисходящая рекурсия - обратный порядок*/
список_целых1(1, [1]):- !.
список_целых1(N, [N|List]) :-
        N1 = N - 1,
        список_целых1(N1, List).
run():-  список_целых(9, IntList), write(IntList), nl,
        список_целых1(9, IntList1),write(IntList1), nl,!.
run().
```

Пример 8.4. Собрать необходимые данные из фактов базы знаний в список. Решить эту задачу можно с помощью встроенного предиката ***findall*.** Цель ***findall(X, P, List)*** порождает список ***List*** всех объектов *X*, удовлетворяющих цели *P*. Если ***findall(X, P, List)*** не находит ни одного решения для *P*, то цель ***findall*** просто терпит неуспех. Если один и тот же объект найден многократно, то его экземпляры будут занесены в список, что приведет к появлению в списке повторяющихся элементов. Удалить из списка повторяющиеся элементы можно с помощью предиката *компл_в_множ(List, List*), преобразующий комплект в множество уникальных элементов. Приведенная ниже программа собирает в список номера групп из фактов *студент(Фамилия, Номер_группы).*

```
/* Программа решения примера 8.4 */
domains
    студент = string.
    список_студентов = студент*.
    integerList = integer*.
class predicates
    принадлежит:(студент,список_студентов) nondeterm anyflow.
    принадлежит:(integer, integerList)nondeterm anyflow.
    присоединить:(integerList, integerList, integerList)
        nondeterm anyflow.
    компл_в_множ:(integerList,integerList) nondeterm anyflow.
    студент:(студент, integer) nondeterm anyflow.
    p:()nondeterm.
clauses
    /* определение отношения <студент>(<ФИО>, <группа>) */
    студент("Иванов", 243).
    студент("Петров", 244).
    студент("Ивагин", 243).
    студент("Лунева", 244).
    студент("Сергеева", 243).
```

```
студент("Рудаков", 244).
/* предикат из одного списка создает другой, в котором все
элементы первого списка встречаются только один раз */
компл_в_множ([X1], [X1]):- !.
компл_в_множ([X1|T1], [X1|T2]):- компл_в_множ(T1, T2),
                                 not(принадлежит(X1, T2)), !.
компл_в_множ([X1|T1], T2):-      компл_в_множ(T1, T2).
p():-     findall(G, студент(F, G), List), write(List), nl,
компл_в_множ(List, List1), write(List1),nl.

run():-   console::init(), p(), nl,!.
run().
```

Результат:

```
[243, 244, 243, 244, 243, 244]
[243, 244]
```

8.10. Составные списки

Составные списки – это списки, состоящие из элементов разного типа, в том числе, возможно, из подсписков. Элемент такого списка задается с помощью структуры. Для каждого типа данных вводится свое имя структуры: для целых – *i()*, для символов – *c()*, для списков – *l()* и т.п. В одной области *llist* описываются все указанные структуры, тогда составной список – это список из элементов *llist*. Как описывать такие списки и как выполнять с ними операции иллюстрирует следующий пример. В нем решается задача преобразования списка с подсписками в обычную последовательность элементов. Для этого используется предикат *линеаризовать*. Составной список сначала выводится, затем преобразуется и вновь выводится с помощью предиката вывода составных списков *plist*.

```
/* Составные списки */
domains
    llist = l(list); i(integer); c(char); s(string).
    list = llist*.
class predicates
    присоединить:(list, list, list) nondeterm anyflow.
    линеаризовать:(list, list) nondeterm anyflow.
    pp:() nondeterm.
    plist:(list) nondeterm (i).
clauses
    присоединить([], Y, Y).
    присоединить([X1|L1], Y, [X1|L3]):-
            присоединить(L1, Y, L3).
```

```
        линеаризовать([], []).
линеаризовать([X|Xvost], OutList):-
        X = l(Y), линеаризовать(Y, Z),
        линеаризовать(Xvost, W),
        присоединить(Z, W, OutList).
линеаризовать([X|Xvost],OutList):-
        линеаризовать(Xvost,W),
        присоединить([X],W,OutList).
plist([]).
plist([H|T]):-      H = i(Y), write(Y," "),
                    plist(T).
plist([H|T]):-      H =c(Y), write(Y," "),
                    plist(T).
plist([H|T]):-      H = s(Y), write(Y," "),
                    plist(T).
plist([H|T]):-      H = l(Y), write("[," "),
                    plist(Y), write("]," "),
                    plist(T).

pp():-      X =[s("massa"), l([ i(4), i(5), c('y')]),  i(8), l([ c('t'), c('w')])],
            plist(X),nl,
            линеаризовать(X, OutList), !,
            plist(OutList), nl.
run():-     console::init(),
            pp(), nl,!.
run().
```

Результаты выполнения программы:

```
massa [4 5 y] 8 [t w ]
massa 4 5 y 8 t w
```

Для работы со списками в *Visual Prolog 7* разработан класс ***list***, который содержит ряд полезных предикатов работы со списками. С ними Вы можете познакомиться в каталоге ***pfc*** – фундаментальные классы Пролога и использовать их в своих программах.

Пример выполнения контрольного задания в консольном режиме

```
/* Пример выполнения задания по теме "Списки"
в консольном режиме*/
domains
    fio=string. fucultet=string. discip=string. tip=string.
    kaf=string. tel=string. denj=string. c_z = string.
    ngruppa=integer. kol_vo=integer. shifr=integer.
```

```
        nau=integer. vremja = integer.
        stringlist = string*. aud = aud(nau,kol_vo).
        audList = aud*.
    class facts – raspisanieDB
        студент:(fio, ngruppa).
        группа:(ngruppa, fucultet, kol_vo, fio).
        дисциплина:(shifr, discip, kol_vo).
        преподаватель:(fio, kaf, tel).
        аудитория:(nau, tip, kol_vo).
        расписание_занятий:(ngruppa, shifr, fio, nau, denj, vremja, c_z).
    class predicates
        список_группы_1:(ngruppa, kol_vo) nondeterm (i,o).
        вывод:(stringList) nondeterm anyflow.
        длина:(stringList, kol_vo) nondeterm anyflow.
        аудитория_макс:(aud) nondeterm anyflow.
        maxList:(aud, audList) nondeterm anyflow.
        max:(aud, aud, aud) nondeterm anyflow.
        p:() nondeterm.
        aud_kol:(aud) nondeterm anyflow.
    clauses
        группа(243, "ФВТ", 23, "Карасев").
        группа(244, "ФВТ", 25, "Карев").

        студент("Иванов", 243).
        студент("Петров", 243).
        студент("Ивагин", 243).
        студент("Лунева", 243).
        студент("Волков", 244).
        студент("Мосин", 244).

        аудитория(132, "ауд", 30).
        аудитория(206, "лаб", 300).
        аудитория(403, "ауд", 30).
        аудитория(411, "ауд", 10).

        /* Предикат создания и вывода списка студентов заданной
        группы и определения числа студентов */
        список_группы_1(Ng, Nst):-
                findall(F, студент(F, Ng), List),
                /* Собрать в список для заданной
                группы фамилии студентов */
                вывод(List),
                /* Вывести этот список на экран */
                длина(List, Nst).

        /* Предикат вывода элементов списка */
        вывод([]).
```

```
        вывод([X|T]):-    write(" ", X), nl,
                          вывод(T).
        /* Предикат определения числа элементов в списке */
        длина([], 0).
        длина([X|T], N):- длина(T,N1),
                          N = N1 + 1.

        aud_kol(aud(Aud, Nsa)):-аудитория(Aud, _, Nsa).
        /* Поиск аудитории с максимальным числом мест */
        аудитория_макс(Audmax):-
                /* Собираем в список List все аудитории с указанием
                числа мест в них в формате aud(Aud, Nsa) */
                findall(Aud, aud_kol(Aud), List),
                /* Поиск Nmax в списке List */
                maxList(Audmax, List).

        /* Поиск максимального элемента в списке */
        maxList(X, [X]).
        /* Если список из одного элемента,
        этот элемент является максимальным */
        maxList(R, [X|T]):-
                maxList(R1, T),
                /* Сначала найти максимальный
                элемент R1 в хвосте списка T,
                затем за максимальный R принять
                большее из X и R1 */
                max(X, R1, R).

        /* Выбрать наибольшее из N1 и N2 */
        max(aud(Au1, N1), aud(Au2, N2), aud(Au1, N1)):-    N1>=N2.
        max(aud(Au1, N1), aud(Au2, N2), aud(Au2, N2)):-    N2>N1.
        p():-   группа(Ng, _, _, _),
                write(" Группа ", Ng), nl,
                список_группы_1(Ng, Nst),
                write("Число студентов в группе ", Ng, "= ", Nst) ,nl, fail.
        p():-   аудитория_макс(Audmax),
                write("Аудитория с максимальным числом мест",
                        Audmax), nl.
clauses
        run():- console::init(),p(),!.
        run().
```

Пример выполнения контрольного задания в режиме графического интерфейса

Задание: в фактах базы данных хранится информация о группах, их студентах, об аудиториях, преподавателях, дисциплинах и расписании занятий в институте. Приложение должно обеспечивать доступ к следующей информации: списки студентов заданной группы, аудитории и их вместимость, преподаватели и их принадлежность кафедре, расписание занятий для заданного преподавателя, для заданной группы, для заданной аудитории, для заданного дня недели.

За основу возьмем проект ***GUI_Raspisanie*** . Загрузим файл проекта и произведем следующие изменения. Дополним имеющуюся форму новыми элементами. В результате она должна иметь вид, представленный на рис. 8.2.

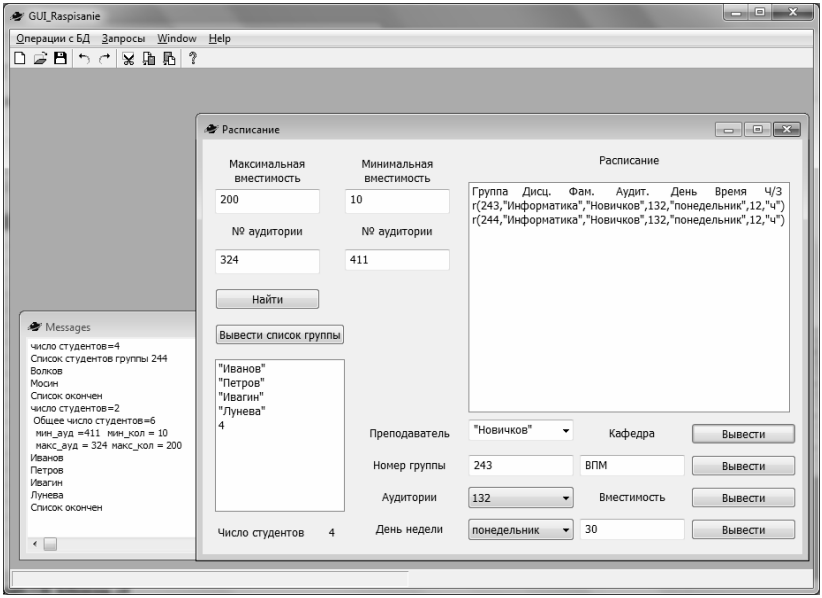


Рис. 8.2. Главное окно задач и форма приложения "Расписание"

Новыми элементами в этой форме являются поля со списком (типа *listEdit_ctl*, *listButton_ctl*): "Преподаватель", "Аудитория", "День недели". Осуществляя выбор в этих полях и нажимая соответствующую им кнопку "Вывести", получаем в окне вывода (типа *listBox_ctl*) "Расписание" расписание занятий для выбранной величины. Чтобы наполнить

элементы типа *listEdit_ctl*, *listButton_ctl*, *listBox_ctl* строками вывода, надо их сначала собрать (*findAll*) в список *List*, затем этот список добавить к элементу: метод *addList*(*List*). Ниже приведены основные файлы программы.

```
/*Описание класса Crasp – заголовочный файл crasp.cl*/
/* Описанные в нем предикаты и типы доступны форме
 и главному окну задач */
class crasp : crasp
        open core
predicates
        classInfo : core::classInfo.
domains
        fio=string. fucultet=string. discip=string. tip=string.
        kaf=string. tel=string. denj=string. c_z = string.
        ngruppa=integer. kol_vo=integer. shifr=integer.
        nau=integer. vremja = integer.
        r = r(ngruppa, discip, fio, nau, denj, vremja, c_z).
        lSrasp = string*.
        lstud = string*.
predicates
        список_группы:(ngruppa, kol_vo, lstud) determ (i,o,o).
        списки:() determ.
        аудитории_макс_мин:(nau, kol_vo, nau, kol_vo)
                nondeterm (o,o,o,o).
        наибольшее:(nau, kol_vo, nau, kol_vo, nau, kol_vo)
                nondeterm (i,i,i,i,o,o).
        наименьшее:(nau, kol_vo, nau, kol_vo, nau, kol_vo)
                nondeterm(i,i,i,i,o,o).
        загрузитьБД:().
        сохранитьБД:().
        студент_п:(fio, ngruppa) nondeterm (i,o) (i,i) (o,o) (o,i).
        аудитория_п:(nau, tip, kol_vo) nondeterm (i,o,o) (i,i,i) (o,o,o) (o,i,i).
        преподаватель_п:(fio, kaf, tel) nondeterm (i,o,o) (i,i,i) (o,o,o) (o,i,i).
        расп_группы:(ngruppa, lSrasp) determ (i,o).
        расп_группыP:(fio, lSrasp) determ (i,o).
        расп_группыA:(nau, lSrasp) determ (i,o).
        расп_группыD:(denj, lSrasp) determ (i,o).
end class crasp

/*Файл реализации класса crasp.pro*/
implement crasp
        open core,stdio
constants
        className = "Prasp/crasp".
        classVersion = "$JustDate: $$Revision: $".
```

```
clauses
        classInfo(className, classVersion).
/* Пример выполнения задания 4 */
class facts – raspisanieDB /* БД фактов */
        kol:(kol_vo) determ.
        студент:(fio, ngruppa).
        sum:(kol_vo) determ.
        группа:(ngruppa, fucultet, kol_vo, fio).
        дисциплина:(shifr, discip, kol_vo).
        преподаватель:(fio, kaf, tel).
        аудитория:(nau, tip, kol_vo).
        расписание_занятий:(ngruppa, shifr, fio, nau, denj, vremja, c_z).
        max_min:(kol_vo, nau, kol_vo, nau) determ.

clauses
        /* С помощью этих предикатов информация из фактов БД
        делается доступной другим модулям */
        студент_п(Фио, НомерГР):-       студент(Фио, НомерГр).
        аудитория_п(Nau, Tip, Kol_vo):- аудитория(Nau,Tip,Kol_vo).
        преподаватель_п(Fio, Kaf, Tel):- преподаватель(Fio, Kaf, Tel).
/*Предикат обновления БД фактов из файла FileName*/
class predicates
        reconsult : (string FileName).
clauses
        reconsult(Filename) :-
                retractFactDB(raspisanieDB),
                file::consult(Filename,raspisanieDB).

/* Предикат выполняется при выборе пункта меню
"ЗагрузитьБД"*/
class facts
        currentDirectory: string:="". %Описание факта currentDirectory
clauses
        загрузитьБД():-
                currentDirectory:=directory::getCurrentDirectory(),
                directory::setCurrentDirectory("..\\"),
                Filename = vpiCommonDialogs::getFileName( "",
                        ["Text files (*.txt)", "*.*"],
                        "Загрузка БД о студентах",
                        [], ".", _),
                directory::setCurrentDirectory(currentDirectory), ! ,
                reconsult(Filename),
                stdIO::writef("База данных % загружена\n", Filename).
        загрузитьБД().
/* Предикат выполняется при выборе пункта меню
"СохранитьБД"*/
```

```
clauses
        сохранитьБД():-
                currentDirectory:=directory::getCurrentDirectory(),
                directory::setCurrentDirectory("..\\"),
                Filename = vpiCommonDialogs::getFileName( "",
                        ["Text files (*.txt)", "*.*"],
                        "Загрузка БД о студентах", [], ".", _),
                        directory::setCurrentDirectory(currentDirectory), !,
                file::save(Filename,raspisanieDB),
                stdIO::writef("База данных % сохранена\n", Filename).
        сохранитьБД().

clauses
/* Формирование списка фамилий студентов группы
и определение числа студентов в группе */
        список_группы(NG, _,_):-
                assert(kol(0)),
                студент(F, NG), write(F), nl,
                retract(kol(N)),
                N1 = N + 1,
                asserta(kol(N1)), fail.
        список_группы(NG,Nst, List):-
                retract(kol(Nst)),
                write("Список окончен"),nl,
                findall(tostring(F),студент(F, NG), List).
/* Для каждой группы выводится список студентов и определяется
число студентов в группе, а также общее число студентов */
        списки():-
                assert(sum(0)),
                группа(Ng, _, _, _),
                write("Список студентов группы ", NG), nl,
                список_группы(Ng,Nst,_),
                write("число студентов="), write(Nst), nl,
                retract(sum(S)),
                S1 = S + Nst,
                asserta(sum(S1)),
                fail.
        списки():-
                retract(sum(S)),
                write(" Общее число студентов="),
                write(S),nl.
/* Определяются аудитории с максимальным и минимальным
числом мест */
        аудитории_макс_мин(Aumax, Nmax, Aumin, Nmin):-
                assert(max_min(0, 0, 1000, 1)),
                аудитория(Au, _, Ns),
```

```
                retract(max_min(Max, Amax, Min, Amin)),
                наибольшее(Au, Ns, Amax, Max, Amax1, Max1),
                наименьшее(Au, Ns, Amin, Min, Amin1, Min1),
                asserta(max_min(Max1, Amax1, Min1, Amin1)),
                fail.
        аудитории_макс_мин(Aumax, Nmax, Aumin, Nmin):-
                retract(max_min(Nmax, Aumax, Nmin, Aumin)).

        наибольшее(Au,Ns,Amax,Max,Amax,Max):- Max>Ns.
        наибольшее(Au,Ns,Amax,Max,Au,Ns):- Ns>=Max.
        наименьшее(Au,Ns,Amin,Min,Amin,Min):- Min<Ns.
        наименьшее(Au,Ns,Amin,Min,Au,Ns):- Ns<=Min.
/*Предикат, собирающий строки расписания для заданной группы,
в список Расп_Сп*/
        расп_группы(Группа, Расп_Сп):-
                findall(X, расписание_занятий(Группа, X),Расп_Сп).
class predicates
        расписание_занятий:(ngruppa,string) nondeterm anyflow.
clauses
        расписание_занятий(Группа, Str):-
                расписание_занятий(Группа, Shifr,Fio, Nau, Denj,
                        Vremja, C_z),
                дисциплина(Shifr, Discip, _),
                Y =r(Группа, Discip,Fio, Nau, Denj, Vremja, C_z),
                Str = tostring(Y).
/*Предикат, собирающий строки расписания для заданного препо-
давателя в список Расп_Сп*/
        расп_группыР(Fio, Расп_Сп):-
                findall(X, расписание_занятийР(Fio, X),Расп_Сп).
class predicates
        расписание_занятийР:(fio,string) nondeterm anyflow.
clauses
        расписание_занятийР(Fio, Str):-
                расписание_занятий(Группа, Shifr,Fio, Nau, Denj,
                        Vremja, C_z),
                дисциплина(Shifr, Discip, _),
                Y = r(Группа, Discip,Fio, Nau, Denj, Vremja, C_z),
                Str = tostring(Y).
/*Предикат, собирающий строки расписания для заданной аудито-
рии в список Расп_Сп*/
        расп_группыА(Aud, Расп_Сп):-
                findall(X, расписание_занятийА(Aud, X),Расп_Сп).
class predicates
        расписание_занятийА:(nau,string) nondeterm anyflow.
clauses
        расписание_занятийА(Nau, Str):-
```

```
                расписание_занятий(Группа, Shifr,Fio, Nau, Denj
                        Vremja, C_z),
                дисциплина(Shifr, Discip, _),
                Y = r(Группа, Discip,Fio, Nau, Denj, Vremja, C_z),
                Str = tostring(Y).

/*Предикат, собирающий строки расписания для заданного дня не-
дели в список Расп_Сп*/
        расп_группыD(Den, Расп_Сп):-
                indall(X, расписание_занятийD(Den, X),Расп_Сп).
class predicates
        расписание_занятийD:(denj,string) nondeterm anyflow.
clauses
        расписание_занятийD(Den, Str):-
                расписание_занятий(Группа, Shifr,Fio, Nau, Den,
                        Vremja, C_z),
                дисциплина(Shifr, Discip, _),
                Y = r(Группа, Discip,Fio, Nau, Den, Vremja, C_z),
                Str = tostring(Y).
end implement crasp

                        /* Файл реализации формы */
implement frmRasp
        inherits formWindow /*Наследование от formWindow*/
        open core, vpiDomains /*Подключаем класс vpiDomains*/

/*Текст пропущен*/
clauses
        display(Parent) = Form :- /*Конструктор формы*/
                 Form = new(Parent),
                 Form:show().
clauses
        new(Parent):-
                formWindow::new(Parent),
                generatedInitialize().
predicates
/*Обработчик кнопки "Найти" аудитории с максимальной
и минимальной вместимостью*/
        onMax_min_a_puchClick : button::clickResponder.
clauses
        onMax_min_a_puchClick(_Source) = button::defaultAction:-
                /*Поиск аудиторий*/
                crasp::аудитории_макс_мин(Na1,Ka1,Na2,Ka2),
                /*Вывод макс. вместим. в поле редактиров. max_ctl*/
                max_ctl:setText(tostring(Ka1)),
                /*Вывод миним. вместим. в поле редактиров. min_ctl*/
                min_ctl:setText(tostring(Ka2)),
```

```
                /*Выв. номер. аудит. в поля редакт. aumax_ctl,
                aumin_ctll*/
                aumax_ctl:setText(tostring(Na1)),
                aumin_ctl:setText(tostring(Na2)),
                /*Вывод информации в окно сообщений*/
                stdio::write(" мин_ауд =",Na2," мин_кол = ", Ka2,
                        "\n макс_ауд = ",Na1," макс_кол = ",Ka1),
                stdio::nl,!.
        onMax_min_a_puchClick(_Source) =button::defaultAction().
predicates
/*Предикат для наполнения поля со списком listсписок_ctl фами-
лиями студентов заданной группы*/
        списокГр:(crasp::ngruppa)procedure.
clauses
        списокГр(Ngruppa):-
                listсписок_ctl:add("Список группы"),
                crasp::студент_п(Фио, Ngruppa),
                listсписок_ctl:add(Фио), fail.
        списокГр(Ngruppa).
predicates
/*Обработчик кнопки "Вывести список группы" для заданной
группы */
        onPushButtonClick : button::clickResponder.
clauses
        onPushButtonClick(_Source) = button::defaultAction:-
                /*Взять текст из поля группа*/
                Ngruppa = ngr_ctl:getText(),
                /*Для заданной группы формируется список
                студентов List и число студентов Kl_vo*/
                crasp::список_группы(toterm(Ngruppa), Kl_vo, List),
                /*В окно вывода listсписок_ctl добавляется список List и
                число студентов Kl_vo*/
                listсписок_ctl:addList(List),
                listсписок_ctl:add(tostring(Kl_vo)),
                kol_ctl:setText(tostring(Kl_vo)),!.
        onPushButtonClick(_Source) = button::defaultAction.
/*Обработчик кнопки "Вывести" для заданной группы расписание*/
predicates
        onPushButtonRaspgrClick : button::clickResponder.
clauses
        onPushButtonRaspgrClick(_Source) = button::defaultAction:-
                /*Взять текст из поля группа*/
                 Ngruppa = ngr_ctl:getText(),
                /*Очистить окно вывода для расписания*/
                listboxrasp_ctl:clearAll(),
                /*Вывести заголовок в окно вывода*/
```

```
                listboxrasp_ctl:add("Группа Дисц. Фам. Аудит.
                        День Время Ч/З"),
                /*Получить список студентов для заданной группы*/
                crasp::расп_группы(toterm(Ngruppa),List),
                /*Добавить этот список в окно вывода*/
                listboxrasp_ctl:addList(List), !.
        onPushButtonRaspgrClick(_Source)=button::defaultAction.
/*Обработчик события "Изменение выбора в поле со списком
аудитория"*/
predicates
        onListButtonAudSelectionChanged:
                listControl::selectionChangedListener.
clauses
        onListButtonAudSelectionChanged(_Source):-
                /*Берем из поля со списком listButtonAud_ctl ном. Ауд.*/
                Na = toterm(listButtonAud_ctl:getText()),
                /*Находим вместимость этой аудитории Kol_vo*/
                crasp::аудитория_п(Na, _, Kol_vo),
                /*Вместимость Kol_vo выводим в поле editAud_ctl*/
                editAud_ctl:setText(tostring(Kol_vo)),!.
                onListButtonAudSelectionChanged(_Source).
/*Обработчик события "Активация формы". Заполняются
все поля со списком и инициализ. соответствующие им поля
редактирования*/
predicates
        onActivate : documentWindow::activateListener.
clauses
        onActivate(_Source):-
                /*Наполняется поле со списком все аудитории*/
                listButtonAud_ctl:setFocus,
                listButtonAud_ctl:clearAll(),
                findall(tostring(Naud),
                crasp::аудитория_п(Naud, _,_), List),
                listButtonAud_ctl:addList(List),
                /*Указатель списка устанавливается на первый эл.*/
                listButtonAud_ctl: selectAt (0, true),
                /*Определяется и выводится соответств.
                выбранной аудитории вместимость*/
                Na = toterm(listButtonAud_ctl:getText()),
                crasp::аудитория_п(Na, _, Kol_vo),
                editAud_ctl:setText(tostring(Kol_vo)),
                /*Наполняется поле со списком все преподаватели*/
                listEditPrep_ctl:setFocus, listEditPrep_ctl:clearAll(),
                findall(tostring(F), crasp::преподаватель_п(F, _,_), ListP),
                listEditPrep_ctl:addList(ListP),
                listEditPrep_ctl: selectAt(0, true),
```

```
                Fio =listEditPrep_ctl:getText(),
                crasp::преподаватель_п(toterm(Fio), Kaf,_),
                editPrep_ctl:setText(Kaf),
                /*Наполняется поле со списком все дни недели*/
                listButtonDen_ctl:setFocus,
                listButtonDen_ctl:clearAll(),
                listButtonDen_ctl:addList(["понедельник", "вторник",
                        "среда", "четверг", "пятница", "суббота"]),
                listButtonDen_ctl: selectAt (0, true),
                !.
        onActivate(_Source).
/*Обработчик события "Изменение выбора в поле со списком преподаватель"*/
predicates
        onListEditPrepSelectionChanged:
                listControl::selectionChangedListener.
clauses
        onListEditPrepSelectionChanged(_Source):-
                Fio = toterm(listEditPrep_ctl:getText()),
                crasp::преподаватель_п(Fio, Kaf,_),
                editPrep_ctl:setText(tostring(Kaf)),
                !.
        onListEditPrepSelectionChanged(_Source).
/*Обработчик кнопки "Вывести" для заданного преподавателя расписание*/
predicates
        onPushButton1Click : button::clickResponder.
clauses
        onPushButton1Click(_Source) = button::defaultAction:-
                Prep = listEditPrep_ctl:getText(),
                listboxrasp_ctl:clearAll(),
                listboxrasp_ctl:add("Группа Дисц. Фам. Аудит. День
                        Время Ч/З"),
                crasp::расп_группыР(toterm(Prep),List),
                listboxrasp_ctl:addList(List),
                !.
        onPushButton1Click(_Source)= button::defaultAction.
/*Обработчик кнопки "Вывести" для заданной аудитории расписание*/
predicates
        onPushButton2Click : button::clickResponder.
clauses
        onPushButton2Click(_Source) = button::defaultAction:-
                Aud = listButtonAud_ctl:getText(),
                listboxrasp_ctl:clearAll(),
                listboxrasp_ctl:add("Группа Дисц. Фам. Аудит. День
```

```
                    Время Ч/З"),
            crasp::расп_группыA(toterm(Aud),List),
            listboxrasp_ctl:addList(List),
            !.
    onPushButton2Click(_Source)= button::defaultAction.
/*Обработчик кнопки "Вывести" для заданного дня недели
расписание*/
predicates
    onPushButton3Click : button::clickResponder.
clauses
    onPushButton3Click(_Source) = button::defaultAction:-
            Den = listButtonDen_ctl:getText(),
            listboxrasp_ctl:clearAll(),
            listboxrasp_ctl:add("Группа Дисц. Фам. Аудит. День
                    Время Ч/З"),
            crasp::расп_группыD(Den,List),
            listboxrasp_ctl:addList(List),
            !.
    onPushButton3Click(_Source)= button::defaultAction.

% This code is maintained automatically, do not update it manually.
% 13:29:02-31.7.2010
% end of automatic code
end implement frmRasp

/*Фрагмент файла TaskWindow.pro – главное окно задач */
/*Обработчик выбора пункта меню "Загрузить БД"*/
predicates
    onFileNew : window::menuItemListener.
clauses
    onFileNew(_Source, _MenuTag):-
            crasp::загрузитьБД(),crasp::списки(),!.
            onFileNew(_Source, _MenuTag).
/*Обработчик выбора пункта меню "Открыть форму запросов"*/
predicates
    onEditUndo : window::menuItemListener.
clauses
    onEditUndo(W, _MenuTag):- S = frmRasp::new(W), S:show().
/*Обработчик выбора пункта меню "Сохранить БД"*/
predicates
    onFileSave : window::menuItemListener.
clauses
    onFileSave(_Source, _MenuTag):-
            crasp::сохранитьБД(),!.
    onFileSave(_Source, _MenuTag).
/*Обработчик выбора пункта меню "Вывести списки групп"*/
```

```
predicates
        onEditRedo : window::menuItemListener.
clauses
        onEditRedo(_Source, _MenuTag):-
                crasp::списки(),!.
        onEditRedo(_Source, _MenuTag).
```

Контрольные вопросы

1. Что такое список?
2. Какие значения, если они существуют, будут присвоены переменным при сопоставлении шаблона [*X1*, *X2*, *X3*|*Z*] со следующими списками: [*a*, *b*, *c*, *d*, *e*], [*a*, *b*, *c*, *d*], [*a*, *b*], [*a*], [].
3. Запишите предложение "список – это структура, состоящая из двух элементов: головы и хвоста в виде списка".
4. Задайте шаблон, представляющий:

- список из трех элементов, второй элемент которого равен 2;
- список, первый элемент которого является подсписком, состоящим, по крайней мере, из двух элементов.

5. Какие значения будут присвоены *X* и *Y* при сопоставлении двух шаблонов [*e*|*X*] и [*Y*, *c*|*Z*]?
6. Какие ответы вы получите на запросы:

- принадлежит(X, ['Р','О','Б','Е','Р','Т']),
- принадлежит(X, ['Б','О','Б']).
- принадлежит(X, ['А','Л','Ь','Ф','А']),
- принадлежит(X, ['Ф','Р','Е','Д']).?

7. Задайте правила, определяющие отношение: *X* – последний элемент списка *L*.
8. Задайте правила, определяющие отношение: длина *L* списка *Z*.
9. Опишите предикат проверки, является ли заданный терм списком.

Контрольные задания

Выполнить свой вариант из контрольных заданий по теме 5 с использованием списков. Для этого сначала собрать в список необходимые данные из фактов базы знаний, затем осуществить их обработку в соответствии с заданием.

Глава 9. СТРОКИ, СИМВОЛЫ И СИМВОЛИЧЕСКИЕ ИМЕНА

9.1. Строки

Под строкой в общем смысле слова понимается последовательность из нуля или более символов. В Прологе такая последовательность используется для образования двух видов термов: *символических имен и строк*. ***Символическое имя –*** это непрерывная последовательность букв, цифр и знака подчеркивания, начинающаяся с маленькой буквы (*стол_стул, a1, м3 и т.д.*). Символическое имя относится к типу ***symbol***. ***Строка*** – это последовательность любых символов, заключенная в **двойные кавычки,** строка относится к типу ***string.*** Например: "Иван да Марья", "Стол и стул" и т.д. Символические имена можно использовать для именования ***объектов и отношений***, а строки только для названия объектов и значений их атрибутов. Символические имена и строки автоматически преобразуются друг в друга, и все предикаты, определенные для строки, могут быть применены и к символическим именам. Основное различие между ними в том, что символические имена хранятся в таблице в оперативной памяти и доступ к ним осуществляется быстрее, чем к строкам, обработка которых ведется посимвольно.

Для ввода строк используется предикат ***Str = readLine()***, для ввода символов – ***Ch = readChar(),*** а для выводов любых термов – предикат ***write(…).*** Ниже приводится пример описания строк, символических имен и символов, а также применение предикатов ввода и вывода для этих типов данных.

```
domains
        символ = char.
        символ_имя = symbol.
        строка = string.
clauses
        run():-   console::init(),write("Введите символ "),
                  Ch = readChar(),H = readChar(),
                  write("Ch =", Ch," "), nl,
                  write("Введите строку "),Str = readLine(),
                  write("Str =", Str," "), nl,
                  write("Введите символ_имя "),
                  hasdomain( symbol,Sym), Sym = read(),
                  write("Sym =", Sym," "),nl,!.
        run().
```

При выполнении программы получим:

```
Введите символ w
Ch = w
Введите строку молоко
Str = молоко
Ввдите символ_имя "молоко"
Sym = молоко
```

9.2. Встроенные предикаты обработки строк

В языке ***Visual Prolog*** имеется достаточно много встроенных предикатов обработки строк. Все они могут быть разделены на две группы: предикаты манипулирования строками и их элементами и предикаты преобразования строк к другим типам данных. Получить информацию об этих предикатах можно, используя справочную систему (классы ***string, string8***). Ниже мы рассмотрим наиболее часто используемые предикаты.

Предикат ***frontchar/3*** – используется для выделения первого символа строки и имеет следующий формат записи:

```
frontchar(String1, Char, String2) determ (i,o,o)
```

Предикат связывает три параметра: первый – строка *String1*, которая делится на символ *Char* – второй параметр, и оставшуюся часть строки *String2* – третий параметр. Ниже приведены примеры использования этого предиката.

Пример 9.1. Написать предикат преобразования строки в список символов. При доказательстве цели *string_chlist("dog", L)* должен получиться список ['d','o','g']

```
/*Преобразование строки в список символов*/
domains
        charlist = char*. /*Список символов*/
class predicates
/*Преобразование строки в список символов*/
        string_chlist:(string, charlist) nondeterm anyflow.
clauses
        string_chlist("", []):- !.
        string_chlist(S, [H|T]):-
                frontchar(S, H, S1),
                /*Выделяем первый символ H
                и подсоединяем [H|T] его */
                string_chlist(S1, T).
                /* к хвосту T, полученному из
```

```
                оставшейся части строки S1 */
        run():-  console::init(),string_chlist("локон волос", L),
                write("локон волос","\t",L), nl,!.
        run().
```

Пример 9.2. Написать предикат замены символа *A* на символ *B*. При доказательстве цели *zamena_simv("локон волос", L, 'л', 'к')* строка "локон волос" преобразуется в строку "кокон вокос".

```
/*Преобразование списка символов в строку*/
class predicates
    chlist_string:(charlist, string)nondeterm anyflow.
clauses
    chlist_string([], "").
    chlist_string([H|T], S):-
            chlist_string(T, S1),
            S = concat(chartostring(H),S1).
/* Предикат замены символов в строке */
class predicates
    zamena_simv:(string, string, char, char) nondeterm (i,o,i,i).
clauses
    zamena_simv(S, S1, A, B):-
            string_chlist(S, LS),
            замена(LS, RLS, A, B),!,
            chlist_string(RLS, S1).
class predicates
    замена:( charlist, charlist, char, char) nondeterm anyflow.
clauses
    замена([], [],_, _).
    замена([A|T1], [B|T2], A, B):-
            замена(T1,T2,A , B).
    замена([X|T1], [X|T2], A, B):-
            замена(T1,T2,A , B).
    run():-  console::init(),
            L = replaceAll("локон волос","л","к"),
            write("локон волос","\t",L), nl,
            zamena_simv("локон волос", L1, 'л','к'),
            write("локон волос","\t",L1), nl,!.
    run().
```

Следующий важный предикат *fronttoken/3* разделяет текст или строку *String1* на первую лексему (слово) *Token* и оставшуюся часть текста или строки *Rest*, и имеет формат:

```
fronttoken(String1, Token, Rest) (i,o,o) (i,i,o) (i,o,i) (i,i,i) (o,i,i)
```

Лексема – это слово, знак препинания, число и т.п. Ниже приведены примеры использования предиката ***fronttoken/3***.

Пример 9.3. Написать предикат преобразования строки (текста) в список слов (лексем). При доказательстве цели *string_namelist("Я играю на гармошке!", X)* должен получиться список [*"Я", "играю", "на", "гармошке", "!"*].

```
/*Программа преобразования текста в список слов и знаков
препинания*/
domains
        namelist = name*.
        /*Список строк */
        name = string.
class predicates
        string_namelist:(string, namelist)nondeterm anyflow.
        /* Предикат преобразования текста в список слов */
clauses
        string_namelist(S, [H|T]):-
                fronttoken(S, H, S1), !,
                /* Выделяется первое слово H и подсоединяется к */
                string_namelist(S1, T).
                /* списку T, полученному при преобразовании
                оставшейся части текста S1 в список слов */
        string_namelist(_, []).
        run():-  console::init(),
                string_namelist("Я играю на гармошке!", X),
                write("Я играю на гармошке!","\t",X), nl,!.
        run().
```

Для соединения строк (соответственно двух, трех, четырех и т.д.) в одну строку могут использоваться следующие предикаты: string::concat/2->, string::concat/3->, string::concat/4->, string::concat/5->, string::concat/6->, string::concatList/1->. Пример использования:

```
"two" = concat("","two")
"22.12.2010" = concat("22",".","12",".","20","10")
"Моя внучка"=concatList(["М","о","я"," ","в","н","у","ч","к","а"])
```

Предикат ***string::length/1*** используется для определения длины строки, например: следующие утверждения истинны ***5 = = length("12345"), 0 = length("")***. Для создания строки нужных символов заданной длины можно использовать предикат ***string::create/2***. Например, ***" " = create(8, " "), "ABABAB" = create(6, "AB").***

В табл. 9.1 и табл. 9.2 перечислены некоторые предикаты и описаны выполняемые ими функции, приведены примеры. Все эти предикаты определены в классе ***string***.

Знак –> означает что предикат выполняет роль не только логической функции, но и, как обычная функция, возвращает значение некоторого объекта предметной области (числа, строки, символа и т.д.).

Табл. 9.1. Встроенные предикаты обработки строк

Предикат	Описание	Назначение
string::subString/3	*subString: (string Source, charCount Position, charCount HowLong) –> string Output*	копирует из строки *Source* подстроку *Output*, начиная с позиции *Position* длиной *HowLong*, например, цель *Str_out = substring ("Мой-Додыр", 4, 2)* даст результат *Str_out = "До"*
string::subchar/2	*subchar: (string Source, charCount Position) –>char Rezchar*	возвращает символ *Rezchar*, находящийся в позиции *Position* строки *Source*
string::search/2	*searchChar: (string Source, char LookChar) –> charCount Position determ*	возвращает позицию *Position* первого появления подстроки *Lookfo*r в строке *Source*
string::searchChar/2	*searchChar: (string Source, char LookChar) –> charCount Position determ*	возвращает позицию *Position* первого появления символа *LookChar* в строке *Source*
string::replaceAll/3	*replaceAll: (string Source, string ReplaceWhat, string ReplaceWith) –> string Output*	заменяет все вхождения строки *ReplaceWhat* на строку *ReplaceWith* в строке *Source*

Табл. 9.2. Предикаты преобразования типов

Предикат	Описание	Назначение
string::charToString/1	*charToString: (char Char) –> string CharAsString*	преобразует символ *Char* в строку *CharAsString*, содержащую этот символ
string::createFromCharList/1	*createFromCharList: (char* CharList) –> string String*	преобразует список символов *CharList* в строку *String*
string::toCharList/1	*toCharList: (string Source) –> char* CharList*	преобразует строку *Source* в список символов *CharList*

Окончание табл. 9.2.

Предикат	Описание	Назначение
string::getCharFromValue/1	*getCharFromValue: (core:: unsigned16 Value)*	преобразует цифру в символ
string::getCharValue/1	*getCharValue: (char Char) –> core:: unsigned16 Value*	пребразует символ в цифру

9.3. Сравнение символов, строк и символических имен

Символы, строки и символические имена можно сравнивать. Например:

```
'п' > 'т'                    /*да*/
"варежка" > "вареник"        /*нет*/
```

Сравнение символов. При сравнении символов происходит сравнение их кодов.

Сравнение строк. При сравнении строк последовательно сравниваются символы в соответствующих позициях. Как только найдены отличающиеся символы, так результат определен как результат сравнения их кодов. Если символы вначале одинаковые, но одна строка короче другой, то она принимается за меньшую.

Пример выполнения контрольного задания

Задание: написать предикат *поиск_по_маске(Str, Maska)*, с помощью которого осуществляется сравнение строки *Str* со строкой-маской *Maska*. Строка-маска – это искомая последовательность символов, включающая символы "*". Каждый символ "*" показывает, что на его месте может находиться любое количество любых символов. Строка-маска задает шаблон, которому может удовлетворять множество строк. Задание: написать предикат ***поиск_по_маске(string, string)*** для тестирования строк, удовлетворяют они заданной маске или нет.

```
/*Поиск по маске*/
domains
        charlist = char*. /*Список символов*/
class predicates
        string_chlist:(string, charlist) nondeterm anyflow.
        /*Преобразование строки в список символов*/
```

```
        поиск_по_маске:(string, string) nondeterm anyflow.
        поиск_по_маске1:(charlist, charlist) nondeterm anyflow.
        пропустить:(charlist, charlist) nondeterm anyflow.
clauses
        string_chlist("", []):-!.
        string_chlist(S, [H|T]):-
                frontchar(S, H, S1),
                /* Выделяем первый символ H
                и подсоединяем [H|T] его */
                string_chlist(S1, T).
                /* к хвосту T, полученному из
                оставшейся части строки S1 */
        /* Определение отношения поиск_по_маске(Str,Maska),
        Str - тестируемая строка,
        Maska - строка-маска для поиска,
        в маске допустим только один управляющий символ "*",
        который обозначает любое количество любых символов */
        поиск_по_маске(String, Maska):-
                string_chlist(String, LS),
                string_chlist(Maska, LM),
                поиск_по_маске1(LS,LM).
        поиск_по_маске1([],[]).
        /*Пустые строки равны*/
        поиск_по_маске1([X|T], [X|T1]):-
                /*Текущие символы совпадают*/
                поиск_по_маске1(T, T1).
        /*Проверить оставшиеся*/
        поиск_по_маске1([X|T], [Zv|T1]):-
                '*' = Zv, /* Найден символ звездочка */
                пропустить(T, T1).
        /*В оставшейся части строки найти символ,
        совпадающий с первым символом за звездочкой в маске*/
        пропустить(_, []).
        /* В маске элементов больше нет*/
        пропустить([X|T], [X|T1]):-
                поиск_по_маске1(T, T1).
        /*Найден символ, совпадающий с символом маски*/
        пропустить([X|T], T1):-
                пропустить(T, T1).
        /*Пропуск символов в соответствии со звездочкой в маске*/
        run():-  поиск_по_маске("Курбатов", "К*р*а*в"),write("yes"),nl,!.
        run():-  write("no"),nl.
```

Контрольные вопросы

1. Что такое строка?
2. Какие виды строк вы знаете?
3. Какие предикаты работы с символами Вы знаете?
4. Что такое строка в двойных кавычках?
5. Как сравнить строки между собой?
6. Каким образом сцепить две строки в одну?
7. Как изменить заданные символы в строке?
8. Назовите основные встроенные предикаты для выполнения операций со строками.

Контрольные задания

Для списков, полученных в задании по теме "Списки", выполнить поиск наименования, фамилии или имени объекта по заданной в варианте маске.

1. Маска: найти наименование по первому известному символу, "А*".
2. Маска: найти наименование по двум первым известным символам, "Пе*".
3. Маска: найти наименование по последнему известному символу, "*в".
4. Маска: найти наименование по двум последним известным символам, "*ов".
5. Маска: найти наименование по известной подстроке символов, "*вор*".
6. Маска: найти наименование, не включающее заданные символы.
7. Маска: найти наименование, включающее только заданные символы.
8. Маска: найти наименование, последним символом которого является цифра.
9. Маска: найти наименование, первым символом которого является цифра.
10. По заданным фамилии, имени и отчеству сформировать строку ФИО.
11. Задано длинное наименование, сформировать его сокращение по первым буквам входящих в него слов.
12. Маска: найти наименование, в котором только некоторые символы неизвестны. Будем предполагать, что на их месте может стоять любой символ, что обозначим символом "?".
13. Маска: найти наименование, в котором на месте буквы "а" может стоять буква "о" или буква "а".

14. Маска: найти наименование с наибольшим числом символов.
15. Маска: найти наименование с наименьшим числом символов.
16. Маска: найти наименование с заданным числом символов.
17. Исключить из всех наименований все пробелы.
18. Маска: найти наименование, содержащее двойные согласные.
19. Маска: найти наименование, содержащее двойные гласные.
20. Маска: найти наименование по маске-строке "<заданные символы>*<заданные символы>*", где * обозначает любое количество любых символов.
21. Маска: найти наименование по маске-строке, содержащей только символы наименования и символы #, обозначающие, что в данной позиции может быть только цифра.

Глава 10. СТРУКТУРЫ

10.1. Введение в структуры

Одним из видов термов является структура. ***Структура*** – это единый объект (отношение), состоящий из совокупности других объектов (отношений), называемых компонентами. Компоненты группируются в структуру для удобства их использования. Структуру следует рассматривать как средство описания сложного составного объекта или сложного отношения. Например:

```
студент(фио("Петров", "Иван", "Ильич"),
        дата_рождения(6, 03, 1972)).
```

Здесь в структуру *студент* объединены два объекта:

```
фио(<фамилия>, <имя>, <отчество>)
дата_рождения(<день>, <месяц>, <год>).
```

В реальной жизни одним из примеров структур является личная карточка студента в деканате, карточка-указатель для библиотечной книги. Например:

```
книга(621026, автор("Братко", "Иван"),
        название("Программирование на языке Пролог"),
        издательство("Москва", "Вильямс"),
        год_издания(2004)).
```

Карточка-указатель содержит несколько элементов: инвентарный номер 621026, сведения об авторе, название книги, дату издания, место издания и т.д. Некоторые из элементов в свою очередь тоже можно разбить на более мелкие элементы. Например, сведения об авторе состоят из фамилии и инициалов, сведения об издательстве - из места издания и названия издательства.

Структура записывается на Прологе с помощью ее функтора (имени структуры) и списка аргументов, описывающих элементы структуры.

```
<функтор – имя структуры>(<список аргументов-элементов
        структуры>).
```

Элементы (их не более 255) заключаются в круглые скобки и разделяются запятыми. ***Функтор*** (***functor*** – имя структуры) записывается перед открывающей круглой скобкой и задается ***символическим име-***

нем. Элементами могут быть любые термы: константы, переменные и в свою очередь структуры.

Например, в структуре:

```
любит(X, "баскетбол", член_клуба("ДОСААФ")).
```

любит – функтор, *X* – переменная, *"баскетбол"* – константа, *член_клуба("ДОСААФ")* – структура с функтором *член_клуба*.

Чтобы легче было понять сложную структуру, ее обычно представляют в виде дерева (см. рис. 10.1). Каждая ветвь может указывать на другую структуру, например:

```
книга( "Анна Каренина", автор("Лев", "Николаевич", "Толстой")).
```

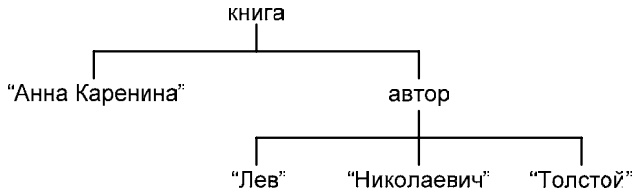


Рис. 10.1. Структура в виде дерева

Две структуры равны, если они имеют один и тот же функтор и одинаковое число аргументов, причем все соответствующие аргументы равны. Например, при согласовании следующего целевого утверждения: *ехать("Андрей", "велосипед") = ехать("Андрей", X) X* будет присвоено конкретное значение "велосипед".

Структуры могут быть вложены одна в другую на любую глубину. Попытка согласовать следующую цель:

```
станок(вал, X, деталь( гайка, F, операция(обтачка, резец, J) )) =
станок(B, 43, деталь(G, "MXII", операция(H, резец, полная))).
```

будет успешной, а переменные *B, X, G, F, H, J* будут конкретизированы и примут соответственно значения *вал, 43, гайка, "MXII", обтачка, полная*.

Рассмотрим простой пример использования структуры, члены семьи с одинаковой фамилией проживают по одному адресу, поэтому достаточно задать адрес одному члену семьи и путем унификации передать этот адрес второму члену семьи.

```
/* Пример использования структуры */
domains
```

```
        личность = личность(фио, адрес).
        фио = фио(фамилия, имя, отчество).
        адрес = адрес(город, улица, дом, квартира).
        фамилия = string. имя = string. отчество = string.
        город = string. улица = string.
        дом = integer. квартира = integer.
    class predicates
        p:().
    clauses
        p():-   P1 = личность(фио("Ковалев","Андрей","Иванович"),
                        адрес("Рязань", "Есенина", 10, 5)),
                P1 = личность(фио("Ковалев", _, _),Adress),
                P2 = личность(фио("Ковалев", "Юрий","Андреевич"),
                        Adress),
                write("P1=", P1), nl,
                write("P2=", P2), nl.
    clauses
        run():- console::init(),
                p().
```

В окне вывода результата получим следующие значения:

```
P1=личность(фио("Ковалев","Андрей","Иванович"),
адрес("Рязань","Есенина",10,5)),
P2=личность(фио("Ковалев","Юрий","Андреевич"),
адрес("Рязань","Есенина",10,5)),
1 Solution
```

Достоинством структуры является возможность обрабатывать несколько единиц данных как одно целое. Рассмотрим пример с записной книжкой, в которой хранится информация о знакомых, их днях рождения и номерах их телефонов. Эта информация описывается фактами типа *записная_книжка(личность(фамилия, имя, отчество), номер_телефона, день_рождения(месяц, день, год))*. В приведенной ниже программе выводится список знакомых, у которых день рождения в текущем месяце.

```
    /* Пример: дни рождения знакомых в текущем месяце */
    domains
        фио = фио( фамилия, имя, отчество).
        день_рождения = день_рождения(месяц, день, год).
        фамилия= string. имя= string. отчество= string.
        месяц= string. день= integer. год = integer.
        номер_телефона = string.
    class predicates
        записная_книжка:(фио, номер_телефона, день_рождения)
```

```
                nondeterm anyflow.
        список_дней_рожд:()procedure.
        месяц_номер:(месяц, integer) nondeterm anyflow.
        контроль_месяц:(integer, день_рождения) nondeterm anyflow.
        вывод_фио:(фио) nondeterm anyflow.
clauses
        список_дней_рожд():-
                write("***************Список дней рождения
                        в текущем месяце***************"),nl,
                write("Фамилия\t\tИмя\t\tОтчество\n"),
                write("*******************************************"),nl,
                %time::getDate(_, This_month, _),
                This_month =2,
                записная_книжка(Fio, _, Date),
                контроль_месяц(This_month, Date),
                вывод_фио(Fio),
                fail.
        список_дней_рожд():-
                write("\n\nНажмите любую клавишу
                        для продолжения "),nl,
                H = readChar().

        вывод_фио(фио( Fio, Ima, Otc)):-
                write(" ",Fio, "\t\t ",Ima, "\t\t ", Otc), nl.

        контроль_месяц(Mon, день_рождения(Month,_,_)):-
                месяц_номер(Month, Month1),
                Mon = Month1.
        записная_книжка(фио("Петров", "Сергей","Сергеевич"),
                "44-66-34", день_рождения("январь", 25, 1996)).
        записная_книжка(фио("Клобков", "Сергей","Иванович"),
                "55-66-34",день_рождения("январь", 17, 1976)).
        записная_книжка(фио("Родина", "Мария","Сергеевна"),
                "44-39-34",день_рождения("май", 15, 1999)).
        записная_книжка(фио("Ветров","Сергей","Андреевич"),
                "33-66-34",день_рождения("февраль", 5, 1986)).
        записная_книжка(фио("Варкова", "Дарья","Петровна"),
                "45-65-34",день_рождения("май", 22, 1993)).

        месяц_номер("январь", 1).
        месяц_номер("февраль",2).
        месяц_номер("март", 3).
        месяц_номер("май", 5).
clauses
        run():-  console::init(),
                список_дней_рожд().
```

10.2. Описание смешанных областей данных

Пусть в базе знаний хранятся такие факты:

```
владеет("Иванов", книга("Анна Каренина",
        автор("Лев", "Николаевич", "Толстой")).
владеет("Иванов", бытовая_техника(стиральная машина)).
владеет("Иванов", собака("Граф")).
```

и сделан запрос, чем владеет Иванов *владеет("Иванов", X)*, на который система должна выдать значения разного типа объектов: книгу, предметы бытовой техники, собаку и т.п. Возникает вопрос, как описать предикат *владеет(string, ?)*? Второй аргумент принадлежит сложной области, включающей разного типа объекты. Ниже рассмотрен пример описания такой области и связанных с ней предикатов.

```
/*Пример использования различных структур,
принадлежащих одной смешанной области*/
domains
        объект= книга(наименование, автор);
                бытовая_техника(наименование);
                собака(наименование);
                дом.
        автор = автор(фамилия, имя, отчество).
        фамилия= string. имя= string.
        отчество= string. наименование = string.
class predicates
        владеет:(наименование, объект) nondeterm anyflow.
clauses
        владеет("Иванов", книга("Анна Каренина",
                автор("Лев", "Николаевич", "Толстой"))).
        владеет("Иванов", бытовая_техника("стиральная машина")).
        владеет("Иванов", собака("Граф")).
        владеет("Иванов", дом).
clauses
        run():-  console::init(),
                владеет("Иванов", X),
                write(X), nl, fail.
        run().
```

В этом примере в области *объект* описаны через точку с запятой четыре различные структуры. Точка с запятой ";" рассматривается как союз "или", а описание объекта читается так: "*объект – это книга, или бытовая техника, или собака, или дом*". Каждый вариант ***составной***

области может быть только ***структурой*** с аргументами (*книга(наименование, автор))* или без них (*дом*).

10.3. Описание области с множественным типом данных

Так как в базе знаний описываются знания общего характера, применимые ко многим объектам предметной области, то параметрами предикатов-заголовков правил чаще всего являются данные разного типа: и числа, и строки, и списки, и структуры. В этом случае можно, используя структуры, создавать области с множественным типом данных. Ниже на примере показано, как это описать.

```
/*Пример описания области со смешанным типом данных*/
domains
        возраст = i(integer); r(real); s(string).
class predicates
        ваш_возраст:(возраст) nondeterm anyflow.
clauses
        ваш_возраст(i(Voz)):- write(Voz),nl.
        ваш_возраст(r(Voz)):- write(Voz),nl.
        ваш_возраст(s(Voz)):- write(Voz),nl.
        run():-   ваш_возраст(i(25)),
                  ваш_возраст(r(31.6)),
                  ваш_возраст(s("восемнадцать лет")),!.
        run().
```

В разделе "Списки" подобное описание используется для представления списков с элементами разных типов.

Контрольные вопросы

1. Что такое структура? В каких случаях она используется?
2. Какие встроенные предикаты позволяют программисту добавлять и удалять из базы знаний структуры-факты?
3. Какие структуры считаются равными?
4. Как происходит процесс унификации структур?
5. Как описать структуры?
6. Что такое смешанная область данных?

Контрольные задания

Переделать программу, разработанную в главах 2, 5 и 8 с использованием структур.

Глава 11. СОЗДАНИЕ ПРИЛОЖЕНИЙ В СРЕДЕ VISUAL PROLOG 7.0-7.3

11.1. Создание консольных приложений

Постановка задачи: разработать консольное приложение на языке *Visual Prolog*, предназначенное для вывода отчета о студенческих группах и о студентах с указанием их возраста.

План решения задачи.

- Создать новый проект *Project/New* в ***консольном*** режиме работы.
- Скомпоновать проект *Build/Build*.
- Наполнить файл реализации (*.pro) текстом программы на Прологе, изменив описания фактов и предикатов в соответствии с синтаксисом языка *Visual Prolog*.
- Скомпилировать и выполнить программу, используя команду *Build/Run in Window*.

Рассмотрим выполнение этих пунктов подробнее.

•***Создание нового проекта***. Выберем пункт *Project/New* и заполним диалоговое окно *Project Settings* (рис. 11.1).

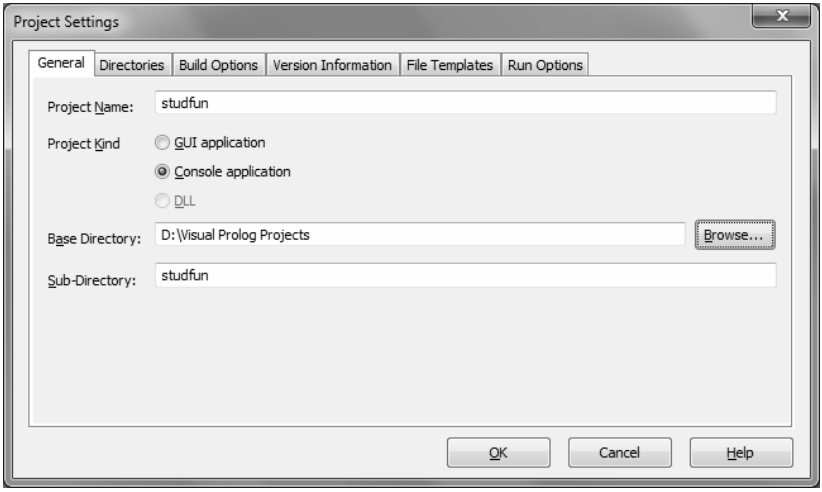


Рис. 11.1. Создание проекта с консольным вариантом интерфейса

Обратите внимание, что используется консольная стратегия, не GUI (графический интерфейс пользователя). Зададим имя проекта *studfun* и укажем каталог (*Base Directory*), где будут храниться файлы

проекта. Нажмем кнопку <OK> , в результате увидим окно с деревом проекта (рис. 11.2).

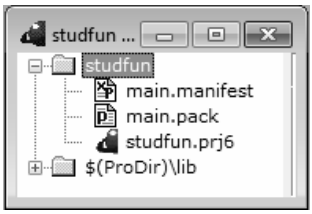


Рис. 11.2. Дерево проекта до его компоновки

• ***Сборка.*** Выберем пункт *Build/Build* из панели задач, чтобы внести прототип класса studfun*(main* в версии 7.3) в дерево проекта (рис. 11.3).

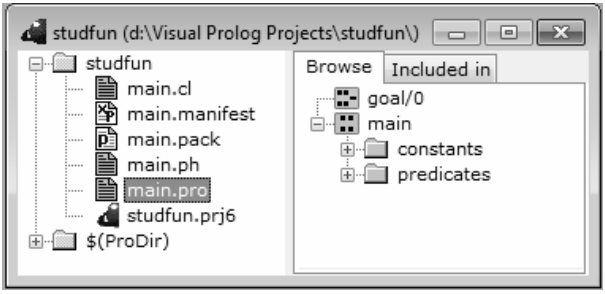


Рис. 11.3. Дерево проекта после его компоновки

• Отредактируем файл реализации *studfun.pro(main.pro в* версии 7.3)

```
implement studfun
        open core, console
constants
        className = "studfun".
        classVersion = "".
clauses
        classInfo(className, classVersion).
domains                 /*Описание типов данных*/
        фио = string. возраст = integer. группа = integer.
class facts             /*Описание фактов "группа"*/
        группа: (группа) nondeterm.
clauses                 /* Утверждения – факты */
        группа(7412).
        группа(743).
class facts             /*Описание фактов "студент"*/
```

```
        студент: (фио, группа) nondeterm .
    clauses                 /*Утверждения – факты*/
        студент("Иванов", 7412).
        студент("Задоя", 743).
        студент("Илларионов", 7412).
        студент("Петрова", 743).
    class facts             /*Описание фактов "возраст_студента"*/
        возраст_студента:(фио, возраст) nondeterm.
    clauses                 /*Утверждения – факты*/
        возраст_студента("Иванов", 19).
        возраст_студента("Задоя", 18).
        возраст_студента("Илларионов", 20).
        возраст_студента("Петрова", 19).
    class predicates        /*Описание предикатов*/
        списки_студентов_по_группам:() procedure.
    clauses                 /*Утверждения –правила*/
        списки_студентов_по_группам():-
                группа(Группа), nl,
                write("Список студентов группы №" ,Группа),
                nl, nl, студент(Фио, Группа),
                возраст_студента(Фио, Возраст),
                write(Фио, " возраст ", Возраст), nl, fail.
        списки_студентов_по_группам().
    class predicates        /*Описание предикатов*/
        список_групп:() procedure.
    clauses                 /*Утверждения – правила*/
        список_групп():-
                write("Список групп"),nl,
                группа(Y), write(Y), nl, fail.
        список_групп():- nl.
    clauses
        run():-   console::init(), список_групп(),
                  списки_студентов_по_группам().
    end implement studfun

    goal
        mainExe::run(studfun::run).
```

Снова откомпилируем программу и запустим её, используя команду *Build/Run in Window*. Чтобы протестировать консольную программу, следует использовать команду *Build/Run in Window*, а не *Execute*. На экране увидим следующий результат (рис. 11.4).

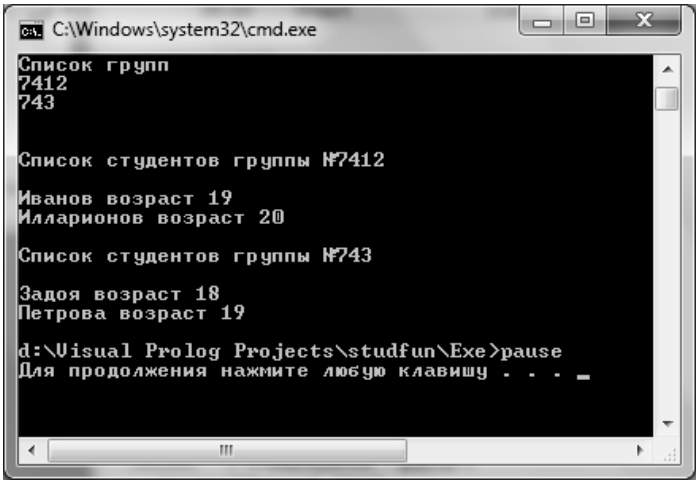


Рис. 11.4. Результат выполнения консольного приложения

Краткое пояснение к проделанным шагам

В языке Visual Prolog реализован объектно-ориентированный подход. Все понятия объектно-ориентированного программирования такие как: классы, объекты, интерфейсы, свойства, поля, методы, инкапсуляция, наследование – имеют место и в языке Visual Prolog.

При создании проекта в дереве проекта появляются директорий стандартных библиотек Пролога *$(ProDir)\Lib* и пакетный файл *studfun.pack (main.pack)* (контейнер) – это совокупность классов и интерфейсов, связанных с определенной тематикой решаемых задач.

В результате сборки (*Build/Build*) создается прототип класса *studfun(main)* и в дереве проекта появляются файлы, в которых этот класс должен быть описан:

- *studfun.cl(main.cl)* –файл, предназначенный для объявления класса. Он содержит описания типов (*domains*) и предикатов (*predicates*), доступных другим программным модулям. Если класс создается с возможностью генерации объектов, то в этом файле объявляются предикаты – конструкторы объектов класса;
- *studfun.pro(main.pro)* – файл, содержащий реализацию класса, т.е. в нем имеются как описания предикатов, так и утверждения, реализующие эти предикаты;
- *studfun.ph (main.ph)* – заголовочный файл пакета.

В рассматриваемом примере вся программа размещается в файле реализации *studfun.pro (main.pro).* После заголовка ***implement*** *studfun* в

предложении ***open*** *core, console* выполняется подключение классов, в частности, объявленных в *PFC* – библиотеке фундаментальных классов Пролога, с использованием которых могут быть созданы различные приложения на Прологе. В этом предложении мы добавляем класс *console*, в котором описаны предикаты *write(), nl* и т.п. Чтобы узнать, какие предикаты класса объявлены доступными, нужно в дереве проекта последовательно раскрыть узлы *$(ProDir)/pfc/console/console.cl* и найти описание нужного предиката.

Объекты и классы

Если предикат или факт предполагается использовать только внутри реализации класса, он объявляется как *class predicates* – предикат класса или *class facts*. Примеры:

```
class predicates
        наибольшее:(возраст, возраст, возраст) nondeterm (i,i,o).
class facts
        возраст_студента:(фио, возраст) nondeterm.
```

Существует и более важное назначение префикса *class* . Оно связано с понятиями классов и объектов, создаваемых классами. Классы бывают двух видов: классы, которые создают объекты, и классы без возможности генерации объектов. В определении первого типа классов (с генерацией объектов) обязательно присутствует файл интерфейса **.i*. В нем описываются все элементы объектов (типы, предикаты и т.п.), доступные другим сущностям проекта. В заголовочном файле **.cl* описываются конструкторы объектов. Для классов второго типа файл интерфейса не нужен, все публичные элементы описываются в заголовочном файле **.cl*.

Если факт несет информацию об объекте, для каждого объекта свою, то он описывается в разделе *facts (*факт объекта*)* файла реализации **.pro*. Если факт несет информацию, необходимую и доступную всем объектам класса, то он должен быть описан в разделе *class facts (*факт класса*)* файла реализации. Все созданные объекты могут пользоваться информацией, хранимой в этом факте, а также изменять эту информацию. Предикаты объекта описываются в интерфейсе *.i, а предикаты класса - в заголовочном файле *.cl. Это правило без исключения: невозможно объявить объектный предикат в классе и предикат класса в интерфейсе. Интерфейсы представляют объектные типы.

Объявления

Синтаксис описания факта или предиката может быть задан в виде:

```
<имя предиката>:(<тип аргумента1>,<тип аргумента2>,
        <тип аргумента n>) <режим детерминизма > < шаблон потока>.
```

Определение предиката неполно без указания типов аргументов, режима детерминизма предиката и (flow pattern) – шаблонов потока. Например:

```
class facts
        студент: (фио, группа) nondeterm anyflow.
```

Описание факта сообщает, что факт имеет два аргумента: первый является фамилией и относится к типу string, второй – номером студенческой группы и относится к типу integer. Режим детерминизма nondeterm говорит о том, что согласование предиката с утверждениями базы знаний может завершиться неудачно или успешно многими способами. Ключевое слово anyflow определяет возможность любого потока данных: оба аргумента или каждый из них могут быть как входными (i), так и выходными (o) величинами. Входные величины – это константы или конкретизированные переменные, выходные - неконкретизированные (свободные) переменные.

Вообще, необязательно предоставлять объявление режима детерминизма. Компилятор самостоятельно сделает выводы о режимах предиката. Если он не сможет это сделать, он выдаст сообщение об ошибке, и тогда можно добавить объявление режима самим.

Режимы детерминизма

Чтобы объявить, имеет ли предикат при согласовании с базой знаний одно решение или несколько, используются следующие ключевые слова:

- ***determ*** **–** предикат может завершиться неудачно (*fail*) или успешно (*succeed*) с одним решением;
- ***procedure*** **–** этот вид предикатов всегда завершается успешно и имеет одно решение. Предикаты *списки_студентов_по_группам(), список_групп()* - процедуры и могут быть объявлены так:

```
class predicates
        списки_студентов_по_группам:() procedure.
        список_групп:() procedure.
```

- ***multi*** **–** предикат не может завершиться неудачно и имеет множество решений;
- ***nondeterm*** **–** предикат может завершиться неудачно или успешно многими способами. Факты *студент* и *возраст_студента* - оба *nondeterm* и имеют следующее объявление:

```
class facts
        студент: (фио, группа) nondeterm .
        возраст_студента:(фио, возраст) nondeterm.
```

Обратите внимание на предикат *run()*. Он объявлен как предикат класса *studfun,* и в разделе ***goal*** осуществляется обращение к нему. В

тело этого предиката можно добавлять предикаты вашей программы для тестирования. Можно рекомендовать изменить этот предикат следующим образом:

```
clauses
    run():-  console::init(),
             <тестируемые предикаты>,
             <вывод результатов>, fail.
    run().
```

Например:

```
clauses
    run():-  console::init(),
             список_групп(),
             списки_студентов_по_группам(), fail.
    run().
```

Шаблоны потоков

Шаблоны потоков указывают для каждого аргумента предиката, является ли он входным или выходным, например шаблон (i,i,o) сообщает, что первый и второй аргументы являются исходными данными, а третий аргумент – результат. Входные аргументы (i) должны быть константами или конкретизированными переменными, а выходные (o) – неконкретизированными переменными. Допустимо указывать для предиката несколько возможных вариантов шаблонов потока следующим образом:

```
возраст_в_группе:(группа, фио, возраст) nondeterm (i,o,o) (o,i,o)
    (o,o,o) (o,o,i).
```

Создать консольное приложение несложно, так что приступайте к реализации своего проекта. Если вам понадобятся предикаты работы со строками, то к файлу реализации подключите класс ***string***. Напомним, что недостающие для решения задачи предикаты можно попробовать найти в узле дерева проекта ***pfc*** (среди базовых классов Пролога) и подключить (***open)*** соответствующий класс к файлу реализации.

11.2. Создание приложений с использованием графического интерфейса. Управление с помощью меню главного окна задач

В качестве примера приложения, работающего с базой данных, рассмотрим задачу разработки справочной системы по аптекам города. Суть задачи в следующем. В базе данных хранится информация об аптеках города, справочные данные по лекарствам и их назначении, а также информация, в какой аптеке какое лекарство имеется, в каком количестве, цена продажи и срок годности этого лекарства. Приложе-

ние должно обеспечивать ответы на следующие запросы: "Какие лекарства есть в определенной аптеке?", "В каких аптеках имеется заданное лекарство и по какой цене?", "Какова минимальная цена на определенное лекарство и в какой аптеке?", "Где расположена аптека и какой у нее телефон?" и т.п.

Информация об аптеках, лекарствах и их наличии хранится в виде фактов в отдельном файле "*db.txt*", называемом динамической базой данных. Ниже приведен сокращенный вариант текста этого файла.

```
clauses
        apteka(17, "Есенина 44", "11-22-76").
        apteka(24, "Новая 31", "22-22-76").
        apteka(78, "Гагарина 12", "33-22-76").
        apteka(137, "Циолковского 11", "44-22-76").
        apteka(200, "Ленина 45", "55-22-76").
        apteka(18, "Есенина 45", "11-22-80").

        lekarstvo(1, "Активированный уголь","укрепляющее").
        lekarstvo(2, "Колдрекс", "жаропонижающее").
        lekarstvo(3, "Аскорбиновая кислота", "Витамин").
        lekarstvo(4, "Аспирин", "жаропонижающее").
        lekarstvo(5, "Витамин А", "Витамин").

        apteka_lek(17, 1, 10, 30, "07.04.01").
        apteka_lek(17, 2, 20, 130, "07.03.01").
        apteka_lek(17, 3, 10, 50, "07.03.23").
        apteka_lek(24, 5, 10, 5, "08.06.01").
        apteka_lek(78, 4, 10, 150, "05.04.01").
        apteka_lek(78, 5, 10, 70, "07.01.01").
        apteka_lek(137, 1, 10, 30, "06.07.01").
        apteka_lek(200, 3, 10, 30, "08.05.01").
        apteka_lek(18, 3, 5, 23, "09.08.01").
```

Факты имеют следующую структуру:

```
apteka(<номер аптеки>, <адрес аптеки>, <телефон>).

lekarstvo(<шифр лекарства>, <название лекарства>,
        <фармакологическая группа>).

apteka_lek(<номер аптеки>, <шифр лекарства>, <количество>,
        <цена>, <срок годности(год.месяц.день)>).
```

В динамическую базу данных можно добавлять факты, а также их удалять и изменять.

Назначение приложения – извлекать из базы данных нужную пользователю информацию. Файл с базой данных подгружается к работающей программе в момент ее выполнения.

Рассмотрим примеры правил, обеспечивающих выполнение требуемых запросов.

Пример 11.1. Получить список аптек с их адресами и телефонами.

```
class predicates
        all_apt:().
clauses
        all_apt():-
                stdIO::write("\nАптеки города\n"),
                stdIO::write("\nНомер аптеки Адрес Телефон\n\n"),
                apteka(Napt, Adress, Tel),
                stdIO::writef("% % %\n",Napt, Adress, Tel),
                fail.
        all_apt():-stdIO::write("\nСписок закончен\n").
```

Здесь предикат *stdIO::write("")* выполняет вывод данных в окно сообщений (*Messages*).

Пример 11.2. Получить список лекарств с указанием наименования и группы.

```
class predicates
        all_lek:().
clauses
        all_lek():-
                stdIO::write("\nСписок лекарств\n"),
                stdIO::write("\nШифр Наименование Группа\n\n"),
                lekarstvo( Sh, Name, Tip),
                stdIO::writef("% % %\n",Sh, Name, Tip),
                fail.
        all_lek():-
                stdIO::write("\nСписок закончен\n").
```

Пример 11.3. В каких аптеках есть нужное лекарство и по какой цене?

```
class predicates
        lek_apteka_cena:(string Lek, integer Napt, string Adress,
                string Tel, real Cena) nondeterm (i,o,o,o,o) (o,i,o,o,o).
clauses
        lek_apteka_cena(Lek, Napt, Adress, Tel, Cena):-
                lekarstvo(Sh, Lek, _),
                apteka_lek( Napt, Sh, _, Cena,_),
                apteka( Napt, Adress, Tel).
```

Пример 11.4. Какие лекарства, в каком количестве и по какой цене имеет аптека?

```
class predicates
        apteka_lek_cena:( integer Napt, string Lek, integer Kol, real Cena)
                nondeterm (i,o,o,o) (o,i,o,o).
clauses
        apteka_lek_cena(Napt,Lec, Cena, Kol):-
                apteka( Napt, _, _),
                apteka_lek(NApt,Sh,Kol,Cena,_),
                lekarstvo(Sh,Lek,_).
```

Работа программы происходит следующим образом. При запуске на экране появляется главное окно задач (*TaskWindow*), внутри него находится окно сообщений (*Messages*). Управление выполнением программы осуществляется с помощью главного меню, в котором есть четыре пункта: "Загрузка базы данных", "Запросы", "Window" и "Help" (см. рис. 11.5).

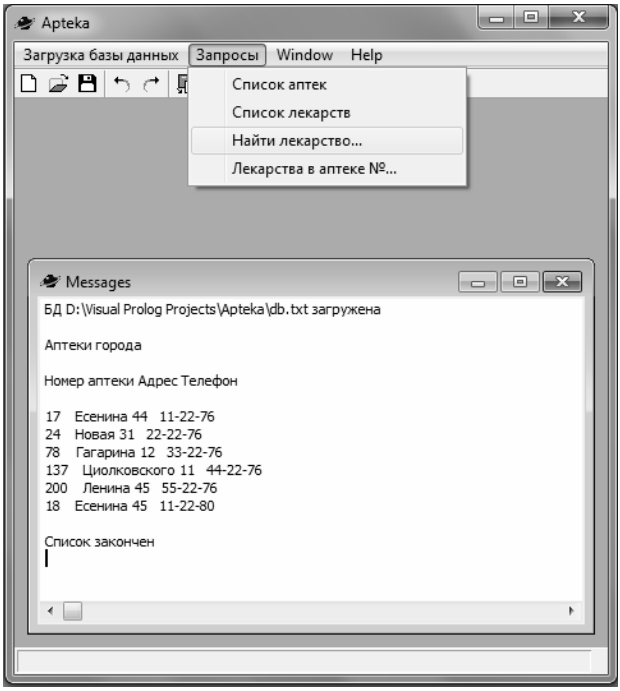


Рис. 11.5. Информационная система "Аптеки города"

Пункт "Загрузка базы данных" используется для загрузки файла с динамической базой данных в область видимости программы, что и должен выполнить пользователь в первую очередь. Затем он может получать нужную ему информацию из базы с помощью запросов.

Список доступных пользователю запросов – это подпункты меню "Запросы": Запросы|Список лекарств, Запросы|Аптеки города, Запросы|Найти лекарство..., Запросы|Лекарства в аптеке. Для выполнения некоторых из них пользователь должен ввести дополнительную информацию. Он это сможет сделать с помощью диалоговых окон. Пункты "Window", "Help" стандартные, "Window" позволяет управлять окнами приложения, "Help" дает возможность получить справочную информацию.

11.2.1. **Создание проекта**

Запускаем программную среду Visual Prolog. В главном меню программы открываем вкладку Project (Проект) и в появившемся подменю выбираем пункт New… (Новый). Появляется окно Project Settings (Установки проекта). В данном окне необходимо ввести имя проекта в поле Project Name (Имя проекта), например имя Apteka.

В поле Project Kind (Тип проекта) выбираем модель GUI (графический интерфейс пользователя) – объектно-ориентированная модель. После нажатия кнопки OK создается проект и появляется окно, в котором показана структура проекта в виде папок и файлов (рис. 11.6).

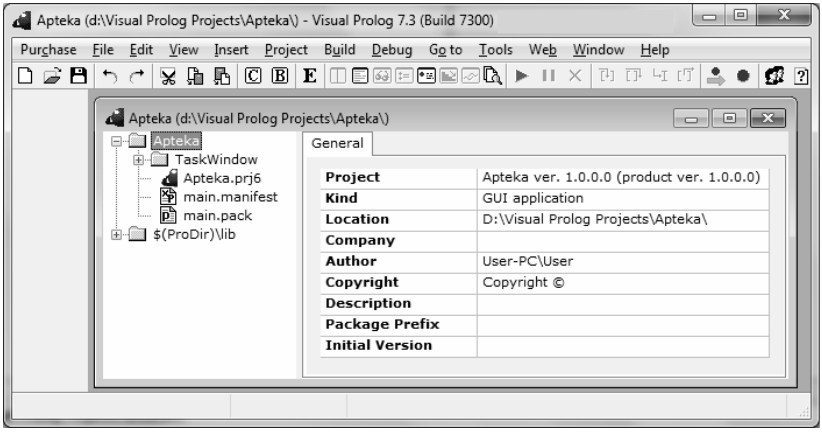


Рис. 11.6. Создание проекта.

Среда разработки создает начальный набор модулей, необходимых для работы с GUI, а также компоненты GUI: главное меню, инструментальная панель вверху, панель статуса внизу, диалог "О программе" и главное Окно Задач (Task Window) программы.

Сразу после того, как мы создали проект таким способом, можно компилировать его и получить пустую GUI программу (см. рис. 11.7). В действительности на этой стадии, программа ничего не будет делать. Однако она будет обладать всей базовой функциональностью, которая должна быть в GUI программе. Visual Prolog создает основной скелет работающей GUI программы и снабжает ее некоторыми наиболее часто требующимися функциональными возможностями.

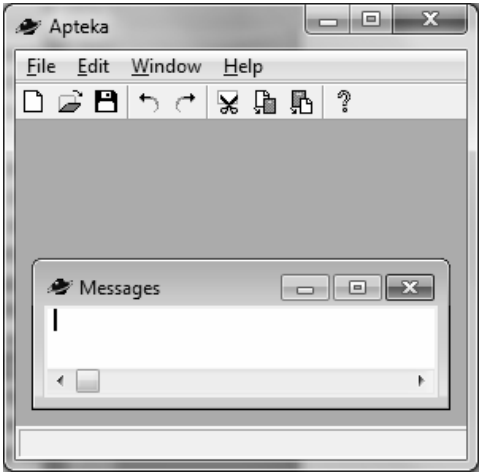


Рис. 11.7. Скелет работающей GUI программы

Например: внутри главного окна приложения (называемого здесь Окно Задач (Task Window)) Visual Prolog создает другое окно с заголовком Messages (сообщения). Это окно используется внутри, чтобы действовать как консоль. Если мы используем в программе предикат *stdio::write*(...), то вывод будет направлен в это окно Messages.

При запуске скомпилированной GUI программы на этой стадии, вы можете перемещаться по пунктам меню, изменять размеры главного окна (Task Window), изменять размеры окна сообщений (Messages). Щелкните правой кнопкой мыши на любом месте внутри окна сообщений – откроется небольшое всплывающее меню, с помощью которого можно очистить содержимое окна сообщений (Messages) и выполнить некоторые другие действия.

Но в данный момент простая GUI программа не имеет никакой логической функциональности, которая необходима. Нужно проделать дальнейшую работу, чтобы получить эту функциональность.

Прежде, чем начинать разработку логики программы, нужно создать и/или изменить некоторые графические управляющие (GUI) компоненты.

Все GUI компоненты хранятся как отдельные файлы ресурсов во время фазы кодирования. В большинстве других языков программирования эти файлы ресурсов отдельно компилируются и затем включаются в основной код во время процесса связывания. Visual Prolog выполняет компиляцию ресурсов и решает вопросы связывания автоматически без вмешательства пользователя.

11.2.2. **Создание модального диалога**

Добавим еще один GUI компонент, который впоследствии понадобится в программе. Это диалоговое окно, которое будет использоваться при вводе наименования лекарства, необходимого для дальнейшей обработки его в программе. В дереве проекта (Project Tree) (где все модули и ресурсы приведены в виде дерева) сделаем правый клик на Task Window и в появившемся контекстном меню выберем пункт New in New Package… (см. рис. 11.8).

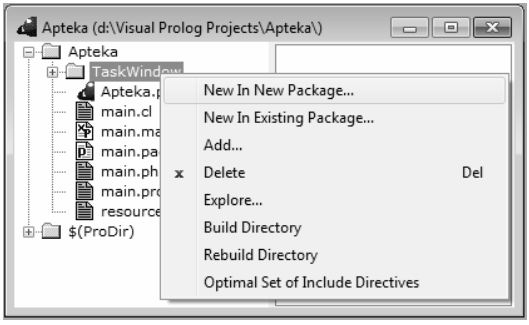


Рис. 11.8. Создание нового элемента проекта

Появится диалог Create Project Item (создать элемент проекта). В левом столбце выберем Dialog, а в поле Name введем lekarstvo (см. рис. 11.9).

Будет создан стандартный диалог для дальнейшего редактирования. В виду того, что не надо описывать функцию помощи для этого диалога, кликнем на кнопке Help (специальном GUI компоненте) и удалим ее нажатием клавиши Del.

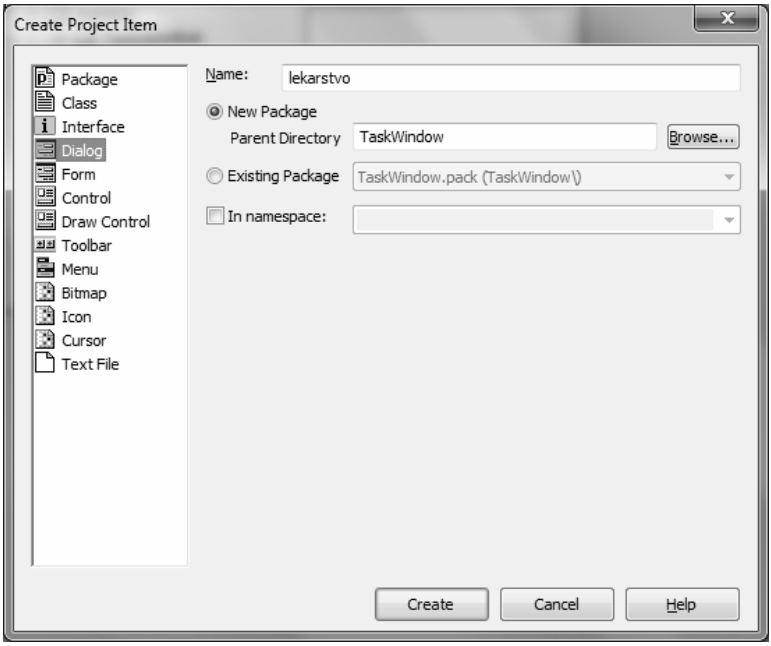


Рис. 11.9. Создание диалога lekarstvo

Оставшиеся две кнопки необходимо выбрать и переместить в диалоговом окне, чтобы сделать его вид более презентабельным. Конечный результат будет приблизительно таким, как показано на рис. 11.10.

Используя элементы для редактирования диалога, добавим статический текст (Static Text). (Слово статический (static) обозначает не переключаемый GUI элемент). Выбрав кнопку создания статического текста (см. рисунок 11.11), следует поместить этот элемент на форму "lekarstvo", в прямоугольной зоне появится текст (как это показано на рис. 11.12).

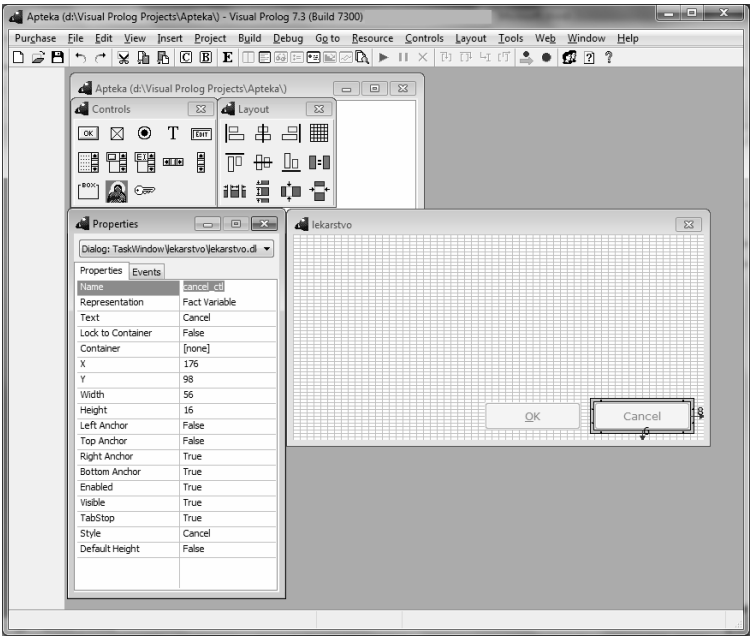


Рис. 11.10. Окно диалога lekarstvo и его свойства

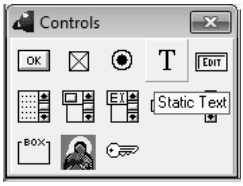


Рис. 11.11. Панель элементов управления

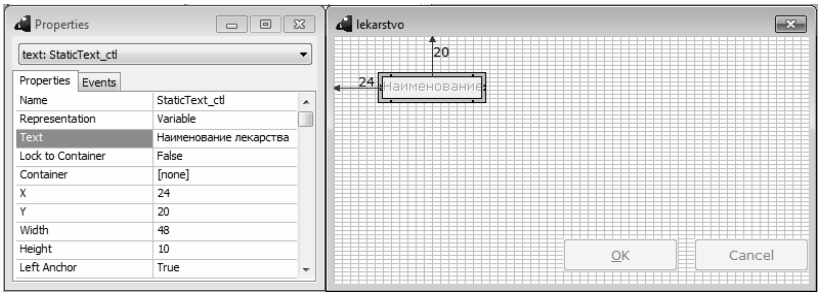


Рис. 11.12. Создание статического текста

Отпустив кнопку мыши, получим на форме новый элемент "Static text", его свойства будут представлены в панели свойств "Properties". В поле Text этой панели напишем "Наименование лекарства". Этот текст появится внутри диалога.

Подобным же образом будем использовать управляющие элементы диалога для того, чтобы добавить поле ввода или элемент редактирования, как показано на рис. 11.13 и рис. 11.14.

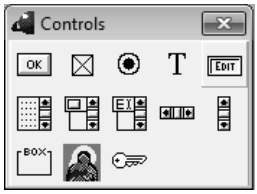


Рис. 11.13. Выбор поля редактирования

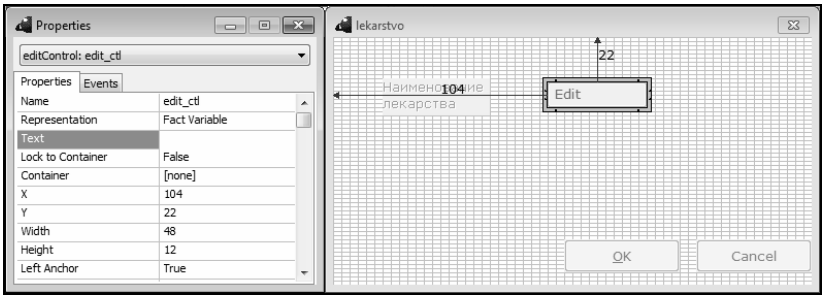


Рис. 11.14. Создание поля ввода и редактирования текста

На этот раз оставим поле Text незаполненным. Имя компонента edit_ctl будет использоваться в программном коде, чтобы ссылаться на данное поле как на фактическую переменную.

В итоге наш диалог выглядит, как показано на рис.11.15.

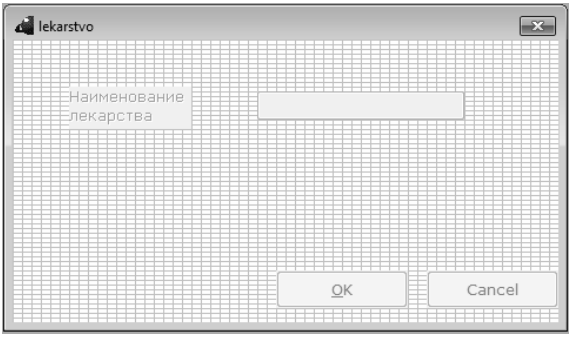


Рис. 11.15. Окно диалога lekarstvo

Остался еще один несделанный шаг. У каждого диалога есть свойство, известное как Visit Order (порядок переключения). Это порядок, по которому компоненты, взаимодействующие с пользователем и находящиеся в диалоговом окне, получают фокус, когда пользователь нажимает клавишу TAB. Фраза "компонент, получающий фокус ввода" подразумевает, что указанный объект немедленно начинает принимать ввод с клавиатуры. Например, если редактируемое поле получило фокус ввода, то в нем можно будет заметить мигающий курсор, а само поле будет готово принимать символы, вводимые с клавиатуры.

Для того чтобы поменять порядок переключения, щелкнем правой кнопкой мыши на окне диалога, и выберем Visit Order в появившемся контекстном меню.

На некоторых GUI компонентах появятся небольшие кнопки (как показано на рис. 11.16). Как видно, элемент редактирования текста (edit_ctl) имеет порядковый номер 3, тогда как порядковый номер кнопки OK – 1.

Это значит, что когда диалог предстает перед пользователем, окно редактирования текста не будет тем самым компонентом, который станет немедленно принимать команды с клавиатуры. Если пользователь нажмёт клавишу TAB, фокус переместится на кнопку Cancel, и только после нажатия TAB еще раз фокус перейдет на компонент редактирования, и программа будет воспринимать ввод с клавиатуры.

Можно сказать, что поле редактирования имеет самый последний порядок посещения по отношению к другим компонентам в диалоге.

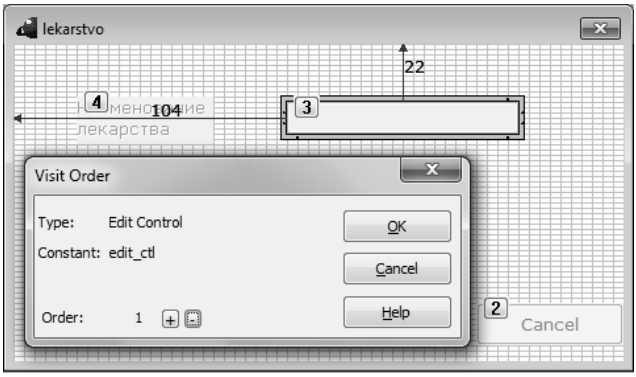


Рис. 11.16. Изменение порядка переключения элементов формы

Чтобы изменить это, щелкнем мышкой на маленькой кнопке с цифрой 3. В результате откроется окно диалога Visit Order (как показано

на рис. 11.16). Можно изменить порядок посещения компонентов, используя кнопки "+" и "–". В нашем случае установим порядок так, чтобы (edit_ctl) имел номер 1. Сейчас, когда диалог будет использоваться в программе, фокус ввода с самого начала будет расположен на компоненте редактирования.

Обратите внимание, что в диалогах есть другое свойство, называемое default push button (кнопка по умолчанию), которое можно установить отдельно от порядка переключения. Событие для кнопки, назначенной кнопкой по умолчанию, будет вызываться при нажатии клавиши ENTER независимо от того, у какого компонента находится фокус ввода. Обычно кнопкой по умолчанию назначается OK.

По умолчанию тип диалога (Type) задан Modal. Термин modal (модальный) отражает тот факт, что когда бы ни был представлен пользователю данный диалог, GUI отменит возможность обращения к другим частям программы на то время, пока диалог является видимым. Только когда диалог закрывается пользователем (при нажатии OK или Cancel), графический интерфейс пользователя снова станет доступным для любых видов действий.

Немодальный диалог (modeless) напротив таких ограничений не делает. Весь GUI интерфейс доступен, когда такой диалог активен. Программирование немодального диалога несколько более сложно.

Используя окно свойств диалога изменим заголовок окна: введем в поле Title текст "Найти лекарство...". В результате получим следующее диалоговое окно (см. рис. 11.17).

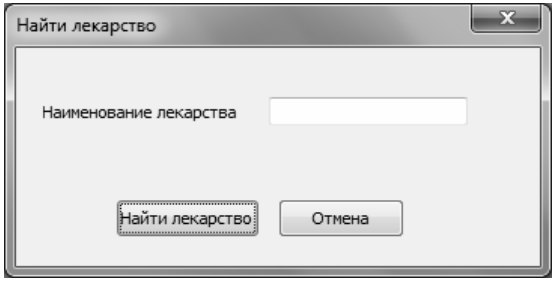


Рис. 11.17. Диалоговое окно "Найти лекарство"

Подобным же образом создадим диалоговое окно ввода номера аптеки (apteka_nomer), для которой мы хотим получить информацию о наличии всех лекарств (см. рис. 11.18).

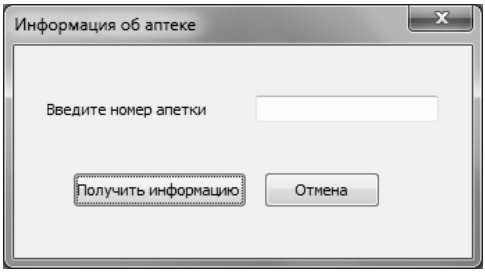


Рис. 11.18. Диалоговое окно "Информация об аптеке"

В данный момент нужно скомпилировать проект.

11.2.3. **Изменение меню**

Сейчас давайте изменим главное меню программы. Как было замечено раньше, программная среда Visual Prolog предоставляет стандартное меню, состоящее из некоторых стандартных компонентов, которые часто используются в программах. Нам нужно переделать это меню, чтобы оно соответствовало функциональным возможностям нашей программы. Используя дерево проекта, выполним двойной щелчок на пункте TaskMenu.mnu.

В результате запустится редактор меню. Главный диалог редактора представлен на рис. 11.19.

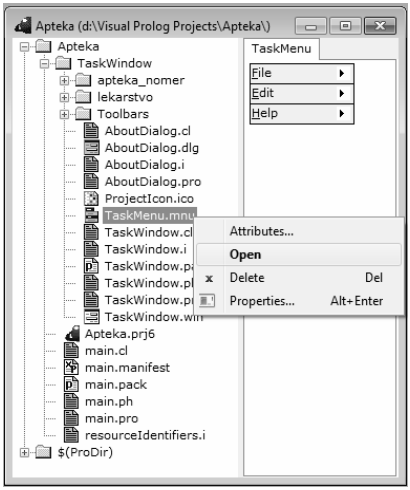


Рис. 11.19. Редактирование меню

После этого откроется диалог TaskMenu (см. рис. 11.20). Мы изменим имя пункта меню с &Edit на &Запросы. Знак амперсанда (&) перед *З* обозначает букву, которая будет подчеркнутой в меню. Пользователи могут использовать эту букву для быстрого доступа к нужному пункту меню. Для проверки этой возможности, щелкнем по кнопке Test (T) на панели инструментов. Вместо главного меню программной среды появится меню, которое мы разрабатываем. Затем можно просмотреть меню и проверить его. Нажав на ESC, вернем меню в первоначальное состояние.

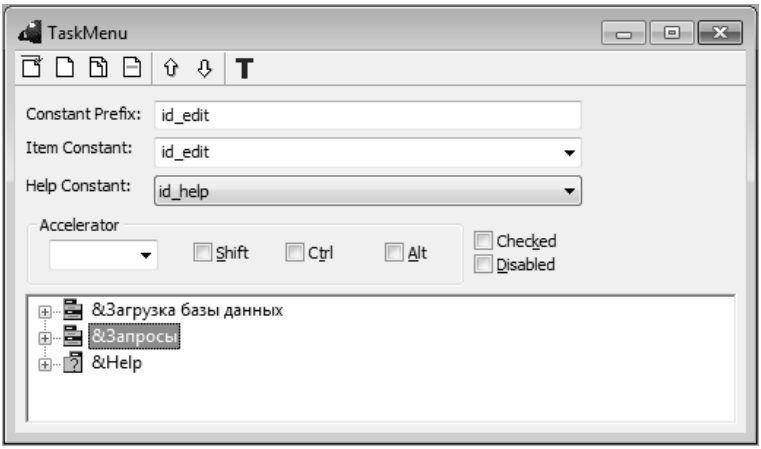


Рис. 11.20. Окно редактирования Меню программы – TaskMenu

Вернемся обратно к главному диалогу TaskMenu. Сделав двойной щелчок на пункте &Запросы, увидим, что он содержит элементы старого меню Edit (Undo, Redo, Cut, Copy и Paste). Удалим все элементы, какие присутствуют в главном пункте (&Запросы). Также изменим Constant Prefix, установив его в id_query. Constant Prefix используется внутри программной среды при создании констант, представляющих различные элементы меню. Эти константы будут использоваться в коде для ссылок на элементы меню.

Добавим некоторые элементы меню под главным пунктом Запросы. Для этого выберем *New SubMenu* (новое подменю) и внесем информацию, представленную на рис. 11.21.

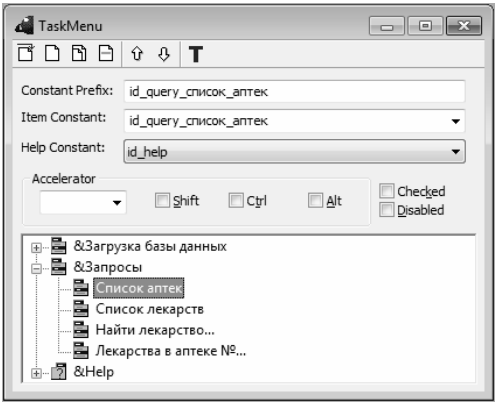


Рис. 11.21. Создание подпунктов меню Запросы (SubMenu)

Отметим, что Constant создается автоматически в зависимости от текста, который вводится для пункта меню. В нашем примере это будет id_query_список_аптек, как видно на предыдущем рисунке.

По соглашению "три точки" используются в конце таких пунктов меню, которые будут вызывать другой диалог. В нашем примере "&Список_лекарств" и "&Аптеки_города" – пункты меню, не содержащие "точек" в отличие от пункта "&Найти_лекарство... ". При вызове этого пункта меню, откроется диалог, в котором надо будет задать наименование лекарства. Аналогично нужно изменить первый пункт имеющегося меню (рис. 11.22).

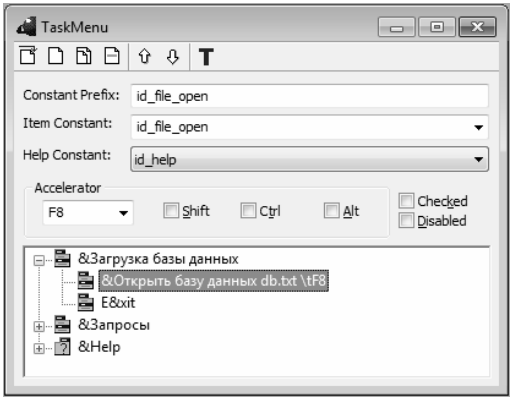


Рис. 11.22. Подготовка пункта меню "&Открыть базу данных"

Остается последний шаг в редактировании меню TaskWindow. По умолчанию, когда создается это меню, пункт *Загрузка базы данных|Открыть базу данных* недоступен. Надо расположить его в определенном месте и сделать доступным, удалив галочку из поля Disabled диалога для этого пункта, как показано на рис. 11.22.

Для пункта меню может быть установлен акселератор (т.е. горячая клавиша), с помощью которого пользователь будет быстро обращаться к той же функции, что и в пункте меню. В примере на рис. 11.22 это клавиша F8.

После того как закроем диалог TaskMenu, необходимо ввести подтверждение, действительно ли нужно сохранить меню. Выберем Save(Сохранить) (рис. 11.23).

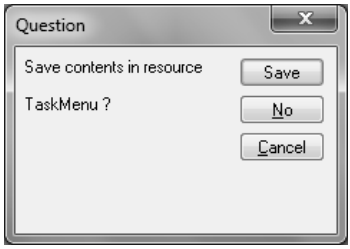


Рис. 11.23. Сохранение проекта меню

Скомпилируем и выполним программу. В результате получим приложение со всеми необходимыми пунктами меню. Пока при выборе любого из пунктов не выполняются никакие действия, так как не определена логика программы.

11.2.4. **Изменение панели инструментов**

Панель инструментов приложения – это еще один полезный GUI компонент. Обычно, он содержит кнопки, отображающие некоторые из функций меню. Эти кнопки работают так же как горячие клавиши в меню. Мы сейчас займемся редактированием панели инструментов. При создании проекта автоматически создается стандартная панель инструментов для программы. Ее можно изменить под свои функции. В дереве проекта (Project Tree) выполним двойной клик на пункте ProjectToolbar.tb (см. рис. 11.24)

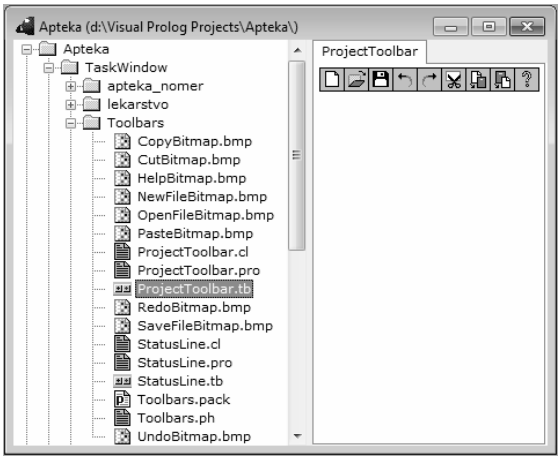


Рис. 11.24. Редактирование панели инструментов

В результате запустится редактор панели инструментов. В верхней части окна представлена панель, которую надо редактировать, а в нижней – различные элементы, доступные для редактирования ее частей.

В панели инструментов имеется набор кнопок с предопределенными изображениями (какие обычно приняты в GUI программах). Если есть желание, то можно изменить эти иконки, но в данном пособии мы не будем вдаваться в такие подробности. Следует упомянуть здесь, что Visual Prolog содержит неплохую маленькую программку для редактирования иконок. Для средних и больших изображений Visual Prolog открывает MS Paint.

Кнопки были спланированы для набора функций из меню. Но так как мы изменили меню, то нам также необходимо изменить кнопки на панели инструментов и поместить их на места.

Для начала, необходимо удалить кнопки для вырезки и копирования, т.к. у нашей программы отсутствуют такие возможности. Для этого выделим эти кнопки и удалим их с панели, нажав Del . В результате она должна выглядеть вот так (см. рис. 11.25).

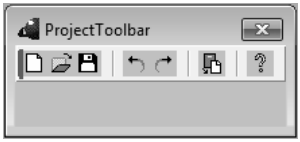


Рис. 11.25. Панель инструментов

Мы должны изменить кнопки Undo, Redo, Paste и Help так, чтобы они представляли следующие пункты меню: Запросы|Аптеки города, Запросы|Список лекарств и Запросы|Найти лекарство ..., Запросы|Лекарства в аптеке № ...(как было замечено ранее, мы не будем изменять иконок для этих кнопок).

Выполним двойной клик на кнопке Undo и в появившемся диалоге (см. рис. 11.26) атрибутов кнопок (Button Attributes) изменим константу Constant, которая представляет кнопку панели, с id_edit_undo на id_query_список_аптек.

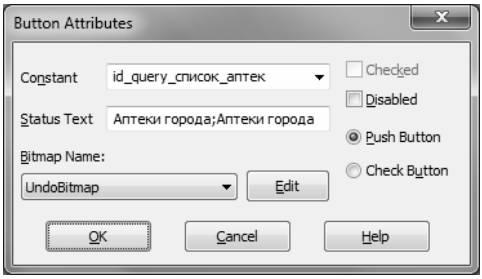


Рис. 11.26. Диалоговое окно атрибутов кнопок

В том же самом диалоге вам надо изменить Status Text с:
Undo; Undo на ***Аптеки города; Аптеки города***

Точка с запятой в строке разбивает ее на 2 части. Первая часть отображается на кнопке, а вторая – появится в строке статуса главного окна.

Подобным же образом, изменим Constant для кнопки Redo на значение id_query_список_лекарств. И Constant для кнопки Paste должно иметь значение id_query_найти_лекарство. Строку статуса для этих 2-х кнопок тоже надо изменить подходящим образом.

Скомпилируем и выполним программу, проверим, изменилась ли панель инструментов.

11.2.5. **Ввод основного кода в программу**

Мы завершили всю работу, связанную с графической функциональностью программы. Сейчас перейдем к написанию кода. Перед этим давайте уясним разницу между способом программирования, использовавшимся прежде, и тем, каким он должен быть в GUI приложении.

Что это значит для программиста? Раньше код содержался внутри одного модуля. Сейчас он находится в нескольких модулях, в зависи-

мости от логики, которая контролируется сообщающимися GUI компонентами.

Давайте рассмотрим некоторый базовый код, который нужен в программе. Откроем TaskWindow.pro и поместим курсор сразу после строчки:

```
classInfo(className, classVersion).
```

после этого добавим следующий код:

```
facts
        currentDir:string:="".
domains
        class facts - aptekaDB
        apteka:(integer Napt, string Adress, string Tel).
        lekarstvo:(integer Sh, string Name, string Tip).
        apteka_lek:(integer Napt, integer Sh, integer Kol, real Cena,
                string Dat).
class predicates
        reconsult:(string Filename).
clauses
        reconsult( Filename):-
                retractFactDB(aptekaDB),
                file::consult(Filename, aptekaDB).
class predicates
        all_apt:().
clauses
        all_apt():-
                stdIO::write("\nАптеки города\n"),
                stdIO::write("\nНомер аптеки Адрес Телефон\n\n"),
                apteka( Napt, Adress, Tel),
                stdIO::writef("% % %\n",Napt, Adress, Tel),
                fail.
    all_apt():-
                stdIO::write("\nСписок закончен\n").
class predicates
        all_lek:().
clauses
        all_lek():-
                stdIO::write("\nСписок лекарств\n"),
                stdIO::write("\nШифр Наименование Группа\n\n"),
                lekarstvo( Sh, Name, Tip),
                stdIO::writef("% % %\n",Sh, Name, Tip),
                fail.
    all_lek():-
                stdIO::write("\nСписок закончен\n").
```

```
class predicates
        lek_apteka_cena:(string Lek, integer Napt, string Adress,
                string Tel, real Cena) nondeterm
                (i,o,o,o,o) (o,i,o,o,o).
clauses
        lek_apteka_cena(Lek, Napt, Adress, Tel, Cena):-
                lekarstvo(Sh, Lek, _),
                apteka_lek( Napt, Sh, _, Cena,_),
                apteka( Napt, Adress, Tel).
class predicates
        apteka_lek_cena:( integer Napt, string Lek, integer Kol, real Cena)
                nondeterm (i,o,o,o) (o,i,o,o).
clauses
        apteka_lek_cena(Napt,Lek, Kol, Cena):-
                apteka( Napt, _, _),
                apteka_lek(NApt, Sh, Kol, Cena,_),
                lekarstvo(Sh, Lek, _).
```

Добавленный код – это логическое ядро программы. В данном случае, мы вставляем его прямо в модуль TaskWindow.pro, потому что хотим, чтобы было видно, как компоненты графического интерфейса (GUI – компоненты) обращаются к логическому ядру для выполнения различных действий. В более сложных примерах, логическое ядро будет находиться отдельно, иногда располагаясь в нескольких модулях. GUI модули (такие как TaskWindow и т.д.) будут в этом случае ссылаться на эти модули отдельно путем расширения области их действия.

В действительности, это хороший стиль – держать логическое ядро программы отдельно от модулей графического интерфейса.

11.2.6. **Инкапсуляция интерактивного кода**

Теперь, когда мы создали логическое ядро программы, нам надо добавить интерактивные части. Сделаем правый щелчок в дереве проекта (Project Tree) на TaskWindow.win. Этот пункт представляет ресурс для главного Окна задач (TaskWindow). Все щелчки, события меню и т.д., происходящие внутри этого окна, должны будут перехватываться обработчиками, написанными для него. Выполнив щелчок правой кнопкой мыши в выделенной области, мы увидим небольшое контекстное меню, как показано на рис. 11.27. Выберем в нем пункт Code Expert.

Эксперт окон и диалогов (Dialog and Window Expert) поможет установить обработчики по умолчанию для элементов управления (интерактивных GUI компонентов) обычного окна или диалога. Закрашен-

ный синий кружок показывает, что для данного элемента нет обработчика. Зеленая галочка означает его наличие (см. рис. 11.27).

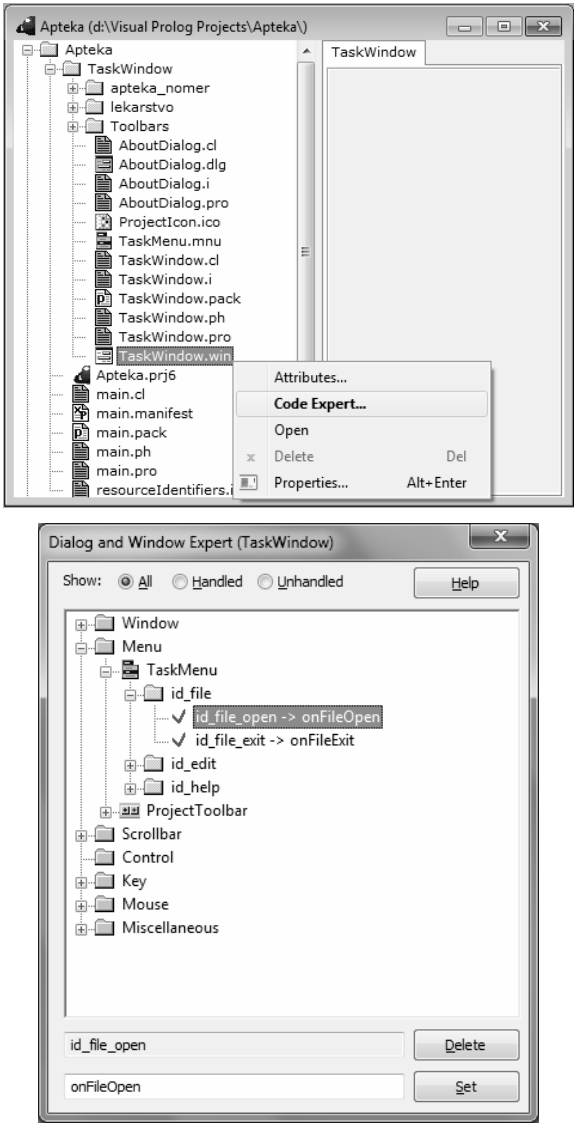


Рис. 11.27. Вызов эксперта окон и диалогов

Используя диалог, убедимся, что для пункта меню, представленного константой id_file_open, установлен обработчик.

Теперь в дереве проекта выполним щелчок на модуле TaskWindow.pro, чтобы выделить его и, используя пункт меню Build, просто скомпилируем его. Если модуль TaskWindow.pro не выделен, тогда пункт Compile в меню Build будет недоступен.

При компиляции модуля дерево проекта реорганизуется таким образом, что все описания предикатов и типов(domains) становятся доступными для пользователя. После окончания компиляции, все предикаты модуля TaskWindow видны в дереве проекта (в правой части окна, см. рис. 11.28).

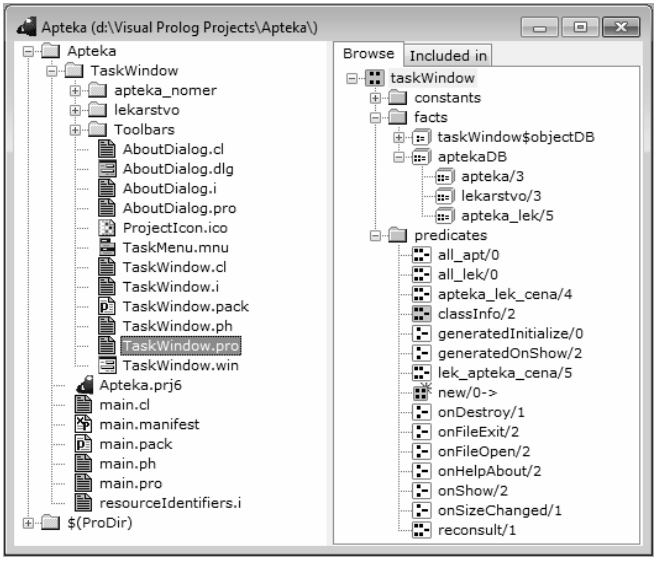


Рис. 11.28. Список предикатов в модуле Taskwindow.pro

Можно заметить, что в списке предикатов модуля Taskwindow.pro появился предикат onFileOpen, который ранее отсутствовал.

Если дважды щелкнуть на этом предикате, то откроется редактор прямо на месте объявления предиката onFileOpen. В случае если их несколько, курсор установится на первом.

Предикат onFileOpen называется обработчиком. Программисту не нужно самому вызывать его. Windows будет автоматически обращаться к обработчику, когда активизируется соответствующий GUI компонент (в случае щелчка на пункте меню).

По умолчанию следующий код будет добавлен для обработчика этого события:

```
onFileOpen(_Source, _MenuTag).
```

Заменим этот код следующим

```
predicates
        onFileOpen : window::menuItemListener.
clauses
        onFileOpen(_Source, _MenuTag):-
                currentDir:= directory::getCurrentDirectory(),
                directory::setCurrentDirectory("..\\"),
                Filename = vpiCommonDialogs::getFileName("*.txt",
                        ["БД Аптек (*.txt)","*.txt","Все файлы","*.*"],
                        "\nЗагрузка...\n", [], ".", _),
                directory::setCurrentDirectory(currentDir),
                !, reconsult(Filename),
                stdIO::writef("БД % загружена\n", Filename).
onFileOpen(_Source, _MenuTag):-
                directory::setCurrentDirectory(currentDir).
```

Из приведенного кода следует, что мы просто добавили тело предиката после заголовка, созданного программной средой. Первое утверждение предназначено для открытия стандартного диалога Windows для загрузки базы данных. Второе утверждение создает успешное завершение действия и в том случае, если пользователь отменит этот диалог.

Если скомпилировать и запустить программу на данной стадии, то можно будет загрузить базу данных аптеки, используя пункт меню *Загрузка базы данных*|*Открыть базу данных*. Для проверки этого следует использовать базу db.txt. Сообщение, появляющееся в окне Messages, информирует нас об успешной загрузке базы данных.

Заметим, что хотя диалог и будет спрашивать "**.txt*" файлы, их не надо путать с обычными текстовыми файлами. Файлы базы данных должны быть форматированы согласно требованиям программы. Любой другой файл приведет к ошибке.

Если файл загружен корректно, то вызывается предикат *stdIO::writef*(…), сообщающий об успешной загрузке базы данных. Программа с графическим интерфейсом не имеет постоянной консоли, однако, Visual Prolog автоматически предоставляет программе окно Messages (Сообщения), которое действует как консоль вывода. Таким образом, результат выполнения предиката *stdIO::writef*(…) появляется в окне Messages. (Если окно закрыто, то нельзя увидеть результат).

Теперь необходимо вернуться к организации диалога с помощью диалоговых окон и эксперта окон (Dialog and Window Expert), и убедиться, что обработчики для пунктов меню *Запросы|Аптеки города*, *Запросы|Список лекарств*, *Запросы|Найти лекарство…*, *Запросы|Лекарства в аптеке…* установлены так, как показано на рис. 11.29.

Рис. 11.29. Окно диалога "Эксперт окон и диалогов"

Для этого в контекстном меню при выделении каждого пункта надо выбрать команду Set Event (Установить событие).

Сейчас для пунктов меню *Запросы|Список лекарств*, *Запросы|Лекарства в аптеке…* и *Запросы|Найти лекарство... Запросы|Аптеки города* добавим следующие тексты процедур после заголовка, сгенерированного Visual Prolog по умолчанию:

```
predicates
        onQueryСписокАптек : window::menuItemListener.
clauses
        onQueryСписокАптек(_Source, _MenuTag):-
                        all_apt().
predicates
        onQueryСписокЛекарств:window::menuItemListener.
clauses
        onQueryСписокЛекарств(_Source, _MenuTag):-
                        all_lek().
predicates
```

```
        onQueryНайтиЛекарство:window::menuItemListener.
    clauses
        onQueryНайтиЛекарство(_Source, _MenuTag):-
                X = lekarstvo::tryGetName(This),
                stdIO::writef("Данные по поиску лекарства \n", X),
                stdIO::write("Номер аптеки Адрес Телефон Цена\n"),
                lek_apteka_cena(X, Napt, Adress, Tel, Cena),
                stdIO::writef("% % % %\n",Napt, Adress, Tel, Cena ),
                fail.
        onQueryНайтиЛекарство(_Source, _MenuTag):-
                stdIO::write("\nСписок закончен\n").
    predicates
        onQueryЛекарстваВАптеке : window::menuItemListener.
    clauses
        onQueryЛекарстваВАптеке(_Source, _MenuTag):-
                X = apteka_nomer::tryGetName(This),
                stdIO::writef("Данные об аптеке № % \n",X),
                stdIO::write("Номер аптеки Лекарство Количество
                        Цена\n"),
                Napt = toTerm(X),
                apteka_lek_cena(Napt, Lek, Kol, Cena),
                stdIO::writef("% % % %\n",Napt, Lek, Kol, Cena ),
                fail.
    onQueryЛекарстваВАптеке(_Source, _MenuTag):-
                stdIO::write("\nСписок закончен\n").
```

Обсудим то, что мы до сих пор делали. Итак, мы разбили код на две части: не интерактивное логическое ядро и те фрагменты, которые требуют ввода от пользователя. Они были размещены в разных местах в различных обработчиках событий. Рассмотрим фрагмент интерактивного кода для пункта меню *Запросы|Найти лекарство...*:

```
    predicates
        onQueryНайтиЛекарство: window::menuItemListener.
    clauses
        onQueryНайтиЛекарство(_Source, _MenuTag):-
                X = lekarstvo::tryGetName(This),
                stdIO::writef("Данные по поиску лекарства \n",X),
                stdIO::write("Номер аптеки Адрес Телефон  Цена\n"),
                lek_apteka_cena(X, Napt, Adress, Tel, Cena ),
                stdIO::writef("% % % %\n",Napt, Adress, Tel, Cena ),
                fail.
        onQueryНайтиЛекарство(_Source, _MenuTag):-
                stdIO::write("\nСписок закончен\n").
```

Процедура получает строку из модального диалогового окна *X = lekarstvo::tryGetName(This)*, представленного созданным ранее окном lekarstvo. Предикат *onQueryНайтиЛекарство(_Source, _MenuTag)* предполагает, что существует предикат *tryGetName*, доступный в модуле *lekarstvo*. Он возвращает наименование лекарства, которое нужно найти в аптеках города.

Как было видно в обработчике *onFileOpen*, мы искали строку (имя файла), возвращаемую из модального диалога. Сам модальный диалог вызывался предикатом *vpiCommonDialogs::getFileName(...)*. Ту же самую стратегию мы применяем для получения строки из окна диалога *lekarstvo*. Единственное различие состоит в том, что *vpiCommonDialogs::getFileName(...)* предоставляет встроенный стандартный диалог Windows. Но для нашего диалога *lekarstvo* нужно написать свой код.

При компиляции программы возникнет одна ошибка:

```
error c229: Undeclared identifier ' lekarstvo::tryGetName/1->'
```

Причина этой ошибки в том, что предикат *onQueryНайтиЛекарство* требует предиката *tryGetName* из модуля *lekarstvo.pro*. Но мы его еще не написали! Давайте уделим внимание исправлению этой ошибки.

Предикат *tryGetName* определен внутри одного модуля, но вызывается из другого. Таким образом, нам надо убедиться в том, что объявление находится не в *.pro частях программы. Файл декларации классов (расширение *.cl) как раз одно из наиболее подходящих для этого мест. Поэтому давайте откроем lekarstvo.cl и добавим туда следующий кусок кода.

```
predicates
        trygetName : (window Parent) ->string Name determ.
```

В *lekarstvo.pro* давайте добавим логическое ядро. Так же как мы сделали для *Taskwindow.pro* напишем его сразу после строки:

```
classInfo(className, classVersion).
```

Вот код, который необходимо писать:

```
domains
        optionalString=none(); one(string Value).

class facts
        name: optionalString:=none().
clauses
```

```
tryGetName(Parent)=Name:- name:=none(),
        _=lekarstvo::display(Parent),
        one(Name)=name.
```

Этот пример иллюстрирует тот случай, когда и факт *name*, и предикат *tryGetName(Parent)* являются фактом и предикатом класса, а не объекта (см. пункт "Объекты и классы" в параграфе 11.1).

Теперь изменим обработчик событий для кнопки OK. Это необходимо для того, чтобы диалог передавал вводимые пользователем данные в поле *name* класса. Если этого не сделать, то предикат *tryGetName* будет возвращать пустую строку. Для этого щелкнем мышкой на форме *lekarstvo* и вызовем контекстное меню, в нем выберем пункт Code Expert …, затем раскроем пункт + Controls, добавим элементу ok_ctl событие onOk и скомпилируем проект.

Код, генерируемый Visual Prolog, представлен ниже:

```
predicates
    onOk : button::clickResponder.
clauses
    onOk(_Source) = button::defaultAction().
```

Этот код необходимо заменить на следующий:

```
predicates
    onOk : button::clickResponder.
clauses
    onOk(_Source) = button::defaultAction():-
                    Name = edit_ctl:getText(),
                    name :=one(Name).
```

Теперь мы, наконец, закончили нашу программу. Если скомпилировать и запустить ее, то не возникнет никаких ошибок.

Для номера аптеки необходимо выполнить аналогичные действия.

Выполнение программы

При запуске программы мы заметим, что желаемые действия не выполняются незамедлительно. Так например можно запускать пункты меню *Запросы|Список лекарств*, *Запросы|Лекарства в аптеке…* и *Запросы|Найти лекарство... Запросы|Аптеки города* без загрузки каких-либо данных, и это не приведет к выдаче какого-либо результата или ошибки, т.к. логика нашей программы заботится о ситуациях, когда отсутствуют данные. Чтобы получать какие-либо результаты, нам нужно выполнить пункт меню Загрузка базы данных | Открыть базу данных... и загрузить базу данных аптеки (db.txt). Содержимое файла db.txt описано в начале п. 11.2.

После этого, мы можем протестировать сначала пункты меню *Запросы|Аптеки города*, *Запросы|Список лекарств*, и *Запросы|Найти лекарство..., Запросы|Лекарства в аптеке…*, затем те же результаты получить, используя кнопки панели инструментов. Ответы на наши запросы будут показаны в окне Messages.

Преимущество GUI программы станет очевидно *при понимании того*, что можно выполнять запросы любое число раз и загружать различные данные в любое время.

11.3. Создание приложений с использованием графического интерфейса. Управление с помощью элементов формы. Создание нового класса.

Постановка задачи

Разработать приложение *VStud* для работы с базой, содержащей информацию о студентах, их характеристиках, результатах последней экзаменационной сессии и т.п. Взаимодействие пользователя с приложением должно осуществляться через главное меню окна задач *(TaskWindow*) и форму *(FStud)* с различными управляющими элементами. Необходимые данные будут вводиться из полей редактирования *(edit_ctl, listEdit_ctl),* а результаты выводиться в окно *сообщений (Messages)* или в поле для вывода списка строк *(listbox_ctl).* Различные вопросы к базе знаний можно будет задавать нажатием одной из соответствующих кнопок *(pushButton_ctl*).

План решения задачи

1. Создать новый проект с именем *VStud,* указав стратегию использования графического интерфейса.

2. Скомпоновать проект и выполнить его. Познакомиться с главным окном задач и окном сообщений. Исследовать дерево проекта.

3. Создать новый пакет с именем *PStud*.

4. В каталоге пакета создать форму *FStud* и новый класс *CStud.*

5. Вновь скомпоновать проект.

6. Заполнить файл реализации (*CStud.pro*) утверждениями (*clauses*) программы. Затем описать факты и предикаты в зависимости от их назначения либо в разделах *class facts, class predicates* файла *CStud.pro,* либо в заголовочном файле (*CStud.cl)* или в файле интерфейса (*CStud.i),* чтобы указать, что эти предикаты могут быть доступны другим элементам проекта.

7. Наполнить форму управляющими элементами: поле ввода, поле со списком, текстовое поле и кнопки.

8. Подключить команду создать и открыть форму к пункту главного меню *File/Открыть форму*.

9. Для каждой кнопки определить предикат-обработчик, вызывающий на выполнение один из запросов к базе знаний.

10. Скомпоновать и выполнить проект.

Факты базы знаний

Предметная область "Студенты" может быть описана следующими фактами:

```
/* Факты базы знаний */
clauses
        группа(7412).
        группа(743).

        студент("Иванов", 7412).
        студент("Задоя", 743).
        студент("Илларионов", 7412).
        студент("Петрова", 743).

        возраст_студента("Иванов", 19).
        возраст_студента("Задоя", 18).
        возраст_студента("Илларионов", 20).
        возраст_студента("Петрова", 19).

        оценки_за_сессию("Иванов",5,5,3).
        оценки_за_сессию("Задоя",3,3,3).
        оценки_за_сессию("Илларионов",5,5,5).
        оценки_за_сессию("Петрова",4,5,3).
```

С помощью следующих предикатов из базы фактов можно получать различную информацию.

```
/* Правила */
/*  предикат - номер студенческой группы */
группа_п(Группа):- группа(Группа).

/* предикат связывающий студента и группу */
студент_п(Фио, Группа):- студент(Фио, Группа).

/* предикат, определяющий возраст студента */
возраст_студента_п(Фио, Возраст):-
                        возраст_студента(Фио, Возраст).

/* предикат получения результатов сессии студента с фамилией
Фио */
оценки_средний_балл(Фио,Оценка1,Оценка2, Оценка3, Ср_балл):-
        оценки_за_сессию(Фио, Оценка1, Оценка2, Оценка3),
```

Ср_балл = (Оценка1+Оценка2+ Оценка3)/3.0.

Приступим к разработке проекта

1. Для создания нового проекта в интегрированной среде разработки (*Visual Prolog IDE)* выберем пункт меню *Project/New* и в диалоговом окне создания проекта заполним только форму General (рис. 11.30). В этой форме обязательно нужно указать имя проекта, стратегию графического интерфейса *(GUI application)* и корневой каталог *(Base Directory)*, в котором будет находиться проект.

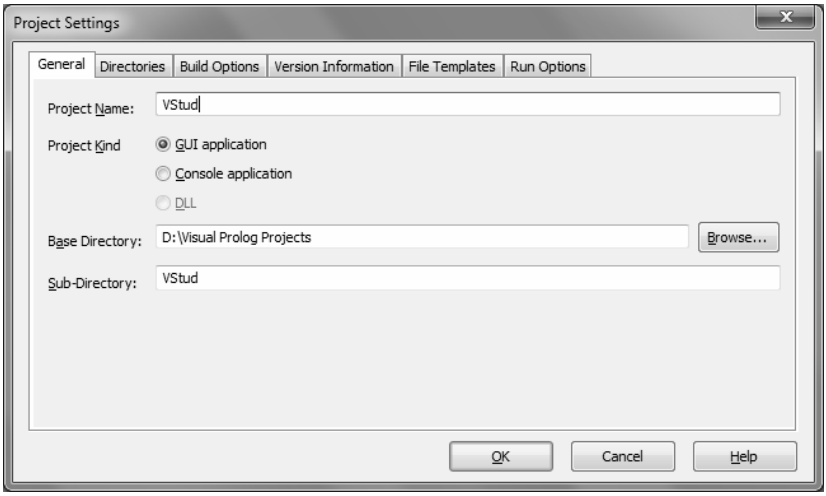


Рис. 11.30. Создание проекта с графическим интерфейсом

2. Скомпонуем проект *(Build/Build)* и выполним его (*Execute*). Появится главное окно приложения (рис. 11.31).

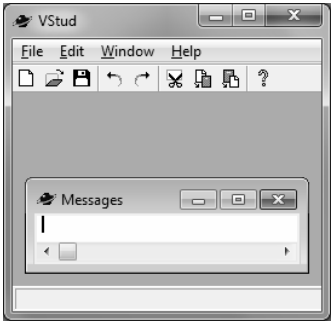


Рис. 11.31. Окно "пустого приложения"

В нем есть меню с основными пунктами (*File, Edit, Window, Help*), панель инструментов и окно сообщений (*Messages*). В это окно могут выводиться результаты выполнения предикатов. С помощью команд меню можно будет управлять выводом нужной информации для пользователя.

Если закрыть окно приложения, становится видным окно с **деревом проекта** (рис. 11.32).

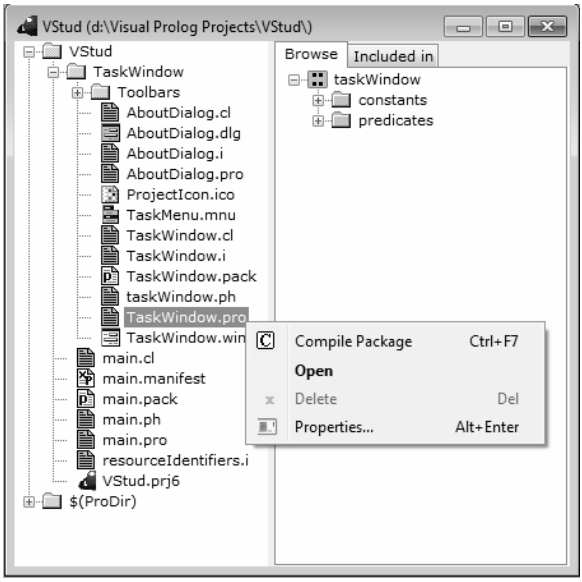


Рис. 11.32. Дерево проекта

Самый лёгкий способ ориентироваться в файлах и ресурсах – это щелкать мышкой по соответствующим элементам дерева проекта: если дважды щёлкнуть по папке, она откроется и отобразит свое содержимое. Если щёлкнуть по элементу правой кнопкой мыши, откроется контекстное меню, как на рис. 11.32.

Рассмотрим назначение различных видов файлов, открывающихся в дереве проекта. *Visual Prolog* – это объектно-ориентированный язык, поэтому все сущности проекта описываются в классах или реализуются в объектах некоторого класса. Для объявления и реализации каждого класса создаются новые файлы, классы могут по теме объединяться в пакеты. Пакет выделяется в отдельный каталог (например, *TaskWindow*). Опишем назначение каждого вида файла:

- **.ph* – заголовочный файл пакета. Пакет – это набор классов и интерфейсов, предназначенных для решения определенного класса задач;
- **.pack* – файл–пакет. Он содержит реализацию пакета и перечисляет все подключаемые к пакету файлы;
- **.i* - файл содержит интерфейс объектов определенного класса, т.е. описывает домены и предикаты объекта, доступные другим сущностям проекта. Интерфейс может рассматриваться как объявление типа объектов;
- **.cl* – файл содержит объявление класса, в нем описываются предикаты и константы, доступные всем объектам класса и другим сущностям проекта. Так как класс может создавать объекты, то в файле **.cl* обычно находятся объявления *конструкторов* класса;
- **.pro* – файл содержит реализацию класса, т.е. в нем располагаются утверждения программы, осуществляющие действия, заявленные в предикатах. Можно сказать, что описание логики приложения, располагается в **.pro* – файлах;
- **.dlg* – файл описывает диалог;
- **.frm* – файл содержит информацию о форме;
- **.win* – файл описывает окно задач;
- **.mnu* – файл содержит информацию о меню

и так далее.

В папке *$ProDir* находятся библиотека (*lib)* и описание основных классов *(pfc)* языка *Visual Prolog*, доступных приложению. Папка *TaskWindow* содержит пакеты и описания классов, соответствующих основным элементам главного окна задач, таким, например, как *Меню, Панель инструментов, Диалог о приложении, Строка статуса* и т.п.

Пакеты удобны тем, что в них мы можем выделить часть кода программы, связанную с определенной темой. Для решения задач нашего проекта создадим новый пакет *PStud* , в нем форму *FStud* и новый класс *CStud*, предназначенный для реализации всех операций с базой знаний "Студенты". Для этого выполним следующие шаги.

3. Создание нового пакета. В дереве проекта выделим корневую вершину *VStud* и по правой кнопке мыши в контекстном меню выберем пункт *New in New Package* (Создание новой сущности проекта в новом пакете). В открывшемся окне (рис. 11.33) укажем тип сущности *Package* (пакет), дадим пакету имя *PStud* и нажмем кнопку *Create.* После создания пакета можно выполнить сборку проекта *Build/Build*.

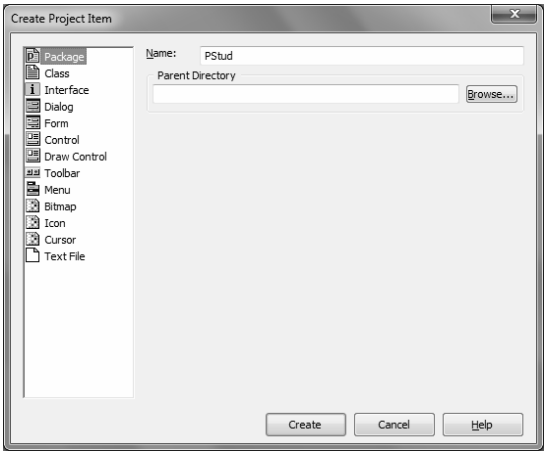


Рис. 11.33. Создание нового пакета

4. Создание формы. В дереве проекта выделим вершину PStud и по правой кнопке мыши в контекстном меню выберем пункт *New in Existing Package* (Создание новой сущности проекта в существующем пакете). В открывшемся окне (рис. 11.34) укажем тип сущности Form (форма), дадим форме имя FStud, увидим имя пакета Existing Package и нажмем кнопку Create. Появится окно свойств формы, принятых по умолчанию. Нажав кнопку <OK>, мы подтвердим указанные значения.

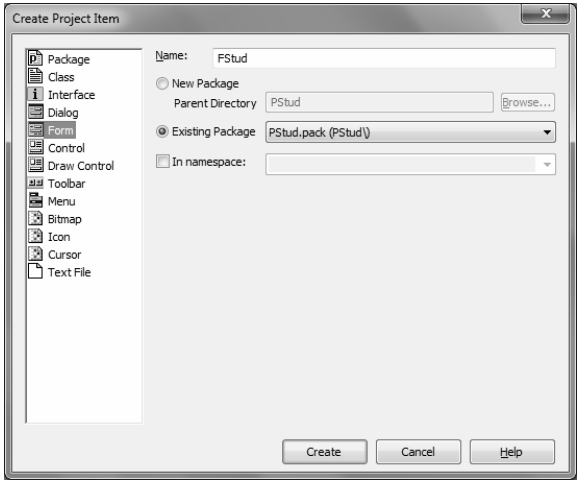


Рис.11.34. Создание формы

В результате получим форму (рис. 11.35) с тремя стандартными кнопками. На рис. 11.35 видны форма и слева от нее панель с набором управляющих элементов, которые можно добавить к форме.

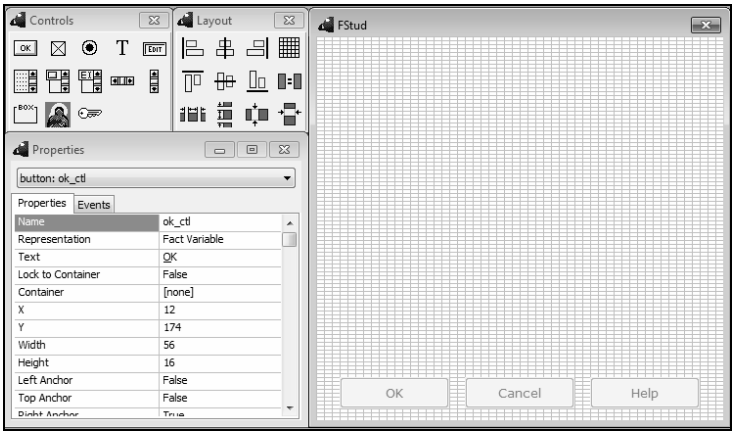


Рис.11.35. Форма приложения и панель с управляющими элементами

5. Создание нового класса. В дереве проекта выделим вершину *PStud* и по правой кнопке мыши в контекстном меню выберем пункт *New in Existing Package*. В открывшемся окне (рис. 11.36) укажем тип сущности *Class* (класс), дадим классу имя *CStud*, выберем существующий пакет *Existing Package*, снимем галочку *Create Interface* и нажмем кнопку *Create*.

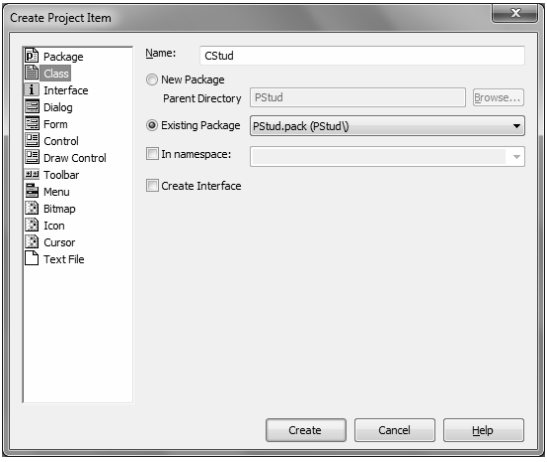


Рис. 11.36. Создание нового класса

При указанных опциях класс будет создан внутри пакета, но без возможности генерации объектов класса. В этом случае файл интерфейса не нужен. В дереве проекта появляются только два файла: заголовочный файл *cStud.cl* – в нем объявляются сам класс, а также все доступные другим сущностям предикаты, типы и константы, и файл реализации *cStud.pro,* который содержит факты базы знаний и утверждения, реализующие выполнение объявленных и описанных предикатов.

6. Выполним сборку проекта *(Build/Build).*

7. Отредактируем файл реализации *cStud.pro,* дополнив его утверждениями нашей программы. Содержимое файла примет следующий вид:

```
implement cStud
        open core
constants
        className = "PStud/cStud".
        classVersion = "".
clauses
        classInfo(className, classVersion).
class facts
        группа:(группа) nondeterm.
        студент:(фио, группа) nondeterm.
        возраст_студента:(фио, возраст) nondeterm.
        оценки_за_сессию:(фио,оценка,оценка,оценка) nondeterm.
clauses
        группа(7412).
        группа(743).

        студент("Иванов", 7412).
        студент("Задоя", 743).
        студент("Илларионов", 7412).
        студент("Петрова", 743).

        возраст_студента("Иванов", 19).
        возраст_студента("Задоя", 18).
        возраст_студента("Илларионов", 20).
        возраст_студента("Петрова", 19).

        оценки_за_сессию("Иванов",5,5,3).
        оценки_за_сессию("Задоя",3,3,3).
        оценки_за_сессию("Илларионов",5,5,5).
        оценки_за_сессию("Петрова",4,5,3).
/*Правила*/
/*предикат для получения номеров студенческих групп*/
        группа_п(Группа):- группа(Группа).
```

```
/*предикат, связывающий студента и группу*/
        студент_п(Фио, Группа):- студент(Фио, Группа).
/*предикат, определяющий возраст студента*/
        возраст_студента_п(Фио, Возраст):-
                возраст_студента(Фио, Возраст).
/*предикат получения результатов сессии студента
с фамилией Фио*/
        оценки_средний_балл(Фио, Оценка1, Оценка2,
        Оценка3, Ср_балл):-
                оценки_за_сессию(Фио, Оценка1, Оценка2, Оценка3),
                Ср_балл = (Оценка1+Оценка2+ Оценка3)/3.0.
end implement cStud
```

Рассмотрим текст программы. Ключевые слова *facts, clauses, constants* указывают на различные разделы программы. После ключевого слова *open* указываются имена классов, подключаемых к модулю реализации. В этом случае к предикатам соответствующего класса можно обращаться по имени <*имя предиката(…)*> без указания имени класса. Если не подключать класс, то обращение к предикату класса будет иметь вид: <*имя класса*>**::**<*имя предиката(…)*>.

Все предикаты должны быть описаны. Если мы хотим, чтобы соответствующий предикат был доступен другим элементам проекта, то он должен быть описан в разделе *predicates* либо файла интерфейса *(*.i)* – это будет предикат объекта, либо в заголовочном файле *CStud.cl* – получим предикат класса. Если предикат или факт используется только внутри файла реализации, то он описывается в разделе *class predicates* или *class facts* самого файла реализации.

В нашем случае факты базы знаний описываются в разделе *class facts*, это означает, что они доступны только в файле *CStud.pro* и образуют динамическую базу данных, которую можно изменять в процессе выполнения программы. Типы данных (*domains*) и предикаты, определяемые с помощью правил, такие как *группа_п( ), студент_п( ), оценки_средний_балл( ) и т.д.*, описываются в заголовочном файле *CStud.cl*, а потому к ним можно обращаться из окна задач, из формы или других элементов приложения. Ниже приведен текст заголовочного файла *CStud.cl*.

```
class cStud
        open core
domains
        группа = integer. фио = string.
        возраст = integer. оценка = integer.
predicates
        classInfo : core::classInfo.
```

```
        группа_п:(группа) nondeterm (o) (i) .
        студент_п:(фио, группа) nondeterm (o,o) (i,o) (o,i) (i,i).
        возраст_студента_п:(фио, возраст)
                nondeterm (o,o) (i,i) (i,o) (o,i).
        оценки_средний_балл:(фио, оценка, оценка, оценка, real)
                nondeterm (i,o,o,o,o) .
    end class cStud
```

При описании предикатов следует указать типы аргументов, режимы детерминизма и шаблоны потока. Все эти понятия и правила описания рассмотрены в пункте "Создание консольных приложений" в подпунктах "Объявления", "Объекты и классы".

На этом создание класса *CStud* завершено. Теперь наша задача научиться с помощью управляющих элементов окна задач и формы извлекать информацию из фактов базы знаний, принадлежащих классу *CStud*.

8. Чтобы форма стала доступной пользователю, ее нужно создать и открыть. В нашем примере эти действия будут выполняться при выборе пользователем пункта меню *File/Открыть форму* главного окна задач. Для этого в дереве проекта откроем папку *TaskWindow*, в ней выделим файл *TaskMenu.mnu* и в контекстном меню выберем пункт *Edit*. В открывшемся окне *TaskMenu (рис. 11.37)* переименуем пункт *New* в *Открыть форму* и, чтобы сделать пункт меню *File/Открыть форму* активным, уберем галочку из переключателя *Disabled*. Аналогично нужно переименовать пункт Edit в пункт Запросы.

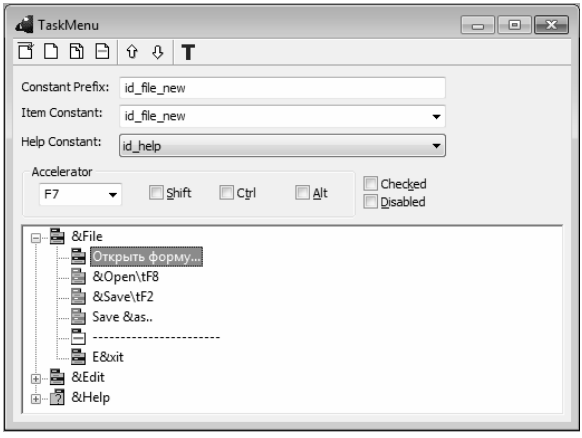


Рис. 11.37. Активизация пунктов меню окна задач

9. На следующем шаге нужно создать обработчик события "Выбор пункта меню *File/ Открыть форму*". Обработчик события в языке *Visual Prolog* – это предикат-правило, в теле которого описываются действия, выполняемые при возникновении события. Код обработчика события создается с помощью программы-мастера, называемой "Экспертом кода". Выполняется это следующим образом. Выделим в дереве проекта файл *TaskWindow.win* (рис. 11.38) и по правой кнопке мыши в контекстном меню выберем пункт *Code Expert* ….

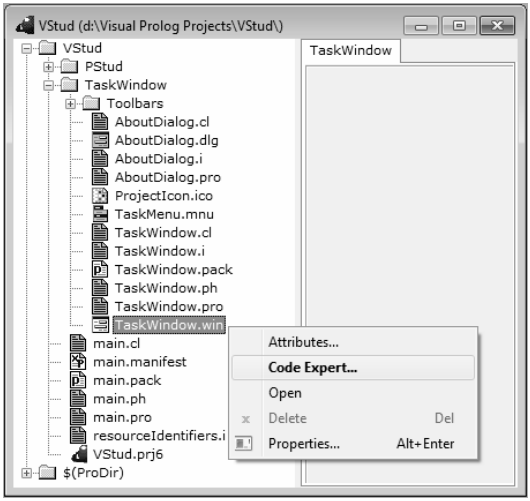


Рис.11.38

Появляется окно "Dialog and Window Expert" (рис. 11.39). В нем последовательно раскроем пункты *Menu, TaskMenu, id_file,* выделим *id_file_new* и нажмем кнопку *Add*, получим результат, представленный на рис. 11.40. Двойным нажатием клавиши мыши на пункт *onFileNew* перейдем в соответствующее место файла *TaskWindow.pro* и переопределим предикат *onFileNew* (обработчик события) следующим образом.

```
predicates
        onFileNew : window::menuItemListener.
clauses
        onFileNew(W, _MenuTag):- S = fStud::new(W), S:show( ).
```

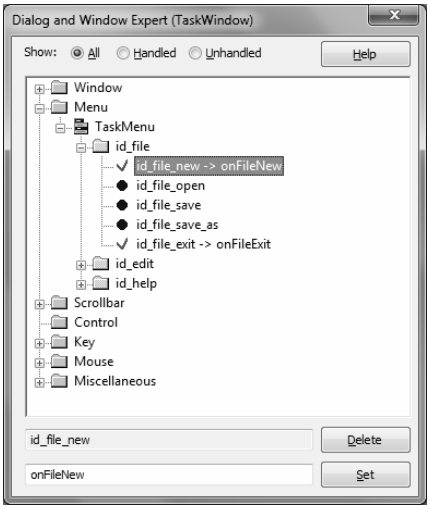


Рис.11.39

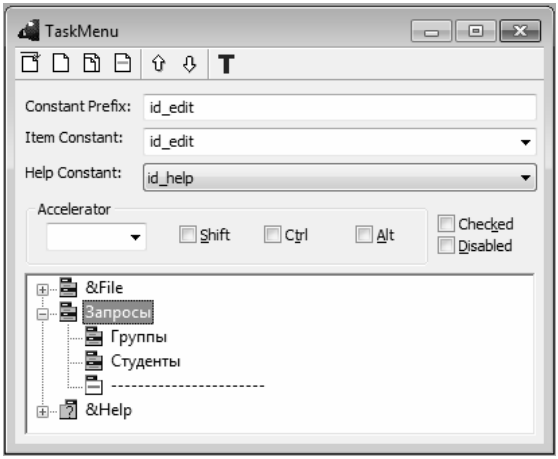


Рис. 11.40

Поясним тело правила. Сначала с помощью конструктора *fStud::new(W)* создается форма *S* – объект класса *fStud.* Затем форма-объект с помощью метода *S:show( )* становится видимой. В этом примере проявляется различие обращения к методам класса и к методам объекта. В первом случае имя класса и имя метода разделяются двойным двоеточием, во втором – одним. Выполним проект и, выбрав в

окне приложения пункт *File/Открыть форму*, убедимся, что открывается созданная форма, пока еще пустая.

Повторим шаги процесса создания обработчиков пунктов меню и применим их к определению новых пунктов *Запросы/Группы* и *Запросы/Студенты* нашего приложения. Первый шаг – это переименовать пункты меню и сделать их активными, удалив галочку *Disabled* в окне *TaskMenu* (рис. 11.40). Попасть в него можно из дерева проекта с помощью команд *TaskWindow/TaskMenu.mnu/Edit.*

Следующая задача: к каждому пункту меню присоединить обработчик события выбора этого пункта. С помощью команд *TaskWindow/TaskWindow.win/Code Expert…*открывается окно *Dialog and Window Expert…* (рис. 11.41), в нем последовательно открываются *Menu/TaskMenu/id_edit*, соответствующий пункт меню выделяется (например, *id_edit_undo*) и выполняется команда *Add*. Двойным щелчком мыши по имени обработчика *onEditUndo* нужно перейти к тексту программы файла *TaskWindow.pro* и написать тело правила обработчика.

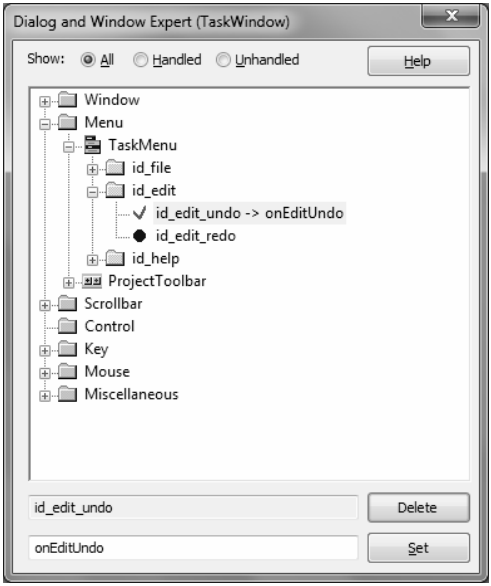


Рис. 11.41

```
predicates
        onEditUndo : window::menuItemListener.
clauses
        onEditUndo(_Source, _MenuTag):-
```

```
            cStud::группа_п(Группа),
            stdIO::write(Группа,"\n"), fail.
    onEditUndo(_Source, _MenuTag).
```

Из текста следует, что при выборе пункта *Запросы/Группы* в окно сообщений будут выведены все группы. Здесь *write( )* – это метод класса *stdIO*, позволяющий выводить информацию в окно сообщений, *группа_п( )* – предикат-метод класса *cStud*.

Аналогичным образом можно определить обработчик и для пункта *Студенты*.

```
predicates
    onEditRedo : window::menuItemListener.
clauses
    onEditRedo(_Source, _MenuTag):-
            cStud::студент_п(ФИО, Группа),
            stdIO::write(ФИО,"\n"), fail.
    onEditRedo(_Source, _MenuTag).
```

10. Теперь займемся разработкой формы. Наполним форму управляющими элементами в соответствии с рис. 11.42.

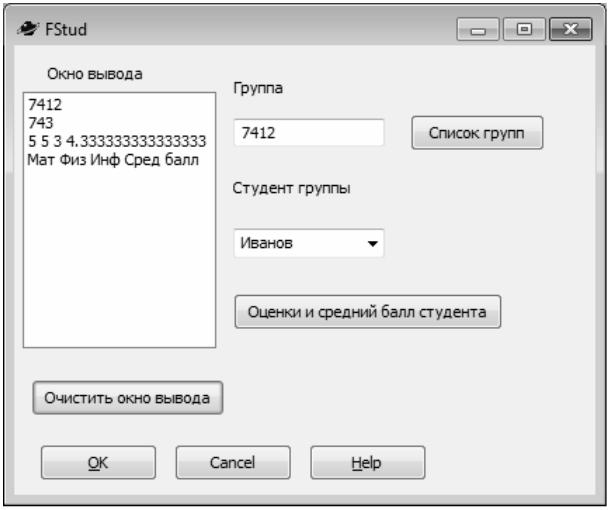


Рис. 11.42. Форма взаимодействия с пользователем.

В форме использованы следующие управляющие элементы: поле ввода *(edit_ctl)* с заголовком "Группа", поле со списком *(listEdit_ctl)* с заголовком "Студент группы", "Окно вывода" – поле для вывода строк *(listbox_ctl)*. Различные заголовки *(staticText*) типа "Группа", "Студент

группы" поясняют назначение элементов. Кнопки *(pushButton)*: "Список групп", "Оценки и средний балл студента" предназначены для выполнения запросов к базе знаний.

С каждой кнопкой свяжем обработчик события нажатия кнопки. Для этого выполним на соответствующей кнопке на форме двойной клик мыши, после выполнения этого действия автоматически будет сгенерирован код в файле *fStud.pro* и произведен переход к предикату-обработчику *onPushButtonClick*. Переопределим его следующим образом.

```
clauses
    onPushButtonClick(_Source) = button::defaultAction():-
            группа_п(Группа),
            listbox_ctl:add(tostring(Группа)),fail.
    onPushButtonClick (_Source) = button::defaultAction().
```

При нажатии этой кнопки в окне вывода появится список групп.

Для того, чтобы стал доступен предикат *групп*а_п и остальные предикаты из класса *CStud*, после ключевого слова *open* надо дописать имя этого класса *cstud*.

```
implement fStud
    inherits formWindow
    open core, vpiDomains, cstud
```

Ниже приводятся обработчики для каждого управляющего элемента. Для создания соответствующих обработчиков нужно выделить управляющий элемент, в окне свойств *Properties* перейти на вкладку *Events* (События), на нужном событии выполнить двойной клик мыши. Будет автоматически сгененирован код для этого обработчика.

Поле ввода "Группа" *(edit_ctl)*, событие "Потерять фокус" *(LoseFocus)*, обработчик *onEditLoseFocus*.

```
clauses
    onEditLoseFocus(_Source):-
            listedit_ctl:clearAll(),
            Группа = toterm(edit_ctl:getText()),
            студент_п(Фио, Группа),
            listedit_ctl:add(Фио),fail.
    onEditLoseFocus(_Source).
```

Сначала поле со списком *(listedit_ctl:clearAll())* очищается, затем из поля ввода берется строка и преобразуется к типу группа *(toterm(edit_ctl:getText()))*, в базе фактов находится студент соответствующей группы, и его фамилия добавляется в поле со списком

(listedit_ctl:add(Фио)). Процесс повторяется, пока все студенты соответствующей группы не попадут в список для выбора.

Кнопка "Оценки и средний балл студента" *(pushButton3_ctl)*, обработчик *onPushButton3Click:*

```
clauses
        onPushButton3Click(_Source) = button::defaultAction():-
                оценки_средний_балл(listedit_ctl:getText(), Q1, Q2, Q3,
                        Sr),
                Str1 =string::concat( tostring(Q1)," ",tostring(Q2), " ",
                        tostring(Q3)),
                Str =string::concat(Str1," ",tostring(Sr)),
                listbox_ctl:add(Str),fail .
        onPushButton3Click(_Source) = button::defaultAction():-
                listbox_ctl:add("Мат Физ Инф Сред балл").
```

Из поля со списком *(listedit_ctl:getText())* берется фамилия студента, в базе фактов находятся его оценки, подсчитывается средний балл, формируется строка с оценками и средним баллом *Str1 =string::concat(tostring(Q1)," ",tostring(Q2)," ",tostring(Q3)), Str =string::concat(Str1," ",tostring(Sr)),*. Эта строка добавляется в окно вывода *(listbox_ctl:add(Str))*.

Кнопка "Очистить окно вывода" (*pushButton1_ctl*), обработчик *onPushButton1Click:*

```
clauses
        onPushButton1Click(_Source) = button::defaultAction():-
                listbox_ctl:clearAll().
```

При нажатии этой кнопки окно вывода очищается.

11. Теперь можно запустить на выполнение разработанное приложение (см. рис. 11.43). Выбрав пункт меню *File/Открыть форму*, откроем форму, введем номер группы в поле "Группа" и выберем фамилию студента в поле со списком, нажмем на кнопку "*Оценки и средний балл студента*". Запрошенная информация появится в окне вывода. Далее можно проверить работу других управляющих элементов формы и пунктов меню *Запросы/Группы*, *Запросы/Студенты* окна задач. Добавляя на форму кнопки или новые пункты меню "Окна задач" и описывая в виде предикатов обработчики событий, можно реализовать новые запросы к базе знаний (рис. 11.43).

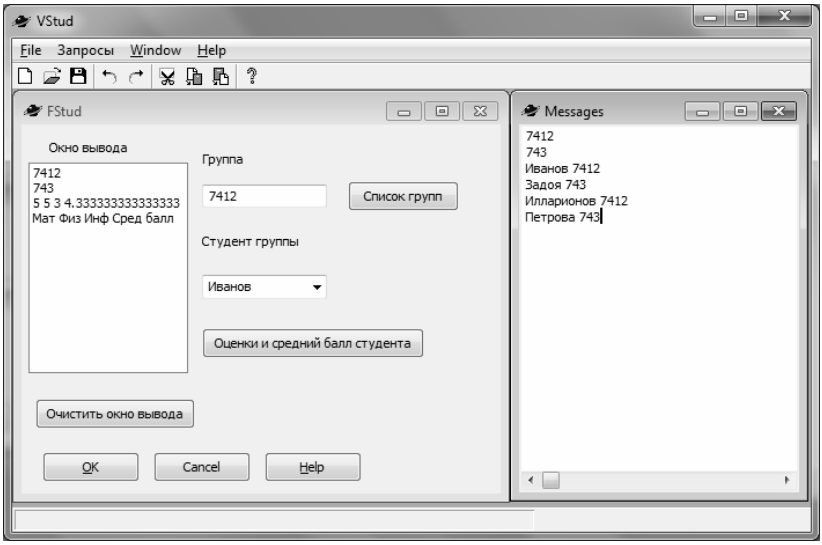


Рис. 11.43. Результаты работы приложения "Студенты"

11.4. Технология создания программы в режиме работы с графическим интерфейсом

1. При создании нового проекта кроме его имени указать режим графического интерфейса.

2. Создать свой пакет *P<имя>* в корневом каталоге.

3. Внутри пакета создать свой класс *C<имя>* (можно без генерации объектов или с генерацией объектов). Без генерации объектов все доступные другим классам сущности должны быть описаны в заголовочном файле *C<имя>.cl*, при генерации объектов все доступные другим классам сущности должны быть описаны в файле интерфейса *C<имя>.i*. Факты, доступные текущему объекту класса, описываются в разделе facts, факты, доступные всем объектам класса, и в режиме без генерации объектов, описываются в разделе class facts.

4. Внутри пакета создать свою форму *F<имя>*, наполнить ее управляющими элементами. К ним относятся: текстовое поле – *StaticText*(может использоваться для ввода и вывода данных), поле редактирования для ввода и вывода данных – *edit_ctl*, окно вывода – *listbox* для вывода списка строк, кнопка – *puchButton_ctl* для инициализации обработчика, вызывающего определенные действия, поле со списком – *listEdit*, *listButton* для выбора из меню элемента ввода.

5. Для каждого управляющего элемента в файле реализации формы *F<имя>.pro* написать предикаты-обработчики определенных событий. В теле этих обработчиков можно обращаться к тем предикатам разработанного класса, которые описаны в заголовочном файле "*C<имя>.cl*" или в файле интерфейса "*C<имя>.i*". При написании обработчиков можно следовать одному из следующих правил.

- В теле обработчика обратиться к предикату разработанного класса, передав ему данные из управляющих элементов. Предикат должен возвращать все необходимые для визуализации результаты решения задачи. Эти результаты передаются управляющим элементам формы для вывода.
- При втором подходе программа обработки данных, представленных в фактах класса, пишется в теле обработчика. При этом для каждого востребованного факта нужно определить предикат и сделать его доступным другим модулям, описав в заголовочном файле "*C<имя>.cl*" или в файле интерфейса "*C<имя>.i*".

6. Определить пункты меню главного окна задач *TaskWindow.mnu*, инициализировать обработчики выбора этих пунктов меню *TaskWindow.win* и с помощью Expert Code написать тело для этих обработчиков в файле реализации *TaskWindow.pro*.

7. При написании обработчиков нужно учитывать следующее: каждый из них описан как **процедура**, т.е. имеет одно решение и это решение истинно (успешно). Чтобы получаемый обработчик работал, можно поступить так:

- предикат класса, помещаемый в тело обработчика, должен быть процедурой;
- обработчик должен состоять как минимум из двух утверждений, в конце тела первых утверждений надо поставить отсечение, исключив возврат при успешном завершении. Последнее утверждение должно быть фактом. Этот вариант подходит тогда, когда тестируемый предикат объявлен *determ;*
- обработчик должен состоять как минимум из двух утверждений, в конце тела первых утверждений надо поставить fail, проинициировав возврат. Последнее утверждение должно быть фактом. Этот вариант подходит тогда, когда тестируемый предикат объявлен *nondeterm или multi*.

8. Отладить полученную программу.

9. Так как управляющие элементы формы работают с текстовыми данными, то возникает необходимость преобразования данных от одного типа к другому. Для этого можно использовать следующие предикаты:

- *tostring(X)* – преобразует терм к строке;
- *toterm(X)* – преобразует строку в терм, тип которого определяется другими предикатами тела правила, либо его можно указать в предикате *hasdomain(<тип данного>,<Переменная>)* – показывает к какому типу данных относится переменная;
- *math::tointeger(X)* – преобразует строку в целое;
- *math::toReal(X)* – преобразует строку в действительное и т.д.

Заключение

В данной главе мы узнали основы GUI и то, как можно легко разработать GUI-программу в Visual Prolog. Мы обнаружили, что среда, предоставленная Visual Prolog, позволяет нам контролировать все GUI-компоненты, которые мы захотим использовать. Мы могли редактировать меню, диалоги и панели инструментов, чтобы они соответствовали смыслу нашей программы.

Затем мы смогли собрать из блоков код Пролога и вставить его в различные части программы, которые затем были безопасно инкапсулированы в различные обработчики. Неинтерактивное логическое ядро было размещено отдельно.

GUI-программа, разработанная таким образом, может использоваться очень гибко, без приведения к требованиям пользователя по отношению к последовательности действий.

Контрольные вопросы

1. Какую информацию нужно указать при создании проекта?
2. Что представляет собой дерево проекта? Как оно изменяется после сборки проекта?
3. В какой файл надо занести утверждения программы?
4. С помощью какой команды запускается на выполнение консольное приложение?
5. Для чего предназначено предложение ***open***?
6. Каков синтаксис описания предикатов в языке *Visual Prolog 7.0*?
7. Как подключить класс к файлу реализации?
8. Что такое режим детерминизма предиката?
9. Какие режимы детерминизма вы знаете?
10. Что такое шаблон потока данных? Как его задавать?
11. Какую информацию нужно указать при создании проекта?
12. Что представляет собой дерево проекта? Какого вида файлы имеются в нем? Каково их назначение?
13. Для чего предназначен пакет? Как создать пакет?

14. Для чего предназначена форма? Как создать форму?
15. Для чего предназначен класс? Как создать класс?
16. Как сделать предикаты класса доступными другим модулям или сущностям проекта?
17. Как подключить класс к файлу реализации?
18. Как создать обработчики выбора пунктов меню "Окна задач"?
19. Перечислите и охарактеризуйте управляющие элементы формы.
20. Как создать обработчики событий, связанных с управляющими элементами?

Задание

Разработать консольное приложение и приложение с графическим интерфейсом для предметной области, указанной преподавателем.

Приложение 1. ЧТО НОВОГО В VISUAL PROLOG 7.0

За последние годы язык программирования Visual Prolog проделал большой путь, и продолжит развиваться в будущем. Самое важное – это сдвиг к объектно-ориентированному программированию и введение параметрического полиморфизма.

Центр Развития Языка Пролог расширяет Visual Prolog и таким образом версия 7.0 содержит множество новых возможностей:

- полиморфизм;
- списки;
- конструкция "if-then-else";
- "or" с большой глубиной вложения;
- foreach (новая конструкция);
- понятность списков.

Также предлагается насладиться новыми преимуществами Vip7.0:

- более быстрый компилятор;
- контроль версий;
- *redBlackTree* PFC и *list* PFC пакеты (PFC - контроллер последовательности команд);
- поддержка переменных в IDE;
- COM упаковщик генерирует структурное выравнивание;
- Grid Control в качестве демо-примера.

Развитие языка привело к некоторым изменениям, которые могут повлиять на существующие проекты.

Полиморфизм

Visual Prolog 7.0 предоставляет параметрический ***полиморфизм***. Можно объявлять параметризированные ***domain`ы.***

Например:

```
domains
    binTree{Elem} = node(binTree{Elem} Left, Elem Node,
                binTree{Elem} Right);
    leaf().
```

Такой полиморфный domain замещает целый ряд (не полиморфных) domain-ов:

```
domains
    bintree_integer = node(bintree_integer Left, integer Node,
                bintree_integer Right);
    leaf().
```

```
domains
        bintree_string = node(bintree_string Left, string Node,
                bintree_string Right);
        leaf().
...
```

Более того, полиморфные предикаты могут управлять полиморфными структурами данных. Данный предикат добавляет узел в упорядоченное бинарное дерево:

```
predicates
        insert : (Elem NewNode, binTree{Elem} Tree) ->
                binTree{Elem} NewTree.
clauses
        insert(NewNode, leaf()) = node(leaf(), NewNode, leaf()).
        insert(NewNode, node(Left, Node, Right)) = NewTree :-
                NewNode <= Node,
                !,
                NewTree = node(insert(NewNode, Left), Node, Right).
        insert(NewNode, node(Left, Node, Right)) = NewTree :-
                NewTree = node(Left, Node,
                insert(NewNode, Right)).
```

Один вот такой предикат может добавлять элементы в бинарное дерево любого типа:

```
clauses
        ppp() :-  StrTree1 = insert("AAA", leaf()),
                StrTree2 = intert("BBB", StrTree1),
        IntTree1 = insert(17, leaf()),
        IntTree2 = insert(23, IntTree1),
        ...
```

Полиморфизм гарантирует безопасность типов, в том смысле, что нельзя добавить integer в дерево string.

```
clauses
        ppp() :-  StrTree1 = insert("AAA", leaf()),
        SomeTree2 = insert(23, StrTree1),
        %type error
        ...
```

Такие ошибки выявляются во ***время компиляции.***

В объектно-ориентированных языках, обычно принято создавать структуры данных вроде бинарного дерева с использованием ***object*** в качестве типа элементов.

```
domains
    bintree_object = node(bintree_object Left, object Node,
                bintree_object Right);
    leaf().
```

В виду того, что любой объект может быть преобразован к ***object***, то вы можете вставлять любой объект в такое бинарное дерево. Однако у этого способа есть свои отрицательные стороны:

– если у вас есть дерево "windows", вы можете добавить объект "person" в это дерево (т.к. они оба являются объектами);

– при извлечении элемента из дерева, у него будет тип ***object***, и его придется ***convert*** (преобразовывать) к его первоначальному типу.

Эти недостатки отсутствуют в полиморфных domain-ах: все элементы в дереве гарантированно имеют один и тот же свой тип, и когда запрашивается элемент из дерева, то у него будет точно такой же тип. (Этот тип, конечно, может быть и ***object***).

Списки

Описание(domain) списков можно сделать полиморфными. Благодаря этому допустимо "повторно использовать" предикаты для списков любого типа:

```
predicates
    isMember : (Elem Elem, Elem* List) determ.
clauses
    isMember(Elem, [Head|Tail]) :- Elem = Head
    or isMember(Elem, Tail).
```

Рассмотрим примеры использования предиката ***isMember***:
списки с типом integer:

```
isMember(17, [23, 45, 32, 17, 58])
```

списки с типом string:

```
isMember("aaa", ["asd", "asd", "asd"])
```

Но его нельзя использовать в следующем случае:

```
isMember("aaa", [23, 45, 32, 17, 58])
% type error
```

Напомним, что списки с элементами разного типа в языке Visual Prolog не существуют, например, такие как этот:

```
L = [23, "aaa", 32, 17, 58] % type error
```

Конструкция "if-then-else"

if-then-else представлена в качестве конструкции языка:

```
clauses
        p(X) = Y :-
                if X = 1
                then
                        Z = 3
                else
                        Z = 7
                end if,
                Y = 2*Z.
```

if-then-else может использоваться без части else:

```
clauses
        p(X) = Y :-
                if trace = true()
                then
                        writef("Started p(%)\n", X)
                end if,
                ...
```

Конструкция "or" (операция "или") с большой глубиной вложения

В Visual Prolog 7.0 стало возможным использовать "or" в операторах:

```
clauses
        p(X) = Y :-
                (X = 1, !, Z = 3 or Z = 7), Y = 2*Z.
```

Здесь использовано ключевое слово "or", но можно его заменить знаком ";".

Мы рекомендуем осторожно пользоваться "or", т.к. этот оператор главным образом предназначается для применения в конструкциях if-then-else:

```
clauses
        p(X) :-  if X = 1 or X > 17
                 then
                         ...
                 end if.
```

Конструкция foreach

Новая конструкция языка foreach может использоваться для организации циклов:

```
...,
foreach p_nd(X) do
        write(X), nl()
end foreach,
...
```

Конструкция "foreach" предназначается как замена предиката fail в циклах:

```
...,
p_nd(X)
write(X), nl()
fail.
```

Конструкция "foreach" будет выполняться столько раз, сколько предикат p_nd(X) будет согласовываться с утверждениями базы знаний. Как только цикл будет успешно завершен, так выполнение перейдет к предикату, следующему за foreach:

```
...,
foreach p_nd(X) do
        write(X), nl()
end foreach,
write("I will continue here"),
...
```

Конструкция "foreach" может быть использована и для организации вложенных циклов:

```
...,
foreach p_nd(X) do
        foreach q_nd(X, Q) do
                writef("Q = %, ", Q)
        end foreach,
        foreach r_nd(X, R) do
                writef("R = %, ", R)
        end foreach,
        writef("\nThat was for X = %\n", X)
end foreach
...
```

Создание списков с помощью нового предиката findall/3

Visual Prolog 7.0 предоставляет новый синтаксис и улучшенную функциональность, как дополнение к **findall/3.**

```
..., List = [ A || p_nd(A) ], ...
```

Строчка сверху соответствует:

```
..., findall(A, p_nd(A), List), ...
```

Новая конструкция возвращает значение и может использоваться следующим образом:

```
..., ppp( [ A || p_nd(A) ] ), ...
```

Были улучшены выполняемые функции, и теперь вы можете вычислять значения прямо на месте:

```
..., List = [ pair(A, math::sin(A)) ||
p_nd(A) ], ...
```

Можно использовать больше вызовов в "теле"

```
..., List = [ R || p_nd(A), S = math::sin(A),
R = pair(A, S) ], ...
```

Более "продвинутая" система типов с введением подтипов

Объявление domain вроде этого:

```
domains
    myType = someType.
```

скорее определит подтип (кроме случая, когда someType является составным или предикатным domain-ом), чем тип-синоним.

Значения типа "myType" могут автоматически использоваться как тип "someType", но значения типа "someType" должны явно преобразовываться к типу "myType".

Целью этого является увеличить безопасность программ. Рассмотрим следующий пример:

```
domains
    styleFlag = integer.
    color = integer.
predicates
    createWindow : (..., styleFlag Style, color Color).
```

"…" предполагает ряд других аргументов. Если **"styleFlag"** это синоним **"integer"**, и **"color"** также синоним **"integer"**, тогда

"styleFlag" и **"color"** являются синонимами друг другу. Таким образом, фиктивный вызов вроде этого

```
clauses
        ppp(...) :-
                Color = getColor(),
                StyleFlag = getStyleFlag(),
                createWindow(..., Color, StyleFlag). % bogus
```

является разрешенным. Теперь такое невозможно: **"color"** нельзя использовать в качестве **"styleFlag"** (и, наоборот) до тех пор, пока не выполнено явное преобразование.

Более строгая обработка числовых типов

Одно очень заметное последствие этого изменения заключается в том, что integer и unsigned отличаются друг от друга более строго. Если у вас есть integer, а вам нужно unsigned, то необходимо их явно преобразовывать.

Более быстрый компилятор

Компилятор Visual Prolog 7 работает быстрее, чем в Visual Prolog 6, точная оценка меняется от файла к файлу.

Контроль версий

Контроль версий – это удобный способ представить информацию о версии в проекте. Новый IDE проект содержит диалог About с параметрами версии.

***RedBlackTree* PFC и *list* PFC пакеты (PFC - контроллер последовательности команд)**

Быстрый способ получить доступ к набору данных очень необходим. Visual Prolog 7 предоставляет redBlackTree пакет для поддержки таких наборов.

Возможность объявления domain-а общего списка позволяет использовать общий PFC пакет для обработки списков.

Поддержка переменных в IDE

IDE переменные могут использоваться в настройках проекта в (Project Settings) для директорий включения (Include directories) и панелей инструментов (Tools Commands), таким образом, становится более удобно регулировать IDE и Project настройки для работы над большими проектами.

Grid Control в качестве демо примера

На VIP ALC конференции осведомились об объектно-ориентированном GridControl, и теперь он включен в демо-примеры, которые доступны путем выполнения установки setup Examples для Visual Prolog 7.0 Commercial Edition.

Проблема обратной совместимости

1. Не поддерживаются ссылочные domain-ы (reference domain).
2. Были введены новые служебные слова, которые нельзя больше использовать в качестве идентификаторов:

СПИСОК ЛИТЕРАТУРЫ

1. Вагин В.Н. Дедукция и обобщение в системах принятия решений. – М.: Наука. Гл. ред. физ.-мат. лит. 1988. – 384 с.
2. Нильсон Н. Принципы искусственного интеллекта: Пер. с англ. – М.: Радио и связь, 1985. – 376 с., ил.
3. ЭВМ пятого поколения: Концепции, проблемы, перспективы/ Под ред. Т. Мотто-ока. Пер.с англ.. – М.: Финансы и статистика, 1984. – 110 с.
4. Цуканова Н.И., Дмитриева Т. А. Логическое программирование на языке Visual Prolog. Учебное пособие для вузов. – М.: Горячая линия – Телеком, 2008. – 144 с.:ил.
5. Братко И. Алгоритмы искусственного интеллекта на языке PROLOG.-М.: Вильямс, 2004. - 640с.
6. Шрайнер П.А. Основы программирования на языке Пролог. Курс лекций. Учебное пособие.-М.: Интренет-университет информационных технологий, 2005. - 170с.
7. Адаменко А., Кучуков А. Логическое программирование и Visual Prolog.-СПб.: BHV-СПб, 2003. - 992 с.
8. Братко И. Программирование на языке Пролог для искусственного интеллекта.-М.: Мир,1990. - 560с.
9. Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог.-М: Мир,1990. - 235с.
10. Малпас Дж. Реляционный язык Пролог и его применение.- М.: Наука.,1990. - 464с.
11. Стобо Дж. Язык программирования Пролог.-М.: Радио и связь, 1993. - 368с.
12. Доорс Дж. и др. Пролог – язык программирования будущего.-М.: Финансы и статистика, 1990. - 144с.
13. Янсон Я. Турбо-Пролог в сжатом изложении.-М.: Мир, 1991. - 94с.
14. Клоксин У., Меллиш К. Программирование на языке Пролог.-М.: Мир, 1987. - 336с.
15. Кларк К., Маккейб Ф. Введение в логическое программирование на микро-Прологе.-М.: Радио и связь, 1987. - 312с.
16. Логическое программирование. Сборник статей.-М.: Мир, 1988. - 368с., ил.
17. Язык Пролог в пятом поколении ЭВМ. Сборник статей.-М.: Мир, 1988. - 501с., ил.
18. Грэй П. Логика, алгебра и базы данных.-М.: Машиностроение, 1989. - 368с., ил.

СОДЕРЖАНИЕ

Вышли в свет и имеются в продаже:

Инженерия автоматизированных информационных систем в *e*-экономике / Под редакцией Эдварда Колбуша, Войцеха Олейничака и Здислава Шиевского; пер. с польск. И. Д. Рудинского. – М.: Горячая линия–Телеком, 2012. – 376 с.: ил., ISBN 978-5-9912-0191-9.

Книга представляет собой зарубежный учебник, содержащий комплексное руководство по современным информационным стратегиям, системам, технологиям и средствам. Авторы рассматривают инженерию автоматизированных информационных систем как прикладную дисциплину и, одновременно, как искусство проектирования, сопровождения и развития приложений. В книге систематизированы знания о функционировании автоматизированных информационных систем в *e*-экономике и даны практические рекомендации по проектированию нового поколения автоматизированных информационных систем, применению баз данных и хранилищ данных, созданию *e*-бизнес приложений и методов руководства процессом разработки автоматизированных информационных систем.

Книга адресована студентам высших технических и экономических учебных заведений, менеджерам информационных проектов и практикующим разработчикам автоматизированных информационных систем.

Логическое программирование на языке Visual Prolog. Учебное пособие для вузов / Н. И. Цуканова, Т. А. Дмитриева. – М.: Горячая линия – Телеком, 2008. – 144 с.: ил., ISBN 978-5-9912-0033-2.

Изложены основы логического программирования на примере языка Visual Prolog. Рассмотрены описание предметной области и структура программы, алгоритм работы интерпретатора, ввод – вывод, приемы и средства организации интерактивных программ, вопросы недетерминированного программирования и управления выполнением программы, различные структуры данных и предикаты работы с ними. Книга содержит многочисленные примеры, иллюстрирующие теоретические положения их решения, а также контрольные вопросы и практические задания. Многие примеры ориентированы на создание реляционной базы данных и написание различных запросов к ней. Пособие может быть полезно при изучении курса «Функциональное и логическое программирование».

Для студентов высших учебных заведений, программистов, специалистов в области искусственного интеллекта и баз данных.

От С к С++: Учебное пособие для вузов / И. Ю. Каширин, В. С. Новичков. – 2-е изд., стереотип. – М.: Горячая линия – Телеком, 2012. – 334 с.: ил., ISBN 978-5-9912-0259-6.

Учебное пособие содержит необходимые теоретические сведения и набор упражнений и задач различной степени сложности, позволяющих приобрести навыки практического программирования на алгоритмических языках С и С++ (Си и Си++) и проконтролировать усвоение материала. Практические задания для программирования на С++ имеют «сквозную» структуру – распределены по мере изложения разделов. Материал книги успешно апробирован авторами в высших технических учебных заведениях.

Для студентов высших и средних учебных заведений, может быть использована начинающими программистами при изучении алгоритмических языков С и С++.

Сервер приложений «Zope». Учебное пособие для вузов / С. Э. Грегер. – М.: Горячая линия – Телеком, 2009. – 256 с.: ил., ISBN 978-5-9912-0112-4.

Рассмотрен язык программирования Python и вопросы применения приложения ZOPE (Z Object Publishing Environment) для разработки Web-приложений. Дано описание синтаксиса языка Python, построения структур данных, разбираются вопросы объектно-ориентированного программирования. Рассмотрены принципы работы сервера приложений Zope, его объектная модель и интерфейс управления. Обсуждаются вопросы синтаксиса, приведены примеры использования языков разработки шаблонов DTML и TAL, взаимодействие с реляционными базами данных, создание поисковых служб. Рассмотрены вопросы настройки системы безопасности сервера. Пособие содержит большое количество примеров и контрольные вопросы по изучаемым темам, что может служить основой для построения лабораторного практикума.

Для студентов вузов, обучающихся по направлению 230100 – «Информатика и вычислительная техника», специальности 230105 – «Программное обеспечение вычислительной техники и автоматизированных систем», может быть полезна специалистам.

Справки по телефону: (495) 737-39-27, WWW.TECHBOOK.RU