

Министерство образования и науки Российской Федерации  
Федеральное государственное бюджетное образовательное  
учреждение высшего профессионального образования  
«Иркутский государственный университет»

Учреждение Российской академии наук  
«Институт динамики систем и теории управления  
Сибирского отделения РАН»

**Е. А. Черкашин**

## **Рекурсивно-логическое программирование**

Учебное пособие

**Иркутск 2013**

УДК 681.3.06 (075.8)

ББК 32.973-01я73

Ч-48

Печатается по решению ученого совета ИМЭИ

**Издание выходит в рамках Программы  
стратегического развития ФГБОУ ВПО «ИГУ»  
на 2012–2016 гг., проект Р121-02-001**

**Рецензенты:**

канд. техн. наук *В. С. Ульянов,*  
канд. физ.-мат. наук *А. А. Лемперт*

**Черкашин Е. А.**

Ч-48      Рекурсивно-логическое программирование : учеб. пособие / Е. А. Черкашин. – Иркутск : Изд-во ИГУ, 2013. – 109 с.  
**ISBN 978-5-9624-0938-2**

В пособии представлены лекционные материалы и лабораторные работы курса «Рекурсивно-логическое программирование»: базовые термины искусственного интеллекта, задачи, методы и их свойства; основы рекурсивно-логического программирования на языке Пролог; типичные задачи, решение которых лаконично представляется как рекурсивные и переборные алгоритмы. Пособие содержит задания на лабораторный практикум по темам «Формализация», «Обработка списков», «Метод Британского музея (отобразить и проверить)» и «Базы данных».

Пособие предназначено для студентов специальности «инженер-программист», «инженер — системный программист». Изучение материала будет полезно студентами других специальностей, так или иначе связанных с программированием, формальной логикой и комбинаторикой.

УДК 681.3.06 (075.8)

ББК 32.973-01я73

ISBN 978-5-9624-0938-2

© Черкашин Е. А., 2013

© ФГБОУ ВПО «ИГУ», 2013

© Институт динамики систем и теории управления  
СО РАН, 2013

# Оглавление

<b>Предисловие</b>	<b>4</b>
<b>1. Введение в область искусственного интеллекта</b>	<b>7</b>
1.1. Термин «искусственный интеллект» . . . . .	9
1.2. Данные и знания . . . . .	14
1.3. Формализмы представления знаний . . . . .	18
<b>2. Язык программирования Пролог</b>	<b>25</b>
2.1. Основные термины . . . . .	25
2.2. Структура языка . . . . .	26
2.3. Программа на языке Пролог . . . . .	29
2.4. Унификация и мэтчинг . . . . .	36
2.5. Лабораторная работа 1: Факты и правила . . . . .	39
<b>3. Списки и их обработка</b>	<b>46</b>
3.1. Грамматический разбор текста . . . . .	55
<b>4. Интерпретации пролог-программ</b>	<b>64</b>
4.1. Декларативная интерпретация . . . . .	65
4.2. Процедурная интерпретация . . . . .	67
<b>5. Управление логическим выводом</b>	<b>74</b>
5.1. Ограничение перебора . . . . .	74
5.2. Лабораторная работа 2: Списки . . . . .	83
<b>6. Предикаты с побочными действиями</b>	<b>87</b>
6.1. Базы данных в Прологе . . . . .	89
6.2. Лабораторная работа 3: Базы данных . . . . .	90
<b>7. Поиск решения на основе перебора</b>	<b>98</b>
7.1. Алгоритм «Британского музея» . . . . .	99
7.2. Лабораторная работа 4: Перебор . . . . .	103
<b>Заключение</b>	<b>108</b>
<b>Рекомендуемая литература</b>	<b>109</b>

# Предисловие

Предлагаемое учебное пособие разработано для студентов специальности «инженер-программист», «инженер — системный программист», однако может быть использовано всеми заинтересованными программистами, желающими овладеть некоторыми методами искусственного интеллекта. Пособие подборку материала по курсу «Рекурсивно-логическое программирование», а также варианты лабораторных работ и методические указания по их выполнению. Оно никоим образом не претендует на полноту излагаемого материала и базируется на личном опыте преподавания. Пособие следует воспринимать как путеводитель, и учащиеся в процессе обучения должны активно использовать литературу, на которую в тексте указаны ссылки. В цитируемом тексте в виде сносок автор позволяет себе высказывать свое отношение к изложенному.

В пособие включены как разделы, разработанные лично автором пособия, так и заимствованные из других источников и адаптированные к современному состоянию, в частности из книг И. Братко, Дж. Малпаса, справочников и т. д. Везде, где произведено такое заимствование, в первом абзаце указана ссылка на оригинальный источник. При подготовке данного пособия автор придерживался той точки зрения, что использованный материал является наиболее простым для понимания студентами, лаконично представляет суть изучаемого материала, поэтому его дальнейшее совершенствование или изложение как-то по-новому не имеет никакого смысла. Автор не преследует целью коммерческое использование учебного пособия. В электронной версии пособия ссылки на литературу в списке литературы — активные и ведут к найденным в Интернете электронным версиям книг. Читатель должен решить самостоятельно: скачивать их из Интернета, покупать в магазинах или искать в библиотеках.

Автор является приверженцем открытых технологий, свободных книг, научного метода познания мира и открытого программного обеспечения. Адрес исходного кода методического пособия — <https://github.com/eugeneai/ais/tree/new-isu>. Исходный код разрешено использовать в соответствии с лицензией

CC BY-NC-SA 4.0 (Attribution-NonCommercial-ShareAlike 4.0 International), которая разрешает использование материала в своих произведениях (необходимо указывать автора оригинальных материалов), запрещает коммерческое использование материалов (ввиду наличия вышеперечисленных заимствований) и требует распространение производных материалов производить по этой же самой лицензии (по предыдущей причине). Адрес лицензии — <http://creativecommons.org/licenses/by-nc-sa/4.0/>.

В тексте пособия использована следующая разметка:

**Жирным шрифтом** выделяются имена существительные и глаголы, на которые, по мнению автора, следует обратить внимание, это — что-то вроде дополнительной семантической разметки текста учебного пособия.

Моноширинным шрифтом приводятся программы, отрывки программ в основном тексте пособия, а также имена идентификаторов, т. е. все, что имеет какое-либо отношение к **тексту программы**.

*Наклонным шрифтом* выделяются **новые** термины, вводимые в текст и возникающие, например, в определениях, а также текст выделенных примеров.

При помощи «кавычек» выделяются метафоры, значения, элементы текстов программ, цитаты, слова, использованные в переносном смысле, и т. д.

Надеюсь, что изучение такого интересного раздела информатики как «Искусственный интеллект», к которому относится рекурсивно-логическое программирование, доставит учащимся не меньше удовольствия, чем в свое время автору этого учебного пособия.

Автор признателен своему научному руководителю академику Российской академии наук С. Н. Васильеву за создание плодотворной почвы для проведения научных исследований в области систем искусственного интеллекта. Автор благодарит преподавателей Института кибернетики, Кафедры вычислительной

техники Национального исследовательского Иркутского государственного технического университета, а также преподавателей Института математики, экономики и информатики Иркутского государственного университета за предоставление возможности изучать область искусственного интеллекта.

Автор благодарит канд. геол.-минерал. наук Т. Ю. Черкашину, канд. техн. наук А. К. Попову за помощь в подготовке данного учебного пособия.

Старший научный сотрудник ИДСТУ СО РАН,  
кандидат технических наук  
Е. А. Черкашин

P. S. Автора всегда можно найти по адресу [eugeneai@icc.ru](mailto:eugeneai@icc.ru), в поле «тема» прошу указывать «РЛП-2013».

# 1. Введение в область искусственного интеллекта

Среди задач, которые решают современные программисты, выделяются задачи создания программных систем математического моделирования и прогнозирования, проектирования и реализации информационных систем и баз данных, системного программного обеспечения. Все перечисленные задачи объединяет одно общее свойство — для широкого практического класса задач можно построить детерминированную процедуру (например, алгоритм) их решения. Существует большой класс задач, для которых такую процедуру построить достаточно сложно, а порой и невозможно. Например, разработать игровую систему, способную играть в шахматы с человеком на достаточно высоком профессиональном уровне. К таким задачам относятся также и задачи поиска решения (планирование действий, или Problem Solving), распознавание образов, экспертные консультации, интеллектуальное управление сложными динамическими объектами и т. д. В каждой такой задаче четко вырисовывается их первое общее **свойство** — необходимость **автоматизации принятия некоторого решения**. В других задачах четко вырисовывается еще одно свойство — **обработка символьной информации**. Примерами задач, обработка информации в которых основывается на преобразовании строк символов, выступают следующие задачи: автоматический перевод текста с одного естественного языка на другой, автоматическое доказательство теорем. В той или иной мере оба выделенных свойства присутствуют в каждой из перечисленных задач.

Средства рекурсивно-логического программирования позволяют представить решение таких задач, алгоритм, в рекурсивном виде или в виде некоторого переборного процесса. Такое представление обладает одним полезным свойством — оно компактно и достаточно близко к исходной математической модели задачи по сравнению с изученной ранее процедурной парадигмой программирования. Программисту не требуется определять все действия, необходимые для достижения результата. Как правило,

достаточно рассказать транслятору, какие данные есть в наличии, объяснить, как они связаны друг с другом и постановкой задачи. Система постарается получить решение самостоятельно. Рекурсивно-логическое программирование прежде всего направлено на решение задач искусственного интеллекта (ИИ)<sup>1</sup>, поэтому в данном учебном пособии необходимо ввести читателя в базовые концепции ИИ. Для начала рассмотрим, как можно определить, относится ли ваша задача к задачам ИИ.

Рассмотрим задачи планирования действий (Problem Solving). Что есть решение в этих задачах? Это — ответ на вопрос «Какие действия необходимо выполнить и в каком порядке их надо выполнять, чтобы достичь цели из некоторого начального состояния?». Получается, что ответ на этот вопрос есть некоторая конечная последовательность действий. Эта последовательность представляется в памяти компьютера в виде некоторого ряда чисел, кодирующего эту последовательность. Построить (найти) эту последовательность, выбрать последовательность из возможных альтернатив — это и есть принятие решения.

Есть еще один интересный аспект алгоритма — массовость, т. е. алгоритмы должны строиться для некоторого класса задач, а не для конкретных входных данных. Что это значит? На вход программы, реализующей алгоритм, подаются какие-либо входные данные, задающие конкретную задачу из класса решаемых алгоритмом задач. Теперь представим такую ситуацию, что на вход алгоритма невозможно подать все необходимые данные, т. е. имеет место *неполнота информации*. Или другой вариант — имеется два эксперимента с разными результатами, но с одинаковым набором исходных данных. Какие данные следует передавать на вход алгоритма? Этот случай связан с *противоречивой информацией*. Разрабатывая программное обеспечение, позволяющее функционировать в таких условиях, приходится создавать подпрограммы, принимающие решение, что следует делать. Например, во втором случае можно запустить алгоритм для каждого набора данных и проанализировать полученные результаты. Может получиться так, что эти результаты не будут сильно отли-

---

<sup>1</sup> В англоязычной литературе данный термин называется *Artificial Intelligence, AI*.



чаться друг от друга, а может этого не получиться. В последнем случае одним из вариантов дальнейших действий является поиск нового атрибута (характеристики), который позволит различать варианты, находящиеся в противоречии.

Задачи, обладающие перечисленными свойствами, и методы их решения на ЭВМ в конечном счете составляют предмет исследования искусственного интеллекта — одного из разделов информатики (Computer Science).

## 1.1. Термин «искусственный интеллект»

В литературе можно найти целый спектр определений термина искусственный интеллект, однако, насколько известно автору, ни один из них не принят как стандарт.

Среди многих точек зрения доминируют три [4]. Согласно **первой**, исследования в области искусственного интеллекта являются фундаментальными исследованиями, в рамках которых разрабатываются модели и методы решения задач, традиционно считавшихся интеллектуальными и не поддававшихся ранее формализации и автоматизации. Согласно **второй** точке зрения, новое направление связано с новыми идеями решения задач на ЭВМ, с разработкой принципиально иной технологии программирования, с переходом к архитектуре ЭВМ, отвергающей классическую архитектуру, которая восходит еще к первым ЭВМ. Наконец, **третья точка зрения**, по-видимому, наиболее прагматическая, состоит в том, что в результате работ в области искусственного интеллекта рождается множество прикладных систем, решающих задачи, для которых ранее создаваемые системы были непригодны.

Достаточно простые определения *искусственного интеллекта* показаны в таблице 1.1 [10]. Выделяются несколько комбинаций двух пар ключевых терминов: «размышлять» и «вести себя», «как человек» и «рационально».

*Искусственный интеллект* как наука насчитывает уже около 60 лет. Задачей этой науки является воссоздание с помощью ис-

Таблица 1.1

Несколько определений искусственного интеллекта

Системы, которые размышляют, как люди	Системы, которые размышляют рационально
Системы, которые ведут себя, как люди	Системы, которые ведут себя рационально

кусственных устройств (в основном с помощью ЭВМ) разумных рассуждений и действий [5].

*Искусственный интеллект* — раздел информатики, изучающий методы, способы и приемы моделирования и воспроизведения с помощью ЭВМ разумной деятельности человека, связанной с решением задач [7].

### 1.1.1. Тест Тьюринга

В книге [10] вводится понятие *агента*. *Агент* — субъект, находящийся в среде, имеющий цель своего существования, взаимодействующий со средой или другими агентами с помощью *рецепторов* и *эффекторов*. Рецепторы воспринимают информацию о среде, а эффекторы — это способ воздействия на среду, которое меняет среду, а следовательно, и информацию о среде. Агентом может являться как программа, так и человек. Вводится понятие *интеллектуального агента* — агента, обладающего интеллектом.

Агенты взаимодействуют друг с другом. Примером такого взаимодействия выступают, например, общение человека с человеком или работа человека с компьютерной программой.

Тест Тьюринга предложен Аланом Тьюрингом (1950), и был разработан, чтобы представить действующее определение интеллекта [10]. Тьюринг определял интеллектуальное поведение как возможность достижения человеческого уровня производительности во всех задачах, где возможно обмануть человека, задающего вопросы. Грубо говоря, предложенный им тест состоял в следующем. Компьютеру задает вопросы человек через удаленное устройство. Тест считается пройденным, если человек не

может сказать, кто или что на другом конце устройства: компьютер или человек.

С точки зрения агентов, этот тест можно представить так: один интеллектуальный агент (человек) по информационному каналу, не позволяющему ему использовать иную информацию, кроме ответов на поставленные им вопросы, анализирует поступающую информацию (ответы собеседника) от другого агента (испытуемого). Если первый агент не в силах определить, кто на другом конце информационного канала — человек или устройство, тогда считается, что испытуемый агент обладает интеллектуальными свойствами.

### 1.1.2. Применение искусственного интеллекта

Всякая задача, для которой неизвестен алгоритм решения, априорно относится к ИИ. Перечислим некоторые направления (задачи) ИИ [4]:

**Восприятие и распознавание образов.** К таким задачам относятся распознавание текста (как печатного, так и рукописного), компьютерное зрение.

**Автоматическое доказательство теорем.** В этом направлении решаются задачи автоматизации математических исследований, разработки формальных (математических) методов логического вывода для поддержки решения других задач ИИ. Это направление нашло применение в задачах верификации программного и аппаратного обеспечения.

**Игры.** Автоматизация решения игровых задач, например, игры в шахматы, калаш, реверси, а также других игр.

**Решение задач (Problem Solving), планирование.** В этих задачах предполагается наличие некоторого выбора из возможных путей решения, требуется найти первое, лучшее или оптимальное решение. Примеры: составление расписания работы учебного учреждения, планирование действий автономного необитаемого аппарата.

**Понимание естественного языка.** Как правило, системы понимания естественного языка являются составляющими информационных систем различного назначения: от автоматических систем заказа билетов до систем ввода экспертного знания.

**Логическое программирование.** Языки и системы программирования высокого выразительного уровня, построенные на основе результатов исследования формально-логических систем, теорий исчислений. Эта область ИИ носит, кроме прочего, инструментальный характер, т. е. логическое программирование является средством реализации систем ИИ.

**Экспертные системы.** Экспертные системы (ЭС) позволяют заменять человека-эксперта в некоторой предметной области программной системой, способной проводить экспертные консультации пользователя. ЭС нашли широкое применение в индустрии.

**Интеллектуальные информационные системы.** Эти системы объединяют разнородные интеллектуальные системы (например, системы речевого общения, решения задач и др.) для организации интеллектуального доступа, обработки информации. Они, как правило, предназначены для работы с конечным пользователем низкой квалификации. Пример: электронные переводчики и разговорники.

**Восприятие и усвоение знаний.** Одна из задач ИИ — это приобретение знаний<sup>1</sup> (обучение, наполнение базы знаний) от человека или самостоятельно из среды функционирования. Системы усвоения знаний используются как подсистемы других интеллектуальных систем.

**Интеллектуальное управление [3].** Новое направление, появившееся на стыке ИИ и теории управления, в котором разрабатываются управляющие системы, основанные на тех или

---

<sup>1</sup> Англ. — Knowledge Acquisition.

иных методах ИИ. В настоящее время наиболее развиты методы управления на основе нечеткой логики и искусственных нейронных сетей. В этом новом направлении ведутся научные разработки Института динамики систем и теории управления СО РАН (<http://www.idstu.irk.ru>).

**Робототехника (Robotics).** Собирательное направление исследований, задачей которого является автоматизация функционирования роботов, вплоть до полной независимости их от человека.

### 1.1.3. Определение задач ИИ в контексте пособия

К сфере искусственного интеллекта относятся задачи, обладающие следующими свойствами [5]:

- в них используется информация в символьной форме: буквы, слова, знаки, рисунки. Это отличает область ИИ от областей, в которых традиционно компьютерам доверяется обработка данных в числовой форме<sup>1</sup>;
- в них предполагается наличие выбора; действительно, сказать, что не существует алгоритма<sup>2</sup>, — это значит сказать, по сути дела, только то, что нужно сделать выбор между многими вариантами в условиях неопределенности, и этот недетерминизм, который носит фундаментальный характер, эта свобода действия являются существенной составляющей интеллекта.

Излагаемый в пособии курс подразумевает непосредственное изучение технических аспектов применения методов ИИ, таких как применимость того или иного метода в конкретной задаче, реализация программных модулей конкретного метода. Поэтому в предлагаемом курсе мы будем использовать следующее определение искусственного интеллекта:

---

<sup>1</sup> Например, математическое моделирование, базы данных.

<sup>2</sup> Интуитивное определение алгоритма: *алгоритм* — это конечная последовательность действий, каждое из которых выполняется за конечное время, приводящая к определенному результату.

*Искусственный интеллект* — область информатики, в которой разрабатываются и исследуются методы построения программных систем и решения задач так или иначе связанных с принятием решения и обработкой символьной информации<sup>1</sup>.

## 1.2. Данные и знания

Одним из фундаментальных терминов ИИ является термин *знание*. Данный термин также является сложным в смысле его конструктивного определения, понятного читателю. Как правило, авторы статей по ИИ сознательно уклоняются давать более или менее точные определения, предполагая, что читателю это уже известно.

Более формальный термин «данные» получил широкое распространение в научно-техническом обиходе, в особенности в практике использования ЭВМ для решения самых разнообразных задач [4]. При этом вся обрабатываемая информация называется данными: начальными, промежуточными или конечными, входными или выходными. Для предложений естественного языка более привычен термин «знание», а чаще — связанный с ним глагол «знать». Ни у кого не вызывает возражений использование этого слова в предложениях вроде «я знаю, как решить задачу», или «я знаю, что вчера Петя встречался с Наташей». Сомнению может подвергаться лишь истинность подобных утверждений, но никак не возможность сочетания слова «знать» с фрагментами предложения, обозначающих любую информацию, о которой говорится, что она кому-то известна.

Вопрос о разделении информации на данные и знания возник при разработке систем ИИ, определяемых в последнее время как системы, основанные на знаниях<sup>2</sup>. Был предложен ряд определений, отражающих различные аспекты этих понятий, но

---

<sup>1</sup> Данное определение задает термин в достаточно узком смысле.

<sup>2</sup> На самом деле это определение не охватывает такую важную отрасль ИИ, как нейронные сети.



касающихся скорее форм (см. раздел 1.3 представления данных и знаний, правил их использования, чем их сути.

## Два подхода к разработке методов и средств ИИ

**«Снизу-вверх».** Суть подхода выражена в фразе *«Давайте создадим механическое (вычислительное) устройство, похожее на (моделирующее) мозг человека, а затем посмотрим, как оно будет решать задачи ИИ».*

**«Сверху-вниз».** В основе этого подхода лежит *разработка методов моделирования процесса мышления человека (логических выводов, логических рассуждений).*

Методы и системы ИИ, основанные на подходе «Снизу-вверх», как правило, представляют собой сложную сеть взаимосвязанных, простых по сути, элементарных агентов. Эта сеть агентов формирует агента высокого уровня, направленного на решение конкретной задачи ИИ. Элементарные агенты сети вносят небольшой персональный вклад в решение агента высокого уровня. Выделяют одно из достоинств этого подхода — *если задача «не решается» какими-то формальными методами, то ее «хоть какое-то» решение может быть получено методами «Снизу-вверх».* Как правило, схема применения описываемых методов и систем состоит из двух этапов: **обучение** на известном наборе «данные — решения» (данные и решения известны) и **решения** новых задач (данные известны, решения — нет).

Известным недостатком, присущим методам и системам «Снизу-вверх», является неопределенность характеристик с точки зрения их практического применения: трудно ответить, например, на вопросы: «Сколько нужно агентов, чтобы решить конкретную задачу? Каковы должны быть связи между агентами?». В каждом конкретном случае требуются эмпирические исследования («сможет — не сможет»).

Типичным представителем подхода «Снизу-вверх» являются нейронные сети.

Моделирование логических выводов и рассуждений — основа подхода «Сверху-вниз». В системах ИИ (агентах), осно-

вывающихся на этом подходе, как правило, **четко выделяют** функциональные блоки «Хранилище Базы знаний», «Машина логического вывода» и интерфейс «Рецептор — Блок рассуждений — Эффектор». В задачу последнего блока входит преобразование информации в/из вид(а), используемый(ого) в первых двух блоках. Именно в этих методах и системах ИИ возникает задача представления знаний в некотором формализованном виде, удобном для осуществления их интерпретации и преобразований в блоке «Машина логического вывода». Примерами систем «Сверху-вниз» выступают язык программирования Пролог, экспертные системы, системы автоматического логического вывода.

Исходя из общих соображений, естественно определить данные как некоторые сведения об отдельных объектах, а знания — о мире в целом. В согласии с таким подходом будем считать, что:

*Данные* представляют информацию о существовании объектов с определенными комбинациями свойств (значений признаков), а *знания* — информацию о существующих в мире закономерных связях между признаками, запрещающих некоторые другие сочетания свойств у объектов.

Отсюда следует, что различие между данными и знаниями можно сформулировать так: *данные* — это информация о существовании объектов с некоторым набором свойств, а *знания* — информация о несуществовании объектов с некоторым набором свойств<sup>1</sup>.

Используя логический формализм (см. далее) представления знаний, продемонстрируем эти понятия в формализованном виде. Отображая наличие упомянутых наборов предикатами  $P$  и  $Q$ , можно представлять *данные* утверждениями с кванторами существования  $\exists$ :

$$\exists w : P(w),$$

а *знания* — утверждениями с его отрицанием  $\neg\exists$ :

$$\neg\exists w : Q(w),$$

---

<sup>1</sup> С научной точки зрения данное предложение не является определением, оно неконструктивно, т. е. не задает логических связей с известными объектами и терминами. В частности, термин «навык» тоже подходит под это же определение.



легко преобразуемыми в утверждения с квантором  $\forall$ :

$$\forall w : \neg Q(w).$$

Вообще говоря, последняя форма (с использованием отрицания некоторого высказывания) практически не используется. Однако используется форма, подобная этой —  $\forall w : A(w) \rightarrow B(w)$ . Здесь, по сути, важен факт присутствия квантора всеобщности, говорящего, что **все объекты**  $w$ , обладающие свойством  $A$ , будут обладать свойством  $B$ .

Тремя базовыми элементами как практической, так и теоретической рациональной деятельности являются<sup>1</sup> следующие [9]:

*Данные*, которые должны прежде всего храниться, а затем, в порядке убывания приоритетов для непосредственной применимости, успешно находиться при нужде, проверяться, поддерживаться в порядке и обновляться при необходимости. Таким образом, они хранятся неизменными, пока не будут явно обновлены, и поэтому обычно внимание уделяют прежде всего сохранению, поддержанию их адекватности меняющемуся состоянию дел и целостности при необходимых изменениях.

*Знания* должны прежде всего преобразовываться. Далее, их нужно хранить, как и данные, они должны быть доступными, они должны конкретизироваться применительно к данной ситуации и обобщаться для целого класса применений. Они, конечно же, должны при необходимости пересматриваться. И, наконец, они должны переводиться с одного языка на другой.

*Умения* прежде всего применяются. Помимо этого, они преобразуются для обеспечения гибкости или приспособления к изменившимся условиям. Далее, они обобщаются и пересматриваются.

При исследовании естественных предметных областей данные представляют первичную информацию, получаемую путем обнаружения некоторых объектов и выявления их свойств — измерения значений признаков. Знания — результат переработки данных, их обобщения. Классическим примером данных служат таблицы движения планет по небесному своду Тихо Браге, при-

<sup>1</sup> Одно из последних, найденных автором пособия в [9], определений.

мером знаний — выведенные из них законы Иоганна Кеплера и затем, как обобщение результатов Кеплера, закон всемирного тяготения Исаака Ньютона.

### 1.3. Формализмы представления знаний

В интеллектуальных системах (ИС) используются различные способы представления знаний. Наиболее известные из них — это *логический, сетевой, продукционный и фреймовый формализмы*<sup>1</sup> (модели) представления знаний.

#### 1.3.1. Логические модели

В основе такого типа лежит формальная система, задаваемая четверкой типа  $M = \langle T, P, A, B \rangle$  [4]. Множество  $T$  есть множество базовых элементов различной природы, например, слов из некоторого ограниченного словаря, деталей детского конструктора, входящих в состав некоторого набора, и т. п. Важно, что для множества  $T$  существует некоторый способ определения принадлежности или непринадлежности 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  $\Pi(T)$ .

Множество  $P$  есть множество синтаксических правил. С их помощью из элементов  $T$  образуют синтаксически правильные совокупности. Например, из слов ограниченного словаря строятся синтаксически правильные фразы, из деталей детского конструктора с помощью гаек и болтов собираются новые конструкции. Декларируется существование процедуры  $\Pi(P)$ , с помощью которой за конечное число шагов можно получить ответ на вопрос, является ли совокупность  $x$  синтаксически правильной.

Во множестве синтаксически правильных совокупностей выделяется некоторое подмножество  $A$ . Элементы  $A$  называются

---

<sup>1</sup> Некоторый формальный способ записи чего-либо.

аксиомами. Как и для других составляющих формальной системы, должна существовать процедура  $\Pi(A)$ , с помощью которой для любой синтаксически правильной совокупности можно получить ответ на вопрос о принадлежности ее к множеству  $A$ . Множество  $B$  есть множество правил вывода. Применяя их к элементам  $A$ , можно получать новые синтаксически правильные совокупности, к которым снова можно применять правила из  $B$ . Так формируется множество выводимых в данной формальной системе совокупностей. Если имеется процедура  $\Pi(B)$ , с помощью которой можно определить для любой синтаксически правильной совокупности, является ли она выводимой, то соответствующая формальная система является разрешимой. Это показывает, что именно правила вывода являются наиболее сложной составляющей формальной системы.

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

**Пример 1.1.** *Здесь и далее рассмотрим, как можно представить следующий набор высказываний (данных и знаний)<sup>1</sup>:*

1. *Все люди смертны.*
2. *Сократ — человек.*

Для начала введем наш язык (элементы детского конструктора). Пусть  $H(x)$  обозначает фразу « $x$  является человеком», а  $M(x)$  — « $x$  — смертен». Необходимо заметить, что  $x$  как элемент языка представляет объекты предметной области, которыми

<sup>1</sup> В разделе рассматривается только представление знаний. Тот факт, что Сократ смертен, нас не интересует...пока.

в нашем случае выступают люди<sup>1</sup>. Можно выделить и отдельные объекты, например Сократа — « $s$ ». Теперь позаимствуем еще несколько символов из исчисления предикатов первого порядка, а именно  $\forall$ ,  $\rightarrow$ , скобки и др. Теперь первое высказывание представляется как

$$\forall x (H(x) \rightarrow M(x)).$$

Это высказывание является *знанием*. Второе высказывание —

$$H(s).$$

Это — *данные*, т. е. существует Сократ, который смертен.

### 1.3.2. Сетевые модели

В основе моделей этого типа лежит конструкция, названная ранее семантической сетью. Сетевые модели формально можно записать в виде

$$H = \langle I, C_1, C_2, \dots, C_n, \Gamma \rangle.$$

Здесь  $I$  есть множество информационных единиц;  $C_1, C_2, \dots, C_n$  — множество типов связей между информационными единицами. Отображение  $\Gamma$  задает между информационными единицами, входящими в  $I$ , связи из заданного набора типов связей. В зависимости от типов связей, используемых в модели, различают *классифицирующие* и *функциональные* сети, а также *сценарии* [4].

В **классифицирующих** сетях используются различные отношения структуризации. Такие сети позволяют в базу знаний вводить различные иерархические отношения между информационными единицами. **Функциональные** сети характеризуются наличием функциональных отношений. Их часто называют вычислительными моделями, так как они позволяют описывать процедуры вычислений одних информационных единиц через другие. В **сценариях** используются казуальные<sup>2</sup> отношения, а

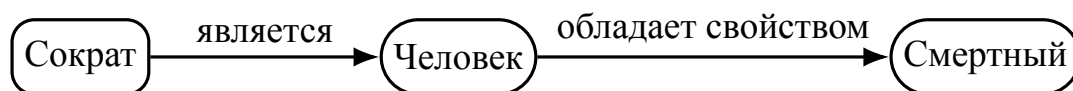
---

<sup>1</sup> ... Могут быть и животные, однако это нас не интересует, так как знания связывают только свойства  $H$  и  $M$ .

<sup>2</sup> Скорей всего, это слово образовано от англ. «case», что значит «случай». Здесь имеется в виду различение некоторых типовых ситуаций и осуществление некоторого действия в этой ситуации.

также отношения типов «средство — результат», «орудие — действие» и т. п.

Если в сетевой модели допускаются связи различного типа, то ее обычно называют *семантической сетью*. Примером семантической сети представления знаний является энциклопедический словарь. Такой словарь содержит определения одних понятий через другие. Как правило, в определениях выделяются те базовые определения, которые содержатся в этом словаре.



**Рис. 1.1.** Пример представления данных (Сократ) и знаний («является» и «обладает свойством») в виде семантической сети

На рис. 1.1 представлен пример 1.1 в виде семантической сети. В вершинах графа располагаются термины и объекты: «Человек», «Смертный», «Сократ». Дуги графа — это логические связи между терминами и объектами, например, «являться» разновидностью (в англоязычной литературе это отношение принято называть «is\_a»).

### 1.3.3. Продукционные модели

В моделях этого типа используются элементы логических и сетевых моделей. Из логических моделей заимствована **идея правил вывода**, которые здесь называются *продукциями*, а из сетевых моделей — описание знаний в виде формализма семантической сети [4].

В результате применения правил вывода к фрагментам сетевого описания происходит трансформация семантической сети за счет смены ее фрагментов, наращивания сети и исключения из нее ненужных фрагментов. Таким образом, в продукционных моделях процедурная информация явно выделена и описывается иными средствами, чем декларативная информация. Вместо логического вывода, характерного для логических моделей, в продукционных моделях появляется вывод на знаниях.

Рассмотрим продукционное представление примера 1.1. В виде сети представим второе высказывание — «Сократ — человек», а первое высказывание представим в виде правила преобразования участка сети в другой вид. Правила преобразования можно представлять в виде структур «Если ..., то ...», т. е., в нашем случае: «Если  $X$  — человек, то  $X$  — смертен». Полученное представление данных и знаний изображено на рис. 1.2.

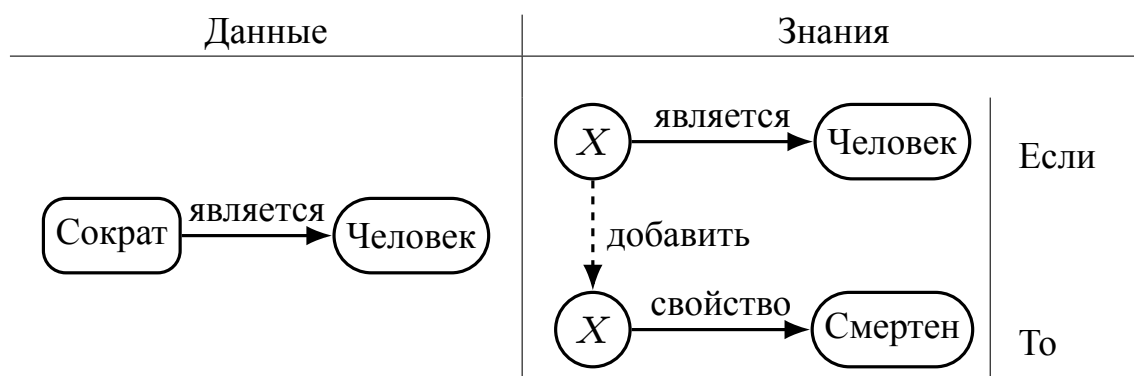


Рис. 1.2. Представление данных и знаний в продукционном виде

### 1.3.4. Фреймовые модели

В отличие от моделей других типов, во фреймовых моделях фиксируется жесткая структура информационных единиц, которая называется протофреймом<sup>1</sup>. В общем виде она выглядит следующим образом [4]:

(Имя фрейма:  
 Имя слота 1 (значение слота 1);  
 Имя слота 2 (значение слота 2);  
 . . . . .  
 Имя слота K (значение слота K)).

Значением слота может быть практически все, что угодно (числа или математические соотношения, тексты на естественном языке или программы, правила вывода или ссылки на другие

<sup>1</sup> Если брать аналогию с объектно-ориентированным программированием (ООП), то протофрейму соответствует описание класса объектов.



слоты данного фрейма или других фреймов). В качестве слота может выступать набор слотов более низкого уровня, что позволяет во фреймовых представлениях реализовать принцип матрешки. При конкретизации<sup>1</sup> фрейма ему и слотам присваиваются конкретные имена и происходит заполнение слотов. Таким образом, из протофреймов получаются фреймы-экземпляры. Переход от исходного протофрейма к фрейму-экземпляру может быть многошаговым, за счет постоянного уточнения значений слотов. Например, структура таблицы 1.2, записанная в виде протофрейма, имеет вид

(Список работников:

    Фамилия (значение слота 1);  
    Год рождения (значение слота 2);  
    Специальность (значение слота 3);  
    Стаж (значение слота 4)).

Таблица 1.2

Сведения о работниках

Фамилия	Год рождения	Специальность	Стаж (годы)
Попов	1965	слесарь	5
Сидоров	1946	токарь	20
Иванов	1925	токарь	30
Петров	1937	сантехник	25

Если в качестве значений слотов использовать данные таблицы 1.2, то получится фрейм-экземпляр

(Список работников:

    Фамилия (Попов-Сидоров-Иванов-Петров);  
    Год рождения (1965-1946-1925-1937);  
    Специальность (слесарь-токарь-токарь-сантехник);  
    Стаж (5-20-30-25)).

Связи между протофреймами задаются значениями специального слота Связь<sup>2</sup>. Часть специалистов ИИ считает, что нет

<sup>1</sup> Аналог конкретизации в ООП — это создание экземпляра класса (конкретного объекта).

<sup>2</sup> Аналог связи наследования в ООП.

необходимости специально выделять фреймовые модели в представлении знаний, так как в них объединены все основные особенности моделей остальных типов.

Считается, что для представления конкретной системы (набора) знаний всегда можно выбрать из каждого класса приведенных формализмов формальный метод, обладающий достаточной выразительностью языка представления этих знаний<sup>1</sup>. Таким образом, формализмы представления знаний эквивалентны, и, если удалось представить знание в одном из формализмов, то это знание можно представить и в остальных трех.

### **Вопросы для самопроверки**

1. Перечислите по крайней мере три свойства (признака) задач искусственного интеллекта.
2. Дайте характеристику терминам «искусственный интеллект», «данные» и «знания».
3. В чем суть теста А. Тьюринга?
4. Перечислите как минимум пять классов задач искусственного интеллекта.
5. Какие формализмы представления знаний существуют, чем они отличаются?

---

<sup>1</sup> Всякий формализм предоставляет несколько уровней (языков) выражения знаний, например, логические формулы можно выражать с помощью исчисления высказываний и исчисления предикатов первого порядка. Исчисление предикатов первого порядка выразительнее исчисления высказываний.



## 2. Язык программирования Пролог

Предметом нашего дальнейшего изучения будет язык логического программирования Пролог. Слово «Пролог» образовано из слияния терминов: «ПРОграммирование в терминах ЛОГики». Пролог относится к классу языков, называемых *сентенциальными*<sup>1</sup>. Классические реализации Пролога используются в вузах мира для обучения студентов методам автоматизации рассуждений. Специальные версии этого языка, например Prolog-III, являются дорогими коммерческими продуктами. Наибольшее распространение Пролог приобрел в Европе. В Америке активно развивался язык функционального программирования Лисп (от LISt Processing), основанный на  $\lambda$ -исчислении Черча. В г. Иркутске в Институте математики, экономики и информатики ИГУ под руководством профессора, доктора физ. мат. наук А. В. Манцивоты разработан язык Flang (Functional-logical LANGuage) — язык функционально-логического программирования, опосредующий свойства языков Пролог и Лисп.

### 2.1. Основные термины

Всякая программа на Прологе состоит из набора фраз (предложений) [2].

Прологовские предложения бывают трех типов: *факты*, *правила* и *запросы*. *Факты* содержат утверждения, которые считаются всегда безусловно верными. *Правила* содержат утверждения, истинность которых зависит от некоторых условий. *Запросы* — цель (условия), которую необходимо выполнить; она, как правило, описывает то, что в конечном счете требуется получить пользователю. С помощью *запросов* пользователь может спрашивать систему о том, какие утверждения являются истинными.

Предложения Пролога состоят из *головы* и *тела*. Тело — это список *целей*, разделенных запятыми. Запятая понимается как конъюнкция.

---

<sup>1</sup> От англ. слова *sentence* (фраза, предложение).

Факты — это предложения, имеющие **пустое** тело. Запросы — только тела правил. Правила имеют голову и (непустое) тело.

```
human(socrates).           % факт языка Пролог
mortal(X) :- human(X).     % правило языка Пролог
```

Здесь `mortal(X)` — *голова* правила, а `human(X)` — *тело* правила. В представленной в виде Пролог-программы системе данных и знаний можно выполнить запрос

```
mortal(socrates).         % Сократ смертен?
```

Каждое предложение заканчивается точкой.

### 2.1.1. Реализации языка

Большинство классических реализаций Пролога — либо интерпретаторы, либо трансляторы в коды виртуальной машины, например *машины Уоррена* (Warren). В данный момент существует множество реализаций Пролога, которые поддерживают компиляцию в машинный код. Из известных реализаций Пролога в интернете доступны GNU-Prolog, Visual Prolog (бывший Turbo Prolog) — компиляторы, SWI-Prolog функционирует на основе виртуальной машины ZIP. Большинство реализаций опираются на стандарт ISO-Prolog, но некоторые, например Visual Prolog, — нет. В данном пособии под ISO-Prolog будем понимать любую реализацию языка Пролог, поддерживающую этот стандарт.

В этом учебном пособии использованы синтаксис и семантика стандарта ISO-Prolog, а реализация — GNU-Prolog<sup>1</sup> [12].

## 2.2. Структура языка

### 2.2.1. Простые типы данных

**Термы.** Объекты данных в Прологе обозначаются *термами*. *Терм* — это структура, обозначающая некоторый объект [8]. Терм может быть константой, переменной или составным термом. Со-

---

<sup>1</sup> В операционной системе Windows удобнее использовать графическую оболочку системы SWI-Prolog со своим интерпретатором [13].

ставной терм соответствует структурам данных в структурном программировании. Константами являются целые и действительные числа, например:

0, -1, 123.4, 0.23E-5,

т. е. будем считать, что числа сами себя обозначают.

Большинство реализаций Пролога поддерживают и целые, и действительные числа. Для того чтобы выяснить, каковы диапазоны и точность чисел, следует обратиться к руководству по конкретной реализации.

К термам относятся также *символы*, такие как:

goldy, a, atom.

*Символ (symbol)* есть любая последовательность символов английского алфавита и чисел, начинающаяся с маленькой английской буквы, либо любая строка, взятая в одинарные кавычки, например 'Российская Федерация'.

Как и в других языках программирования, термы обозначают конкретные элементарные объекты моделируемой предметной области, а все другие типы данных в Прологе составлены из сочетаний других термов: чисел и символов.

**Переменные.** Имена переменных начинаются с заглавных букв английского алфавита или с символа подчеркивания «\_» и содержат только символы букв, цифр и подчеркивания. Ниже приведены примеры переменных:

X, Variable, \_3, \_variable.

Понятие *переменной* в Прологе отличается от принятого во многих языках программирования. Переменная не рассматривается как выделенный участок памяти. Она служит для обозначения объекта, на который нельзя сослаться по имени. Переменную можно считать **локальным<sup>1</sup> именем** для некоторого объекта. Переменная также является термом.

---

<sup>1</sup> Внутри утверждения.

Если переменная используется только один раз, необязательно называть ее. Она может быть записана как анонимная переменная, состоящая из одного символа подчеркивания «\_». Кроме этого, анонимная переменная обозначает объекты, свойства которых нас не интересуют. Эта переменная может обозначать высказывания «что-либо», «значение, которое нас не интересует», «все, что угодно». Переменные являются элементарными объектами языка Пролог.

**Область действия переменных.** Областью действия переменной является *утверждение* (факт, правило или запрос). В пределах одного утверждения одно и то же имя принадлежит одной и той же переменной. Два утверждения могут использовать одно имя переменной совершенно различным образом. Правило определения области действия переменной справедливо также в случае рекурсии и в том случае, когда несколько утверждений имеют одну и ту же голову.

Единственным исключением из правила определения области действия переменных является анонимная переменная, например \_ в факте `likes(X, _)`. Каждая анонимная переменная есть **отдельная сущность**. Она применяется тогда, когда конкретное значение переменной несущественно для данного утверждения. Таким образом, каждая анонимная переменная отличается от всех других анонимных переменных в утверждении.

Переменные, отличные от анонимных, называются *именованными*, а неконкретизированные (переменные, которым не было присвоено значение) называются *свободными*.

**Сложные термы.** Завершает список синтаксических единиц сложный терм, или структура. Все, что не может быть отнесено к переменной, числу и символу, называется сложным термом. Сложный терм состоит из чисел, символов, переменных и других сложных термов, связанных так называемыми функторами. Пример сложного терма — двоичное дерево:

```
t(t(t(nil, t(nil, nil)), nil), t(t(nil, nil),  
t(t(nil, nil), t(nil, nil)))).
```

## 2.3. Программа на языке Пролог

Программирование на языке Пролог отличается своими особенностями [2]:

- Декларативная интерпретация (смысл) программ главенствует над процедурным. При этом программист в качестве программы выписывает **то, что он хочет получить**, а не **то, как это надо получать** (вычислять, искать и т. д.).
- Программист осуществляет задание *отношений*, а не процедур и функций, что является более общим формальным аппаратом представления алгоритмов.

**Утверждения.** Программа на языке Пролог есть совокупность *утверждений*. Утверждения включают цели и хранятся в базе знаний Пролога. Таким образом, база знаний Пролога может рассматриваться как программа на Прологе. В конце утверждения ставится точка «.». Утверждения также называются предложениями или фразами.

Основная операция Пролога — доказательство целей, входящих в утверждение.

Существуют два типа утверждений:

**Факт** — это утверждение, которое безусловно истинно.

**Правило** состоит из головы и одной или более хвостовых подцелей, которые истинны при некоторых условиях.

Правило обычно имеет несколько хвостовых подцелей в форме конъюнкции. Конъюнкцию можно рассматривать как логическую функцию «&». Таким образом, правило истинно, если истинны все его хвостовые подцели.

Примеры фактов:

```
dog(rex).           % rex - собака.  
parent(goldy, rex). % rex - родитель goldy.
```

Примеры правил:

```
dog(X) :- parent (X, Y), dog(Y).  
    % Всякий, у которого родитель - собака,  
    %   тоже является собакой.  
human(X) :- man(X).  
human(X) :- woman(X).  
    % Человек - это мужчина или женщина.
```

Символом «:-» обозначена логическая связка « $\leftarrow$ » (если), а символом «, $\rightarrow$ » — логическая операция конъюнкция «&». Прибегая к помощи исчисления предикатов первого порядка, первое правило можно переписать так:

$$\forall x \forall y (parent(x, y) \& dog(y) \rightarrow dog(x)).$$

Напоминаем, что правила состоят из *заголовка* и *тела*. Например, `dog(X)` и `human(X)` — заголовки правил, а `man(X)` и `woman(X)` — части тел правил (подцели).

Введем еще один термин — *предикат*<sup>1</sup>. Суть этого слова следующая: несколько объектов связаны некоторым свойством. Например, `parent(X, Y)` связывает некоторый объект  $X$  с некоторым объектом  $Y$  свойством «быть родителем». Факты и заголовки правил и назовем предикатами. Предикаты различаются между собой названием (идентификатором) и количеством аргументов, т. е. предикат `parent(X, Y)` обозначается выражением `parent/2`.

Разница между правилами и фактами чисто семантическая. Так, факты, приведенные выше, записываются в виде правил следующим образом:

```
dog(rex) :- true.                % dog(rex).  
parent(goldy, rex) :- true.    % parent(goldy, rex).
```

Теперь рассмотрим нашу первую программу на языке ISO-Prolog.

---

<sup>1</sup> В англ. языке — Predicate, происходит от соответствующего латинского слова, обозначающего слово «свойство».

```
dog(rex).                % rex - собака.  
dog(X) :- parent(X, Y), dog(Y).  
% Всякий, у которого родитель - собака,  
% тоже является собакой.  
  
parent(goldy, rex). % rex - родитель goldy.  
  
human(X) :- man(X).  
human(X) :- woman(X).  
% Человек - это мужчина или женщина.
```

Предикаты с одинаковым именем в большинстве версий Пролога должны быть сгруппированы, т. е. не разрешается, например, вставлять между фактом `dog(rex)` и правилом `dog(X) :- parent(X, Y), dog(Y)` другие предикаты (факты и правила), например, `human/2`. Имена предикатов пишутся с маленькой латинской буквы, некоторые версии поддерживают русские имена идентификаторов (в кодировке UTF-8). Некоторые версии ISO-Prolog, например GNU-Prolog, неадекватно реагируют на пробел между именем предиката и открывающей скобкой. Встроенный предикат `true/0` обозначает тождественно-истинное высказывание, он не связывает никакие объекты.

В синтаксисе языка Пролог помимо связки «,  
» («и»), есть еще связка «;  
» («или»), с помощью которой последнее правило `human/1` можно переписать в следующем виде:

```
human(X) :- man(X) ; woman(X).  
% Человек - это мужчина или женщина.
```

**Типы данных языка Пролог.** Простые типы данных в языке Пролог представлены следующими идентификаторами:

**Symbol** — тип, обозначающий «символы», т. е. имена (названия, обозначения) объектов (понятий). Например, нашего Сократа можно обозначить символом `socrates`<sup>1</sup> или в виде строки в одинарных кавычках `'Сократ'`. Символы — это

---

<sup>1</sup> Мы помним, что символы начинаются с маленькой английской буквы.



специальные структуры, похожие на строки, которым сопоставляется некое индивидуальное число из таблицы символов Пролога. В процессе решения задачи Пролог оперирует не строками, а этими числами. В стандартной библиотеке ISO-Prolog есть различные средства для манипуляции символами как строками (сложение, поиск подстроки и т. п.), при этом для каждого нового символа создается свое число в таблице символов.

**Char** — тип, обозначающий индивидуальные символы ASCII, например 'A'. Этот тип ничем не отличается от Symbol.

**String** — последовательность символов (русских, английских и др.), заключенная в двойные кавычки (как в языке C, C++) «"»». В ISO-Prolog строками являются списки символов-букв. К строкам применимы все операции со списками (см. ниже). В данном учебном пособии мы не будем использовать строки.

**Integer** — типы данных, обозначающие целые величины.

**Float** — тип данных, обозначающие рациональные величины в виде чисел с плавающей точкой.

В стандарте ISO-Prolog данные обозначения типов используются только для документирования прикладного пользовательского интерфейса библиотек<sup>1</sup>. В системе Visual-Prolog интерфейсы предикатов задаются в специальных секциях в явном виде.

**Сложные типы данных.** Сложные типы данных (структуры) в Прологе создаются с помощью так называемых *функторов*. Функтор — это терм, состоящий из простых типов данных и других функторов. Например, конкретное двоичное дерево, состоящее из четырех вершин, задается так:

```
t(t(nil,5,nil), 10, t(t(nil,15,nil),20, nil)).
```

---

<sup>1</sup> Англ. «Application Programming Interface», API.



Здесь `nil`<sup>1</sup> — символ, использованный нами для обозначения пустого поддерева, `t/3` — трехаргументный функтор (точнее, функциональный символ), объединяющий три более простых термина в одну структуру (терм) `t(<левое поддерево>, <данные>, <правое поддерево>)`. Вторым аргументом в нашем примере `t/3` — число, которое хранится в дереве.

**Пример 2.1.** *Поиск в двоичном дереве, используя полный перебор его вершин.*

```
% search(tree, integer).
% истинный, если второй аргумент
% находится в дереве (первый аргумент).
search( t( _, X, _), X).
% В вершине найдено требуемое число.
search( t( L, Y, _), X) :- Y <> X,
    % Если требуемого числа здесь нет
    search( L, X).
    % то попробовать найти его в левом
    % поддереве.
search( t( _, Y, R), X) :- Y <> X,
    % Если требуемого числа здесь нет
    search( R, X).
    % то попробовать найти его в правом
    % поддереве.
```

Эта программа может выполнить, например, такой запрос:

```
search( t( t(nil, 5, nil), 0,
          t(nil, 8, t(nil, 10, nil))), 10).
```

**Запросы.** После записи утверждений в базу знаний вычисления (логический вывод целевого утверждения, цели) инициируется вводом *запроса*.

Запрос выглядит так же, как хвост правила, образуется и обрабатывается по тем же правилам, но его ввод производится после

---

<sup>1</sup> Точнее, `nil/0`.

загрузки файла с программой в среду выполнения. Запрос состоит из ряда хвостовых целевых утверждений, записываемых, чаще всего, в виде конъюнкции. Приведем примеры запросов:

```
parent(X, Y), dog(Y).
```

или

```
dog(goldy).
```

В большинстве Пролог-систем запросы вводятся в специальной командной строке рядом с приглашением «?-», вместо вертикальной черты может быть число — номер запроса. Например, представленные выше запросы вводятся в систему GNU-Prolog следующим образом:

```
?- parent(X, Y), dog(Y).
```

или

```
?- dog(goldy).
```

Далее мы будем обозначать выполняемые вручную запросы при помощи именно такой записи, если не оговорено особо.

```
?- dog(goldy).
```

Если необходимо инициировать запрос при загрузке базы знаний (программы), то согласно стандарту ISO надо использовать предикат `initialization/1`.

```
:- initialization(dog(goldy)).
```

Этот запрос выполнится во время загрузки программы. Аналогично выполняются запросы на выполнение различных директив. Для этого используется следующая декларация:

```
:- dynamic(fib/2).
```

Конкретно в этом случае происходит запуск директивы транслятора, которая сообщает системе, что предикат `fib/2` будет динамическим, т. е. новые экземпляры `fib/2` можно будет добавлять в рабочую память Пролога. Запросы, включающие предикаты пользователя как подцели, в некоторых системах также можно запускать при помощи этой конструкции<sup>1</sup>.

---

<sup>1</sup> Наши эксперименты с GNU-Prolog показали, что единственный способ задания запроса над предикатом пользователя — это использование предиката `initialization/1`.

**Загрузка программы.** Редактирование программ Пролога возможно в Windows-версии системы GNU-Prolog во встроенном редакторе. Разумеется, программист может использовать свой любимый текстовый редактор, а затем просто загрузить программу в интерпретатор Пролога.

В классических реализациях языка Пролог введение списка утверждений в Пролог-систему осуществляется при помощи встроенного предиката `consult/1`. Единственным аргументом этого предиката `consult/1` является атом, который интерпретируется системой как имя файла, содержащего текст программы на Прологе. Файл открывается, и его содержимое записывается<sup>1</sup> в базу знаний. Если в файле встречаются управляющие команды, они сразу же выполняются. Возможен случай, когда файл не содержит ничего, кроме управляющих команд, например, для загрузки других файлов. Для ускорения набора команды загрузки пользователи Пролога изобрели для себя следующую конструкцию, являющуюся синонимом предикату `consult/1`:

```
?- [ '<имя файла>' ].
```

В Прологе имеются и другие методы добавления и удаления утверждений из базы данных. Некоторые реализации языка поддерживают модульную структуру, позволяющую разрабатывать модульные программы.

В заключение раздела дадим формальное определение основной части синтаксиса Пролога, используя форму записи Бэкуса — Наура, называемую бэкусовской нормальной формой.

```
<запрос> ::= <хвост утверждения>
<правило> ::= <голова утверждения> ":-"
    <хвост утверждения>
<факт> ::= <предикат> "."
<голова утверждения> ::= <предикат>
<предикат> ::= <идентификатор> [ "(" <термы> ")" ]
```

<sup>1</sup> Фактически производится добавление новых знаний и фактов из загружаемого файла. Если в файле содержатся определения предикатов, которые уже имеются в рабочей памяти системы, то происходит обновление этих предикатов. Это может стать причиной неработоспособности программы. Рекомендуется иногда выходить из интерпретатора и очищать рабочую память.

```
<предикаты> ::= <предикат> [ ", " <предикаты> ]  
<хвост утверждения> ::= <предикаты> [ "; "  
    <предикаты> ] ". "  
<термы> ::= <терм> [ ", " <термы> ]  
<терм> ::= <число> | <переменная> | <строка> |  
    <символ> | <функтор>  
<функтор> ::= <идентификатор> [ "(" <термы> ")" ]
```

Знак « $::=$ » здесь читается как «состоит из». Данное определение синтаксиса не включает операторную, списковую и строковую формы записи классического Пролога. Однако любая программа на Прологе может быть написана с использованием вышеприведенного синтаксиса. Специальные формы только упрощают понимание программы. Как мы видим, синтаксис Пролога не требует пространного объяснения. Но для проектирования и реализации хороших программ необходимо глубокое понимание языка.

## 2.4. Унификация и мэтчинг

Одним из наиболее важных аспектов программирования на Прологе являются понятия унификации, мэтчинга и конкретизации переменных.

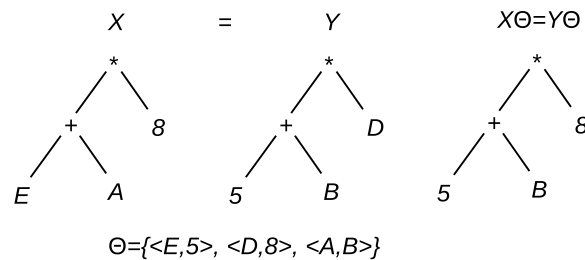
Пролог пытается отождествить термы при доказательстве (согласовании) целевого утверждения. Например, для согласования запроса  $\text{dog}(X)$  цель  $\text{dog}(X)$  будет унифицирована с фактом  $\text{dog}(\text{rex})$ , в результате чего переменная  $X$  станет конкретизированной:  $X = \text{rex}$ .

Унификация<sup>1</sup> — это процесс, на вход которого поступают два терма  $t_1$  и  $t_2$ . Процесс унификации анализирует структуры этих термов и ищет их схожие элементы. Конечная цель унификации — найти множество подстановок структур (термов) в переменные, найденные в  $t_1$  и  $t_2$ . Подстановки должны быть такие, что, заменяя все переменные в  $t_1$  и  $t_2$  на структуры из этой подстановки,  $t_1$  и  $t_2$  превращаются в одинаковые структуры. Если

---

<sup>1</sup> Англ. unification, русскоязычные неточные смысловые эквиваленты: «сравнение», «согласование», «сверка».

удается построить такое множество, то считается, что унификация прошла успешно. Собственно, в результате унификации прологовские переменные и получают свои значения. Пример унификации арифметических выражений показан на рис. 2.1. На этом рисунке  $\Theta$  — множество подстановок (первый аргумент — переменная, в который подставляется терм — второй аргумент). Выражение  $X\Theta$  — результат применения (application) подстановки  $\Theta$  в выражение  $X$ .



**Рис. 2.1.** Унификация двух арифметических выражений

Рассмотрим правила унификации более подробно. Терм  $X$  унифицируется с термом  $Y$  по следующим правилам. Если  $X$  и  $Y$  — символы или числа, то они унифицируемы, только если они одинаковы. Если  $X$  является константой или структурой, а  $Y$  — неконкретизированной переменной, то  $X$  и  $Y$  унифицируемы и  $Y$  принимает значение  $X$  (и наоборот). Если  $X$  и  $Y$  — структуры, то они унифицируемы тогда и только тогда, когда у них один и тот же главный функтор и арность и каждая из их соответствующих компонент сопоставима. Если  $X$  и  $Y$  — неконкретизированные (свободные) переменные, то они сопоставимы, в этом случае говорят, что они сцеплены. В таблице 2.1 приведены примеры унифицируемых и неунифицируемых термов.

Следует сказать, что в большинстве реализаций Пролога для повышения эффективности его работы допускается существование циклических унификаторов<sup>1</sup>. Например, попытка унифицировать термы  $f(X)$  и  $X$  приведет к циклической подстановке  $X = f(X)$ , который определяет бесконечно-вложенный терм

<sup>1</sup> Унификация без проверки вхождения терма имеет сложность  $O(n)$ , тогда как строгая —  $O(n^2)$ .

$f(f(f(\dots)))$ , что еще и логически некорректно. Иногда такую неполную унификацию называют мэтчинг (matching).

Таблица 2.1

Иллюстрация унификации

Терм <sub>1</sub>	Терм <sub>2</sub>	Унифицируемы?
jack(X)	jack(human)	да: X=human
jack(person)	jack(human)	нет
jack(X, X)	jack(23, 23)	да: X=23
jack(X, X)	jack(12, 23)	нет
jack(_, _)	jack(12, 23)	да
f(Y, Z)	X	да: X = f(Y, Z)
X	Z	да: X = Z

Возможность унификации (мэтчинга)) двух термов проверяется с помощью оператора «=».

Ответом на запрос

?- 3 + 2 = 5

в ISO-Prolog будет нет, так как термы неунифицируемы (в ISO-Prolog оператор «=» не вычисляет значения своих аргументов). ISO-Prolog воспринимает арифметические операции как функторы, а арифметические выражения — как сложные структуры; для вычисления арифметических выражений используется специальный предикат «is».

Унификация часто используется для доступа к подкомпонентам термов. Так, например, если нам требуется выбрать правое поддерево (см. пример 2.1), можно провести следующую унификацию (T — исходное дерево, ST — поддерево).

```
T=t( t( nil, 7, nil), 5, t( t(nil, 10, nil), 8,
    nil)),
( _, _, ST) = T
% теперь ST = t( t(nil, 10, nil), 8, nil).
```

Отрицание оператора «=» записывается как «\=» или «\+ (\_ = \_)».

## 2.5. Лабораторная работа 1: Факты и правила

Формализация высказываний естественного языка в виде Пролог-программы.

**Задание.** В работе<sup>1</sup> требуется формализовать высказывания в виде программы на языке Пролог. В программе требуется выполнить ряд запросов, объяснить выдаваемые Пролог-системой результаты.

**Цель работы.** Приобрести навыки формализации высказываний на естественном языке в виде фактов, правил и запросов языка Пролог.

**Время работы.** На выполнение работы отводится два академических часа.

### Индивидуальные задания

1. Флэш — собака. Ровер — собака. Бутси — кошка. Стар — лошадь. Флэш черная. Бутси коричневая. Ревер рыжая. Стар белая. Домашнее животное — собака или кошка. Животное — домашнее животное или лошадь. У Тома есть собака не черного цвета. У Кейта есть лошадь или что-то черного цвета.

**Запросы:**

- Ровер рыжая?
- Определить клички всех собак.
- Определить владельцев чего-либо.
- Определить владельцев животных небелого цвета.

---

<sup>1</sup> Задание на лабораторную работу 1 разработано преподавателем кафедры вычислительной техники Института кибернетики НИ ИрГТУ, доцентом, канд. техн. наук С. С. Сосинской.

2. Бутси — коричневая кошка. Корни — черная кошка. Мактэвити — рыжая кошка. Флэш, Ровер и Спот — собаки; Ровер — рыжая, а Спот — белая. Все животные, которыми владеют Том и Кейт, имеют родословные. Том владеет всеми черными и коричневыми животными. Кейт владеет всеми собаками небелого цвета, которые не являются собственностью Тома.

Алан владеет Мактэвити, если Кейт не владеет Бутси и если Спот не имеет родословной. Флэш — пятнистая собака.

**Запросы:**

- Какие животные не имеют хозяев?
- Найдите всех собак и укажите их цвет.
- Укажите всех животных, которыми владеет Том.
- Перечислите всех собак Кейта.

3. Определить следующие отношения: СЫН, ДОЧЬ, ОТЕЦ, МАТЬ, МУЖЧИНА и ЖЕНЩИНА. Описать факты для некоторых из них.

**Запросы:**

- Определить всех сыновей определенной матери.
- Определить всех детей определенной пары родителей.
- Определить родителей определенного человека.
- Является ли определенный человек женщиной?

4. Мэри любит персики. Мэри любит кукурузу. Мэри любит яблоки. Бет любит то, что любит Мэри, если это — фрукт и если он красный. Бет любит то, что любит Мэри, если это кукуруза. Персики — фрукт. Яблоки — фрукт. Цвет персиков желтый. Цвет апельсинов оранжевый. Цвет яблок красный. Цвет яблок желтый.

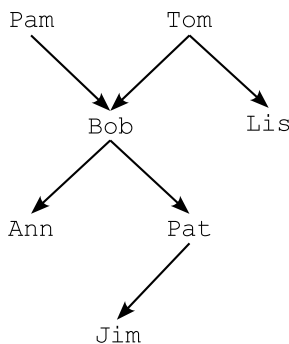
**Запросы:**

- Что любит Бет?
- Любит ли Мэри кукурузу?



- Какие фрукты известны?
- Какого цвета фрукты, которые любят Бет и Мэри?

5. Задано дерево родственных связей.



Кроме того, определить отношения ПОЛ, РЕБЕНОК, РОДИТЕЛЬ\_РОДИТЕЛЯ, ПРЕДОК и МАТЬ.

**Запросы:**

- Кто родитель Pat?
- Есть ли у Lis ребенок?
- Кто потомки Pat?
- Является ли Pam матерью Bob?

6. Медведь большой. Слон большой. Кот маленький. Медведь коричневый. Кот черный. Слон серый.

Любой черный или коричневый предмет является темным.

**Запросы:**

- Кто одновременно большой и темный?
- Есть ли коричневые маленькие слоны?
- Есть ли большие и темные медведи?
- Есть ли черный кот?

7. Мэри, Сьюзи и Джейн работают в дневную смену. Сэм, Джейн, Боб и Патриция работают в вечернюю смену. Знают друг друга те, кто работает в одну смену.

**Запросы:**

- Знают ли друг друга Мэри и Джейн?
- Кто работает в дневную смену?
- Есть ли кто-то, кто работает в обе смены?
- Есть ли кто-то, кто не знает друг друга?

8. Можно совершить путешествия, перечисленные в таблице 2.2. Любые две транспортные компании являются конкурентами, если они обслуживают один и тот же маршрут. Можно путешествовать из одного города в другой, если можно добраться из одного города в другой через промежуточный город.

**Запросы:**

- Являются ли Амтрак и Пиплз конкурентами?
- Какие компании дают возможность путешествовать из Нью-Йорка в Вашингтон?
- Можно ли путешествовать из Бирлингтона в Портленд?
- Определить всех конкурентов.

**Таблица 2.2**

Расписание рейсов

Компания	Из	В	Вид транспорта
Амтрак	Нью-Йорк	Бостон	Ж/д
Транзит	Нью-Йорк	Принстон	Ж/д
Амтрак	Бостон	Портленд	Ж/д
Грейхаунд	Бостон	Портленд	Автобус
Амтрак	Нью-Йорк	Вашингтон	Ж/д
Пиплз	Нью-Йорк	Вашингтон	Самолет
Пиплз	Бирлингтон	Нью-Йорк	Самолет

9. Определить факты о принадлежности студента определенной студенческой группе. Считается, что два студента знают друг друга, если они учатся в одной группе.

**Запросы:**

- Кого знает определенный студент?
- Определить состав определенной группы.
- В каких группах учатся люди с определенным именем?
- Знает ли один студент другого?

10. Заданы факты о маршрутах движения автобусов между двумя разными городами, в которых указаны: номер маршрута, названия двух городов, день и время отправления и прибытия. Известны также фамилии водителей, работающих на определенных маршрутах. Можно попасть из одного города в другой, если существуют автобусные маршруты из первого города во второй или из первого города в промежуточный, и из промежуточного во второй.

**Запросы:**

- Можно ли проехать из одного города в другой?
  - Указать автобусы, выходящие из определенного города в определенный день, и время отправления.
  - Перечислить фамилии водителей определенного маршрута.
  - Указать дни и часы отправления определенного маршрута.
11. Есть факты об отцах некоторых людей и о братьях некоторых людей. Определить отношение ДЯДЯ.

**Запросы:**

- Определить братьев конкретного человека.
  - Кто является отцом конкретного лица?
  - Связаны ли два человека отношением ОТЕЦ?
  - Определить, является ли один человек дядей другого.
12. Определить отношения РОДИТЕЛЬ, ЖЕНЩИНА как набор фактов, правило РАЗЛИЧНЫ, СЕСТРА (определяемое через РОДИТЕЛЬ, ЖЕНЩИНА и РАЗЛИЧНЫ) и ТЕТЯ (определяемое через РОДИТЕЛЬ и СЕСТРА).

**Запросы:**

- Кто является родителями определенного человека?
- Определить всех детей определенных родителей.
- Определить, есть ли сестры у определенного человека.
- Определить, есть ли тетя у определенного человека.

## Методические указания к лабораторной работе

Процесс построения некоторого формального представления высказываний естественного языка называется *формализацией*. Что это такое? Ответ на этот вопрос столь сложен, сколь сложен ответ на вопрос: Что такое модель? В научных кругах под формализацией понимается словосочетание «дружеский шарж», т. е. формальное представление некоторого естественного объекта (например, высказывания) — это дружеский шарж.

Продemonстрируем на примерах, почему формализация — это именно **шарж**. Пусть дано высказывание: «Лена любит кататься на велосипеде и на горных лыжах». Какая логическая связка будет соответствовать союзу «и»?... На самом деле это будет связка « $\vee$ », потому, что с формально-логической точки зрения высказывание обозначает: «Лена катается на велосипеде **или** горных лыжах». Второй пример: «Я пойду домой, а моя жена на работу». Здесь союз «а» по смыслу соответствует логической связке « $\&$ ». Таким образом, формализация естественного текста не может быть сделана «в лоб», необходимо понять, что было сказано.

При выполнении лабораторной работы следует придерживаться следующих общих правил:

1. Прочитать весь текст высказывания и определить, что будет **объектами**, а что **свойствами**, связывающими эти объекты. Например, пусть даны следующие высказывания: «Аня любит Колю. Коля любит Лену. А Лена смотрит в светлое будущее.» Тогда, объектами будут: Аня, Коля, Лена и «светлое будущее», а свойствами — отношения «любит» и «смотреть в», которые связывают два объекта («Кто» «любит» «Что», «Кто» «смотрит в» «Что»).
2. Свойства объектов могут быть заданы перечислением либо через другие известные свойства. В нашем примере свойство «любит» задается перечислением:

```
be_in_love(ann, niko).  
be_in_love(niko, helen).
```

Но высказывание, вроде «любовного треугольника», можно задать через `be_in_love/2`:

```
love_triangle(X, Y, Z) :- % любовный треугольник
    be_in_love(X, Y),      % первого рода, когда
    be_in_love(Z, Y).      % двое любят одного.
love_triangle(X, Y, Z) :- % любовный треугольник
    be_in_love(X, Y),      % второго рода – без-
    be_in_love(Y, Z).      % ответная любовь.
```

Признаком хорошей формализации (дружественности шаржа) является, как и везде в программировании, хорошая гибкость и интерпретируемость программы: более сложные отношения формулируются через более простые; свойства в достаточной мере абстрактны.

### Вопросы для самопроверки

1. Какие структурные единицы формируют программу на языке Пролог?
2. Перечислите простые структуры данных Пролога.
3. Что такое «терм», в чем отличие переменной от символа?
4. Приведите пример унификации двух структур, представляющих логические выражения.
5. Какова будет унифицирующая подстановка  $\Theta$  двух следующих термов:  $X = \text{fib}(Y+1)$  и  $Y = \text{fib}(C+5+D)$ <sup>1</sup>?

---

<sup>1</sup>  $Y = \text{fib}((C+5)+D)$ .

### 3. Списки и их обработка

Кроме описанных в разделе 2.3 структур данных, создаваемых с помощью функторов, в Прологе существует еще одна структура данных — *список* [2].

**Списковая форма записи.** Задачи, связанные с обработкой списков, на практике встречаются очень часто. Скажем, нам понадобилось составить список студентов, находящихся в аудитории. С помощью Пролога мы можем определить список как последовательность термов, заключенных в скобки. Приведем примеры правильно построенных списков Пролога:

```
[jack, john, fred, jill, john]
[name(john, smith), age(jack, 24), X]
[X, Y, date(12, january, 1986), X]
[]
```

В Turbo Prolog структура списков должна быть определена в явном виде в секции Domains. Например, список, состоящий из строковых значений, определяется в Turbo Prolog так:

```
Domains
    str_list = string *
```

Символ «\*» определяет свойство нового типа `str_list` «быть списком» строк.

Запись `[H | T]` определяет список, полученный добавлением элемента (терма) `H` в начало списка `T`. Говорят, что `H` — голова, а `T` — хвост списка `[H | T]`. На запрос

```
?- L = [a | [b, c, d]].
```

будет получен ответ

```
L = [a, b, c, d],
```

а на запрос

```
?- L = [a, b, c, d], L2 = [2 | L].
```

---

получим ответ

$$L = [a, b, c, d], L2 = [2, a, b, c, d].$$

Запись  $[H \mid T]$  используется для того, чтобы определить голову и хвост списка. Так, запрос

$$[X \mid Y] = [a, b, c].$$

дает ответ

$$X = a, Y = [b, c].$$

Заметим, что употребление только имен переменных  $H$  и  $T$  необязательно. Кроме записи вида  $[H \mid T]$ , для выборки термов используются переменные. Запрос

$$?- [a, X, Y] = [a, b, c].$$

определит значения

$$X=b, Y=c,$$

а запрос

$$?- [person(X) \mid T] = [person(john), a, b].$$

значения

$$X=john, T=[a, b].$$

Можно отделять в голове списка несколько элементов, соответствующая запись выглядит так:  $L = [X1, X2, X3 \mid T]$ .

**Некоторые стандартные отношения для обработки списков.** Покажем на примерах использование записи вида  $[H \mid T]$  вместе с рекурсией для определения некоторых полезных отношений над списками.

*Принадлежность элемента списку.* Сформулируем задачу проверки принадлежности данного терма списку<sup>1</sup>.

---

<sup>1</sup> Используется методика рассуждения, представленная в книге [2]. Сначала рассматриваются самые простые варианты входных данных, затем сложные.



Граничное условие (база индукции): Терм  $R$  содержится в списке  $[H \mid T]$ , если  $R = H$ .

Рекурсивное условие (индуктивный шаг): Терм  $R$  содержится в списке  $[H \mid T]$ , если  $R$  содержится в  $T$ .

Первый вариант записи определения на Прологе имеет вид:

```
in(R, L) :-  
    L=[H | T], H=R.  
in(R, L) :-  
    L=[H | T], in(R, T).
```

Цель  $L = [H \mid T]$  в теле обоих утверждений служит для того, чтобы разделить список  $L$  на голову и хвост.

Можно улучшить программу, если учесть тот факт, что Пролог сначала унифицирует с целью голову утверждения, а затем пытается унифицировать его тело. Новая процедура `in` определяется таким образом:

```
in(R, [R | T]).  
in(R, [H | T]) :- in(R, T).
```

На запрос

```
?- in(a, [a, b, c]).
```

будет получен ответ `yes`. На запрос

```
?- in(b, [a, b, c]).
```

тоже ответ `yes`.

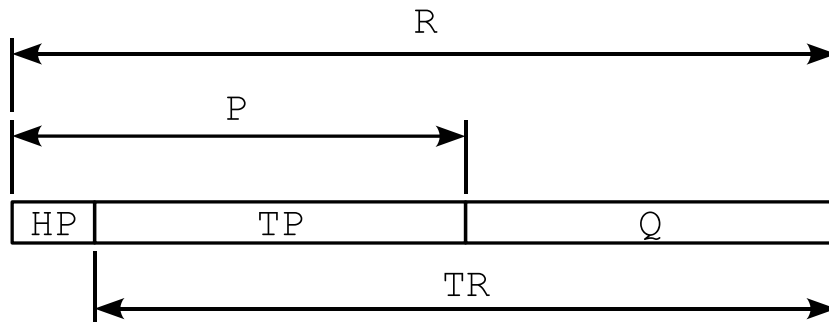
Существуют реализации Пролога, где предикат принадлежит (`in`) является встроенным, например, в ISO-Prolog этот предикат называется `member/2`.

*Соединение двух списков.* Задача присоединения списка  $Q$  к списку  $P$ , в результате чего получается список  $R$ , формулируется следующим образом:

Граничное условие: Присоединение к  $[]$  списка  $Q$  дает  $Q$ . Рекурсивное условие: Присоединение к концу списка  $P$  списка  $Q$  выполняется так:  $Q$  присоединяется к хвосту  $P$ , а затем спереди добавляется голова  $P$ .

Определение (см. рис. 3.1) можно непосредственно записать на Прологе:

```
conc([], Q, Q).
conc(P, Q, R) :- P=[HP | TP],
    conc(TP, Q, TR), R=[HP | TR].
```



**Рис. 3.1.** Конкатенация списков  $[HP \mid TP]$ ,  $Q$ ,  $[HP \mid TR]$

Однако, как и в предыдущем примере, воспользуемся тем, что Пролог унифицирует с целью голову утверждения, прежде чем пытаться согласовать тело:

```
conc([], Q, Q).
conc([HP | TP], Q, [HP | TR]) :-
    conc(TP, Q, TR).
```

На запрос

```
?- conc([a, b, c], [d, e], L).
```

будет получен ответ

```
L = [a, b, c, d],
```

но на запрос

```
?- conc([a, b], [c, d], [e, f]).
```

ответом будет no.

Часто процедура «присоединить» используется для получения списков, находящихся слева и справа от данного элемента:

```
?- conc(L, [jim | R], [jack, bill, jim,
    tim, jim, bob]).
    L = [jack, bill], R = [tim, jim, bob];
    % далее идет еще одно решение
    L=[jack, bill, jim, tim], R=[bob].
```

В командной строке GNU-Prolog при выполнении запроса и при наличии множества решений интерпретатор, выдав очередное решение, приостанавливает процесс решения задачи и ожидает реакцию пользователя на незримый вопрос «Что делать дальше — остановить процесс поиска решения, вывести новое решение, вывести все решения?» Если вам достаточно одного решения, то, нажимая клавишу «Enter», вы вернетесь в командную строку запроса. Если вас интересует еще одно решение, то при нажатии клавиши «;» Пролог попытается найти это решение. GNU-Prolog позволяет вывести все решения, для этого предназначена клавиша «a»<sup>1</sup> (all). Некоторые задачи могут порождать бесконечное количество решений. Остановка процесса бесконечного порождения решений и «зациклившейся» программы осуществляется нажатием комбинации «Ctrl-C» и выполнением команды отладчика «a» (abort).

Вот еще один пример использования процедуры «присоединить». Здесь производится разрезание списка на два подсписка всеми возможными способами:

```
?- conc(L, R, [jack, bill, jim, tim, jim, bob]).
    L = [], R = [jack, bill, jim, tim, jim, bob];
    L = [jack], R = [bill, jim, tim, jim, bob];
    L = [jack, bill], R = [jim, tim, jim, bob];
    L = [jack, bill, jim], R = [tim, jim, bob];
    L = [jack, bill, jim, tim], R = [jim, bob];
    L = [jack, bill, jim, tim, jim], R = [bob];
    L = [jack, bill, jim, tim, jim, bob], R = [].
```

Необходимо заметить, что многие прологовские программы могут быть использованы как по прямому назначению, так и в

---

<sup>1</sup> Латинская маленькая буква «a».

---

обратном направлении. В этом, в частности, состоит сила логического программирования.

*Индексирование списка.* Задача получения  $N$ -го термина в списке определяется следующим образом:

Граничное условие: Первый терм в списке  $[H \mid T]$  есть  $H$ .

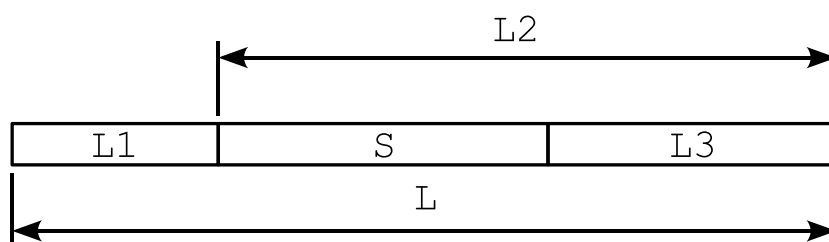
Рекурсивное условие:  $N$ -й терм в списке  $[H \mid T]$  является  $(N - 1)$ -м термом в списке  $T$ .

Данному определению соответствует программа:

```
get([H | T], 1, H).    % Граничное условие
get([H | T], N, Y) :- % Рекурсивное условие
    M is N - 1,
    get(T, M, Y).
```

*Принадлежность одного списка другому* можно проверить с помощью разбиений:

Список  $S$  является подсписком  $L$ , если  $L$  можно разбить на два списка  $L1$  и  $L2$ , и  $L2$  можно разбить на два списка  $S$  и  $L3$ .



**Рис. 3.2.** Отношение «подсписок» `sublist/2`.

```
sublist(L, S) :-
    conc(L1, L2, L),
    conc(S, L3, L2).
    % Вместо L3 можно подставить "_".
```

#### *Перестановки списка*

Если исходный список пуст, то и перестановка этого списка — пустой список. Если исходный список не пуст, то следует получить перестановку хвоста  $L$  этого списка, и затем добавить голову  $X$  списка к полученному списку.

```
rep([], [])
rep([X | L], R) :-
    rep(L, L1),
    intro(X, L1, R).
intro(X, L, [X | L]). % вставка в голову
intro(X, [Y | T], [Y | L]) :- % в хвост
    intro(X, T, L).
```

**Сортировка списков.** Рассмотрим несколько методов сортировки списков.

*Сортировка списка методом пузырька.* Для упорядочения списка  $S$  необходимо:

Найти в  $S$  два смежных элемента  $X$  и  $Y$  таких, что  $X > Y$ , и поменять их местами;

Если в  $S$  нет ни одной пары смежных элементов  $X$  и  $Y$  таких, что  $X > Y$ , то считать, что  $S$  уже отсортирован.

```
buble_sort(L1, L2) :-
    exchange_one(L1, L3), !,
    buble_sort(L3, L2).
buble_sort(L, L).

exchange_one([X, Y | T], [Y, X | T]) :-
    X > Y.
exchange_one([X, Y | T], [X | R]) :-
    \+ X > Y,
    exchange_one([Y | T], R).
```

---

*Сортировка списка методом вставки.* Для упорядочения списка  $C$  необходимо:

Пустой список считаем упорядоченным.

Упорядочить хвост списка  $C$ .

Вставить голову списка  $C$  в упорядоченный хвост, поместив ее в такое место, чтобы получившийся список остался упорядоченным.

Программу приводить не будем, оставим это как упражнение.

*Быстрая сортировка списка*

Для упорядочения списка  $C$  необходимо:

Пустой список считаем упорядоченным.

Удалить из списка первый элемент  $X$  и разбить оставшуюся часть на два списка:  $L$  — с элементами, меньшими  $X$ , и  $M$  — со всеми остальными элементами.

Упорядочить список  $L$  с получением списка  $SL$ .

Упорядочить список  $M$  с получением списка  $SM$ .

Получить результирующий упорядоченный список  $SC$  как объединение  $SL$ ,  $X$  и  $SM$ .

Например, исходный список  $C = [5, 3, 7, 8, 1, 4, 7, 6]$ .

Удаляем  $X = 5$ .

Список  $[3, 7, 8, 1, 4, 7, 6]$  разбиваем на два:

$L = [3, 1, 4]$ ;  $M = [7, 8, 7, 6]$ .

Сортируем последние списки, получаем:

$SL = [1, 3, 4]$ ;  $SM = [6, 7, 7, 8]$ .

Результат  $SC$  получаем объединением  $SL$  и  $[X \mid SM]$ .

Итак,  $SC = [1, 3, 4, 5, 6, 7, 7, 8]$ .

```
quick_sort([], []).
quick_sort([X | T], SC):-
    distrib(T, X, L, M),
    quick_sort(L, SL),
    quick_sort(M, SM),
    conc(SL, [X | SM], SC).
```

```
distrib([], _, [], []).  
distrib([H | T], X, [H | T1], L) :-  
    X > H, distrib(T, T1, L).  
distrib([H | T], X, L, [H | T1]) :-  
    \+ X > H, distrib(T, L, T1).
```

**Построение списков из фактов.** Иногда бывает полезно представить в виде списка информацию, содержащуюся в известных фактах. В большинстве реализаций Пролога есть необходимые для этого предикаты:

`bagof(X, Y, L)` определяет список термов `L`, конкретизирующих переменную `X` как аргумент предиката `Y`, которые делают истинным предикат `Y`.

`setof(X, Y, L)` все сказанное о предикате `bagof` относится и к `setof`, за исключением того, что список `L` отсортирован и из него удалены все повторения.

`findall(X, Y, L)` все сказанное о предикате `bagof` относится и к `findall`, за исключением того, что список `L` может быть пустым, если нет ни одного истинного предиката `Y`. Предикат `findall/3` всегда истинный.

Если имеются факты:

```
dog(rex).  
dog(goldy).  
dog(fido).  
dog(reke).  
dog(fido).
```

то на запрос

```
?- bagof(D, dog(D), L),
```

так же как и для `findall/3`, будет получен ответ

```
L = [rex, goldy, fido, reke, fido],
```



в то время как

```
?- setof(D, dog(D), L)
```

дает значение

```
L = [fido, goldy, reke, rex].
```

На запрос

```
findall(D, cat(D), L)
```

ответом будет

```
L = [],
```

а подобные запросы `bagof/3` и `setof/3` завершаются неудачей.

### 3.1. Грамматический разбор текста

Пролог обладает большими возможностями по сопоставлению объектов с эталоном<sup>1</sup>, поэтому данный язык программирования хорошо подходит для обработки текстов [6]. На Прологе можно с успехом реализовать генераторы отчетов, текстовые редакторы и трансляторы с различных языков. В данном разделе рассматриваются программы, предназначенные для обработки текстов. На примере этих программ демонстрируется непосредственное практическое применение систем Пролог.

Действия, выполняемые программой обработки текстов, разбиваются на две фазы. В течение первой фазы, называемой *лексическим анализом*, входной текст преобразуется из внешней формы в некоторое внутреннее представление. Во время второй фазы выполняется анализ или тот или иной вид обработки внутреннего представления текста. *Система грамматического разбора* — это процедура, которая распознает синтаксические структуры высокого уровня (объекты) во внутреннем представлении текста.

---

<sup>1</sup> Англоязычный вариант термина — pattern matching.

### 3.1.1. Стратегии грамматического разбора

Одна из причин, по которой системы грамматического разбора вызывают столь большой интерес, заключается в существовании близкой аналогии между *стратегиями грамматического разбора* и *стратегиями решения задач* в целом. В интерпретаторе языка Пролог по умолчанию принята стратегия решения задач с *обратным ходом решения* (backward chaining inference). Решение начинается с гипотезы (т. е. с запроса), которая затем разбивается на субгипотезы (т. е. подцели правила), далее каждая субгипотеза делится на еще более мелкие составные части и т. д. Исходная гипотеза будет подтверждена, когда интерпретатор дойдет до субгипотез, которые уже нельзя разделить на составные части (т. е. до фактов). Альтернативой служит стратегия решения задач с *прямым ходом решения* (forward chaining inference). Такая стратегия применяется в языках OPS-5 и CLIPS. Решение начинается с фактов, а затем отыскиваются заключения, вытекающие из них. Далее на основании этих заключений делаются заключения более высокого уровня и т. д. Это происходит до тех пор, пока не будет достигнуто искомое заключение [6].

Система нисходящего грамматического разбора базируется (как и Пролог) на стратегии с обратным ходом решения, а система восходящего разбора — на стратегии с прямым ходом решения (как и язык OPS-5). Вообще говоря, нисходящий грамматический разбор более эффективен, чем восходящий: стратегия с обратным ходом решения вовлекает в процесс фактический материал только при необходимости. В классе задач построения трансляторов существуют некоторые грамматические конструкции, разбор которых можно реализовать только восходящим методом. Поскольку сам Пролог основывается на стратегии с обратным ходом решения, реализация на Прологе систем нисходящего грамматического разбора может быть осуществлена достаточно прямолинейным способом. Реализация восходящего грамматического разбора немного сложнее, так как для применения стратегии с прямым ходом решения требуется процедурная трактовка языка Пролог. Существует также ряд других задач, для которых более предпочтительно использование стратегии с прямым ходом решения.

### 3.1.2. Лексический анализатор

Лексический анализатор распознает сочетания символов, поступающих из входного потока, и вырабатывает поток лексем. Каждая лексема представляет одну из строк символов. Множество лексем, сгенерированных лексическим анализатором, образует внутреннее представление входного потока (рис. 3.3).

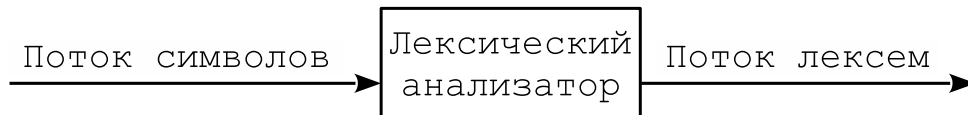


Рис. 3.3. Лексический анализатор

Реализация лексического анализатора проще всего представляется в Turbo-Prolog. В этой версии языка в стандартной библиотеке есть предикат `fronttoken/3`, который отделяет и разделяет входную строку на некоторую лексему и оставшуюся часть строки. Лексемы — это либо числа, либо знаки, либо слова, разделенные пробелами.

```

lex(Str, [Tok | Tokens]) :-
    fronttoken(Str, Tok, RStr), !,
    lex(RStr, Tokens).
lex(S, []). % Для пустых строк.
  
```

Теперь `lex/2` будет истинным по определению, если строка `Str` разбивается на список лексем `[Tok | Tokens]` по правилам предиката `fronttoken/3`.

В стандарте ISO есть предикат `read_token/2`, который считывает лексему из входного потока, например, файла или специальным образом ассоциированного с потоком атома (строки).

```

lex1(Stream, L) :-
    read_token(Stream, Term), !,
    (
        Term=punct(end_of_file), % Конец файла?
        L=[], !; % Да.
        !, % Нет.
    )
  
```

```
        conv_lex(Term, T), % Убрать разметку.
        lex1(Stream, Tail), % Следующая лексема.
        app(T, Tail, L)    %
    ).
lex(Atom, L):-
    open_input_atom_stream(Atom, Stream),
    lex1(Stream, L),
    close_input_atom_stream(Stream).
```

В этой версии предикат `lex1/2` — вспомогательный. Реализация `lex/2` в SWI-Prolog будет иной.

Проведем испытания нашего лексического анализатора в GNU-Prolog<sup>1</sup>:

```
?- lex('The cow shakes the tail', L).
   L = ['The', 'cow', 'shakes', 'the', 'tail']

lex('Slithy toves did gyre', L).
   L = ['Slythy', 'toves', 'did', 'gyre']

lex('The cow jumped over the Moon.', L).
   L = ['The', 'cow', 'jumped', 'over', 'the',
        'Moon', '.']

lex('FOR I:=1 TO 2013 DO BEGIN END ;', L).
   L = ['FOR', 'I', ':', '=', '1', 'TO', '2013',
        'DO', 'BEGIN', 'END', ';']
```

Видно, что лексический анализатор работает приемлемо.

### 3.1.3. Система нисходящего грамматического разбора

Система грамматического разбора — это программа, которая распознает синтаксические объекты в потоке лексем. Здесь описана программа нисходящего грамматического разбора `object/3`. На вход программы поступают список слов и название определенного синтаксического объекта более высокого уровня. Программа

---

<sup>1</sup> В Visual-Prolog запрос остается таким же, но одинарные кавычки надо заменить на двойные.

разбора добьется успеха, если в начале списка обнаружит слова, из которых формируется требуемый синтаксический объект. В противном случае программа потерпит неудачу. Грамматика для программы *object/3* — это несложное подмножество английского языка. Распознаваемые программой синтаксические объекты — это все части предложений английского языка, такие как «предложение», «глагольная группа» или «артикль». Программа *object/3* одновременно представляет собой и словарь, и грамматику [6].

Согласно терминологии грамматического разбора *терминальный символ* (или *терминал*) — это входная лексема, поступающая из блока синтаксического анализа, а *нетерминальный символ* (или *нетерминал*) — это синтаксический объект, образованный комбинацией терминальных или нетерминальных символов. Множество терминалов, известное системе разбора, называется ее *словарем*. Компоненты каждого нетерминального символа специфицируются при помощи *грамматического правила*, а множество грамматических правил, известных системе разбора, образует ее *грамматику*.

В описываемой системе используется простая грамматика, которая в форме Бэкуса — Наура представляется так:

```
<предложение> ::= <глагольная группа>
                  <группа существительного>
<группа существительного> ::= <артикль>
                              <существительное>
<глагольная группа> ::= <глагол>
                       <группа существительного>
```

В программе каждый терминал, входящий в состав словаря, представляется фактом *object*, а каждый нетерминал, входящий в грамматику, представляется правилом *object/3*.

```
% Принятые имена переменных
% I - входной список лексем
% O, R - выходной список лексем
% нетерминалы:
```

```
object(I, 0, 'предложение') :-  
    object(I, R, 'группа существительного'),  
    object(R, 0, 'глагольная группа').
```

```
object(I, 0, 'группа существительного') :-  
    object(I, R, 'артикль'),  
    object(R, 0, 'существительное').
```

```
object(I, 0, 'глагольная группа') :-  
    object(I, R, 'глагол'),  
    object(R, 0, 'группа существительного').
```

% терминалы:

```
object(['the' | R], R, 'артикль').  
object(['cow' | R], R, 'существительное').  
object(['tail' | R], R, 'существительное').  
object(['shakes' | R], R, 'глагол').  
object(['walks' | R], R, 'глагол').
```

Заметьте, что форма нетерминальных правил `object` в точности соответствует форме грамматических правил.

Современные реализации Пролога содержат в качестве подсистемы транслятор DCG (Definite clause grammars<sup>1</sup>) [11]. DCG удобна для представления грамматических правил при создании трансляторов. DCG не входит в стандарт ISO, и поэтому считается, что ее наличие и реализация зависит от системы программирования Пролог. Грамматические правила нашего примера в DCG будут представлены в следующем виде:

```
% Нетерминалы:  
'предложение' -> 'группа существительного',  
    'глагольная группа'.  
  
'группа существительного' ->  
    'артикль', 'существительное'.
```

---

<sup>1</sup> Грамматика, построенная на определенных предложениях.

'глагольная группа' -> 'глагол',  
'группа существительного'.

% терминалы:

'артикл' -> ['the'].

'существительное' -> ['cow', 'tail'].

'глагол' -> ['shakes', 'walks'].

Такая форма транслируется языком Пролог в представление, аналогичное представлению `object/3`, только имена правил преобразуются в такие же имена предикатов, добавятся два аргумента `I`, `O` (входной и выходной список лексем). Правила DCG могут включать дополнительные аргументы, которые при трансляции будут добавляться в заголовок к `I`, `O`. Дополнительные аргументы можно использовать для синтеза необходимых структур, например деревьев синтаксического разбора.

**Использование программы `object/3`.** В качестве первого аргумента процедуры `object` передается входной список лексем. Третий аргумент — это название определяемого объекта. Вторым аргументом образован частью списка, остающейся после того, как из начала списка будет взят терминал или нетерминал. Функции аргументов можно проиллюстрировать на примере запроса:

```
?- object(['cow', 'horse', 'goat'], Rest,  
         'существительное').  
Rest = ['horse', 'goat']
```

Запрос обозначает: «Можно ли взять существительное из начала списка `['cow', 'horse', 'goat']`, и если да, то какая часть списка останется?» Ответ на данный запрос показывает, что это возможно и что остается список `['horse', 'goat']`. Запрос подтверждает, что слово `cow` является существительным.

Подобным образом, запрос к процедуре `object/3` подтвердит или опровергнет предположение о том, что список слов образует нетерминальный символ:



```
?- object(['the', 'cow', '.' ], L,  
          'глагольная группа').  
no.
```

```
?- object(['the', 'cow', '.' ], L,  
          'группа существительного').  
L = ['.'] % успех
```

```
?- object(['the', 'cow', 'shakes', 'the',  
          'tail'], L, 'предложение').  
L = [] % успех
```

**Использование процедуры object/3 в обратном направлении.** Все аргументы процедуры object/3 являются двунаправленными. Это означает, что при помощи запросов к процедуре object можно также сгенерировать любые синтаксические объекты, какие только можно построить по входному списку лексем, или даже сгенерировать все возможные предложения по заданному словарю и грамматике.

```
% проанализировать входной список:  
?- object(['the', 'tail'], L, Object).      % (1)  
L = ['tail'],  
Object = 'артикль'.  
L = [],  
Object = 'группа существительного'  
  
% сгенерировать все предложения:  
?- object(X, [], 'предложение').          % (2)  
X = ['the', 'cow', 'shakes', 'the', 'cow'];  
X = ['the', 'cow', 'shakes', 'the', 'tail'];  
X = ['the', 'cow', 'walks', 'the', 'cow'];  
X = ['the', 'cow', 'walks', 'the', 'tail'];  
X = ['the', 'tail', 'shakes', 'the', 'cow'];  
X = ['the', 'tail', 'shakes', 'the', 'tail'];  
X = ['the', 'tail', 'walks', 'the', 'cow'];  
X = ['the', 'tail', 'walks', 'the', 'tail'].
```

Обратите внимание, что во втором запросе второй аргумент «[]» указывает, что после окончания грамматического разбора должен остаться пустой список.

Предложения, сгенерированные программе разбора, синтаксически правильны, однако их *семантическая корректность* не гарантируется. Программа может вырабатывать бессмысленные предложения вроде ['the', 'cow', 'walks', 'the', 'tail'], что переводится как «Корова идет [по] хвост[y]».

Что можно ожидать получить от следующего запроса?

?- object(A, B, C).

### Вопросы для самопроверки

1. Пусть  $L=[1, 2, 3, 4]$ ,  $L=[X1, X2 \mid Q]$ . Каково будет значение  $Q$ ?
2. Справедливо ли высказывание  $[]=[\_ \mid []]$ ?
3. Каков будет результат выполнения `findall(X, fail, L)`?
4. Дополните программу синтаксического анализа поддержкой неопределенного артикля английского языка.

## 4. Интерпретации программ Пролога

Интерпретация программы — предложение естественного языка, например русского, которое выражает смысл программы или части программы. Интерпретация, в некоторой степени, — процесс, обратный формализации модели. Интерпретации строятся, например, для того, чтобы объяснить другому человеку, не знакомому с языком Пролог, как работает программа, какие результаты она получает, что такое решение и т. д.

Различают два способа [2] интерпретации (задания смысла) программы на Прологе, а именно:

- *декларативная интерпретация;*
- *процедурная интерпретация.*

Декларативный смысл касается только **отношений**, определенных в программе. Таким образом, декларативный смысл определяет, **что** должно быть результатом работы программы. С другой стороны, процедурный смысл определяет еще и **как** этот **результат был получен**, т. е. как отношения реально обрабатываются Пролог-системой.

Способность Пролог-системы прорабатывать многие процедурные детали самостоятельно считается одним из специфических преимуществ Пролога. Это свойство побуждает программиста рассматривать декларативный смысл программы относительно независимо от ее процедурного смысла. Поскольку результаты работы программы в принципе определяются ее декларативным смыслом, последнего (опять же в принципе) достаточно для написания программ. Этот факт имеет практическое значение, поскольку декларативные аспекты программы являются, обычно, более легкими для понимания, нежели процедурные детали. Чтобы извлечь из этого обстоятельства наибольшую пользу, программисту следует сосредоточиться, главным образом, на декларативном смысле и по возможности не отвлекаться на детали процесса вычислений. Последние следует в возможно большей мере предоставить самой Пролог-системе.

Декларативный подход и в самом деле часто делает программирование на Прологе более легким, чем на таких типичных процедурно-ориентированных языках, как Паскаль. К сожалению, однако, декларативного подхода не всегда оказывается достаточно. Далее станет ясно, что, особенно в больших программах, программист не может полностью игнорировать процедурные аспекты по соображениям эффективности вычислений. Тем не менее следует поощрять декларативный стиль мышления при написании Пролог-программ, а процедурные аспекты игнорировать в тех пределах, которые устанавливаются практическими ограничениями.

Рассмотрим формальное определение декларативного и процедурного смыслов программ базисного Пролога. Но сначала давайте взглянем на различия между этими двумя семантиками. Рассмотрим предложение

$$P :- Q, R.$$

где  $P$ ,  $Q$  и  $R$  имеют синтаксис предикатов. Приведем некоторые способы декларативной интерпретации этого предложения:

$P$  — истинно, если  $Q$  и  $R$  истинны.

Из  $Q$  и  $R$  следует  $P$ .

А вот два варианта его процедурного прочтения:

Чтобы решить задачу  $P$ , **сначала** реши подзадачу  $Q$ , а **затем** — подзадачу  $R$ .

Чтобы достичь  $P$ , **сначала** достигни  $Q$ , а **затем**  $R$ .

Таким образом, различие между декларативной и процедурной интерпретациями заключается в том, что последняя определяет не только логические связи между головой предложения и целями в его теле, но еще и **порядок**, в котором эти цели обрабатываются.

## 4.1. Декларативная интерпретация

Декларативная интерпретация программы определяет, является ли данная цель истинной (достижимой), и если да, при каких

значениях переменных она истинна. Для точного определения декларативной интерпретации нам потребуется понятие *конкретизации* предложения (факта, правила или запроса). Конкретизацией предложения  $S$  называется результат подстановки в него на место каждой переменной некоторого терма. *Вариантом* предложения  $S$  называется такая конкретизация  $S$ , при которой каждая переменная заменена на число, символ, другую переменную или структуру. Например, рассмотрим предложение:

$\text{has\_a\_child}(X) \text{ :- parent}(X, Y).$

Приведем два эквивалентных варианта этого предложения:

$\text{has\_a\_child}(A) \text{ :- parent}(A, B).$

$\text{has\_a\_child}(X) \text{ :- parent}(X, Y).$

Примеры конкретизации этого же предложения:

$\text{has\_a\_child}(\text{piter}) \text{ :- parent}(\text{piter}, Z).$

$\text{has\_a\_child}(\text{barry}) \text{ :- parent}(\text{barry}, \text{caroline}).$

Пусть дана некоторая программа и цель  $G$ , тогда, в соответствии с декларативной семантикой, можно утверждать, что:

цель  $G$  истинна (т. е. достижима или логически следует из программы) тогда и только тогда, когда:

- (1) в программе существует предложение  $G$ , такое, что
- (2) существует такая его ( $G$ ) конкретизация  $I$ , что
  - (а) голова  $I$  совпадает с  $G$  и
  - (б) все цели в теле  $I$  истинны.

Это определение можно распространить на вопросы следующим образом. В общем случае вопрос к Пролог-системе представляет собой список целей, разделенных запятыми. Список целей называется истинным (достижимым), если все цели в этом списке истинны (достижимы) при одинаковых конкретизациях переменных. Значения переменных получаются в результате поиска наиболее общей конкретизации.

Таким образом, запятая между целями обозначает конъюнкцию целей: они **все** должны быть истинными. Однако в Прологе возможна и **дизъюнкция** целей: должна быть истинной **по крайней мере одна** из целей. Дизъюнкция обозначается точкой с запятой.

$$P :- Q ; R, S.$$

Это выражение читается так:  $P$  — истинно, если истинно  $Q$  **или** истинны **оба**  $R$  <sup>1</sup>  $S$ . То есть смысл такого предложения тот же, что и смысл следующей пары предложений:

$$P :- Q.$$

$$P :- R, S.$$

## 4.2. Процедурная интерпретация

Процедурная семантика определяет, **как** пролог-система отвечает на вопросы. Ответить на вопрос — это значит удовлетворить список целей. Этого можно добиться, приписав встречающимся переменным значения таким образом, чтобы цели логически следовали из программы. Можно сказать, что процедурная семантика Пролога — это процедура вычисления списка целей с учетом заданной программы. Вычислить цели — это значит попытаться достичь их.

Назовем эту процедуру вычислить. Как показано на рис. 4.1, входом и выходом этой процедуры являются:

входом — программа и список целей,

выходом — признак «успех»/«неуспех» и подстановка переменных.

Смысл двух составляющих выхода такой:

- (1) Признак «успех»/«неуспех» принимает значение да, если цели достижимы, и нет — в противном случае. Будем говорить, что да сигнализирует об **успешном** завершении, и нет — о **неуспехе**.

<sup>1</sup> Запятая имеет приоритет перед точкой с запятой.



**Рис. 4.1.** Входы и выходы процедуры вычисления списка целей

- (2) Подстановка переменных порождается процедурой только в случае успешного завершения; в случае неуспеха подстановка отсутствует.

**Пример 4.1.** Пример, иллюстрирующий процедурную семантику Пролога: последовательность вычислений, выполняемых процедурой `вычислить`.

**ПРОГРАММА.** Пусть дана следующая программа.

```

big(bear).      % Предложение 1
big(elephant).  % Предложение 2
big(cat).       % Предложение 3
brown(bear).    % Предложение 4
black(cat).     % Предложение 5
gray(elephant). % Предложение 6
dark(Z) :-      % Предложение 7:
    black(Z).   % Любой черный объект
                % является темным
dark(Z) :-      % Предложение 8:
    brown(Z).   % Любой коричневый объект
                % является темным
  
```

**ЗАПРОС.** Зададим следующий запрос.

```

?- dark(X), big(X).
    % Кто одновременно темный и большой?
  
```



## Шаги вычислений

- (1) Исходный список целевых утверждений:

`dark(X), big(X).`

- (2) Просмотр всей программы от начала к концу и поиск предложения, у которого голова сопоставима с первым целевым утверждением

`dark(X).`

Найдена формула 7:

`dark(Z) :- black(Z).`

Замена первого целевого утверждения конкретизированным телом предложения 7 - порождение нового списка целевых утверждений.

`black(X), big(X).`

- (3) Просмотр программы для нахождения предложения, сопоставимого с `black(X)`. Найдено предложение 5: `black(cat)`. У этого предложения нет тела, поэтому список целей при соответствующей конкретизации сокращается до

`big(cat).`

- (4) Просмотр программы в поисках цели `big(cat)`. Ни одно предложение не найдено. Поэтому происходит возврат к шагу (3) и отмена конкретизации `X = cat`. Список целей теперь снова

`black(X), big(X).`

Продолжение просмотра программы ниже предложения 5. Ни одно предложение не найдено. Поэтому возврат к шагу (2) и продолжение просмотра ниже предложения 7. Найдено предложение 8:

`dark(Z) :- brown(Z).`

Замена первой цели в списке на `brown(X)`, что дает

`brown(X), big(X)`

- (5) Просмотр программы для обнаружения предложения, сопоставимого `brown(X)`. Найдено предложение `brown(bear)`. У этого предложения нет тела, поэтому список целей уменьшается до

`big(bear).`

- (6) Просмотр программы и обнаружение факта `big(bear)`. У него нет тела, поэтому список целей становится пустым. Это указывает на успешное завершение.

В оставшейся части данного раздела приводится несколько более формальное и систематическое описание приведенного процесса. Конкретные операции, выполняемые в процессе вычисления целевых утверждений, показаны в приведенном примере 4.1. Возможно, следует изучить этот рисунок прежде, чем знакомиться с последующим общим описанием.

Чтобы вычислить список целевых утверждений

$$G_1, G_2, \dots, G_m,$$

процедура **вычислить** делает следующее:

- Если список целей пуст — завершает работу **успешно**.
- Если список целей не пуст, продолжает работу, выполняя (описанную далее) операцию просмотр.

- Операция просмотр: Просматривает предложения программы от начала к концу до обнаружения первого предложения  $C$ , такого, что голова  $C$  сопоставима с первой целью  $G_1$ . Если такого предложения обнаружить не удастся, то работа заканчивается **неуспехом**.

Если  $C$  найдено и имеет вид

$$H :- B_1, \dots, B_n,$$

то переменные в  $C$  переименовываются, чтобы получить такой вариант  $C'$  предложения  $C$ , в котором нет общих переменных со списком  $G_1, \dots, G_m$ . Пусть  $C'$  — это

$$H' :- B'_1, \dots, B'_n.$$

Сопоставляется  $G_1$  с  $H'$ ; пусть  $S$  — результирующая конкретизация переменных. В списке целей  $G_1, G_2, \dots, G_m$ , цель  $G_1$  заменяется на список  $B'_1, \dots, B'_n$ , что порождает новый список целей:

$$B'_1, \dots, B'_n, G_2, \dots, G_m.$$

Заметим, если  $C$  — факт, тогда  $n = 0$ , и в этом случае новый список целей оказывается короче, нежели исходный; такое уменьшение списка целей может в определенных случаях превратить его в пустой, а следовательно, — привести к успешному завершению.

Переменные в новом списке целей заменяются новыми значениями, как это предписывает конкретизация  $S$ , что порождает еще один список целей

$$B''_1, \dots, B''_n, G_2, \dots, G_m.$$

- Вычисляет (используя рекурсивно ту же самую процедуру) этот новый список целей. Если его вычисление завершается успешно, то и вычисление исходного списка целей тоже завершается успешно. Если же его вычисление порождает неуспех, тогда новый список целей отбрасывается и происходит возврат (backtrack) к просмотру программы. Этот просмотр продолжается, начиная с предложения, непосредственно следующего за использованным только-что предло-

жением  $C$ , и делается попытка достичь успешного завершения с помощью нового предложения.

#### 4.2.1. Примеры интерпретаций программ

Для того чтобы продемонстрировать «полярность» декларативной и процедурной интерпретации, рассмотрим два примера:

**Пример 4.2.** *Определить декларативную и процедурную семантику (смысл) следующего предложения языка Пролог.*

$P :- P.$

**Декларативная интерпретация.** Семантика (смысл) предложения выражается фразой: « $P$  истинно, если истинно  $P$ ». С декларативной точки зрения фраза совершенно корректна<sup>1</sup> (эта фраза — тавтология), однако в процедурном смысле это предложение бесполезно. Более того, для Пролог-системы такое предложение порождает серьезную проблему. Рассмотрим запрос:

?-  $P.$

При использовании вышеприведенного предложения  $P$  будет заменено на ту же самую цель  $P$ , она, в свою очередь, будет заменена снова на  $P$  и т. д. В этом случае система войдет в бесконечный цикл, не замечая, что никакого продвижения в вычислениях не происходит. Рассмотрим другой пример.

**Пример 4.3.** *Определить декларативную и процедурную семантику следующего предложения языка Пролог.*

$P.$

$P :- P.$

С точки зрения декларативного смысла добавление  $P$  как факта излишне, как излишне было бы добавление  $P :- P$  к факту  $P$ . В

---

<sup>1</sup> Нетрудно доказать ее истинность при любых истинностных значениях  $P$ , например при помощи метода «от противного».

обоих случаях мы имеем дело с **истинными**<sup>1</sup> высказываниями. Однако с процедурной точки зрения программа представляет интерес. Пусть в Пролог-систему введен запрос:

?- P.

Цель P сопоставляется с фактом P, и при этом получается пустое множество — выполнение завершается **удачно**. Если P — это часть тела какого-либо правила C, то в процессе доказательства C возможны возвраты, в том числе и к P. В этом случае сопоставление цели (подцели правила C) P с фактом P **отменяется** и производится замена цели P по правилу P :- P на «новую» цель P. Далее, для этой цели выполняются вычисления, начиная с сопоставления ее с фактом P. Всякий раз, отменяя предыдущее сопоставление цели P с фактом P, Пролог порождает новую (как правило, такую же) ветвь поиска сопоставлений для подцелей в C, стоящих за подцелью P. Таким образом, возврат (backtrack) из C никогда не может быть сделан<sup>2</sup>.

Описанный предикат P находит свое применение в подпрограммах пользовательских интерфейсов как способ организации бесконечного цикла. Более того, он имеет собственное имя:

Repeat.

Repeat :- Repeat.

## Вопросы для самопроверки

1. Дайте определение декларативной (описательной) интерпретации Пролог-программы.
2. Дайте определение процедурной интерпретации Пролог-программы.
3. Как можно использовать предикат repeat?
4. Какова процедурная интерпретация следующего высказывания языка Пролог: A :- Q, W, E; R, T, Y.?

<sup>1</sup> Факт P истинен по определению, а P :- P истинно, так как это тавтология (теорема).

<sup>2</sup> За исключением случая, если в теле C есть предикат «отсечение» («!»), и этот предикат будет достигнут.

## 5. Управление логическим выводом

Прежде всего программист может управлять процессом вычислений по программе, располагая ее предложения и цели в том или ином порядке [2]. В данной главе мы рассмотрим еще одно средство управления, получившее название *отсечение* (cut) и предназначенное для ограничения автоматического перебора в механизме вывода Пролога [2].

### 5.1. Ограничение перебора

В процессе достижения цели Пролог-система осуществляет автоматический перебор вариантов, делая возврат при неуспехе какого-либо из них. Такой перебор — полезный программный механизм, поскольку он освобождает пользователя от необходимости программировать его самому. С другой стороны, ничем не ограниченный перебор может стать источником **неэффективности** программы. Поэтому иногда требуется его ограничить или исключить вовсе. Для этого в Прологе предусмотрена конструкция «отсечение».

Давайте сначала рассмотрим простую программу, процесс вычислений в которой содержит ненужный перебор. Мы выделим те точки этого процесса, где перебор бесполезен и ведет к неэффективности.

Рассмотрим двухступенчатую функцию на рис. 5.1. Связь между  $X$  и  $Y$  можно определить с помощью следующих правил:

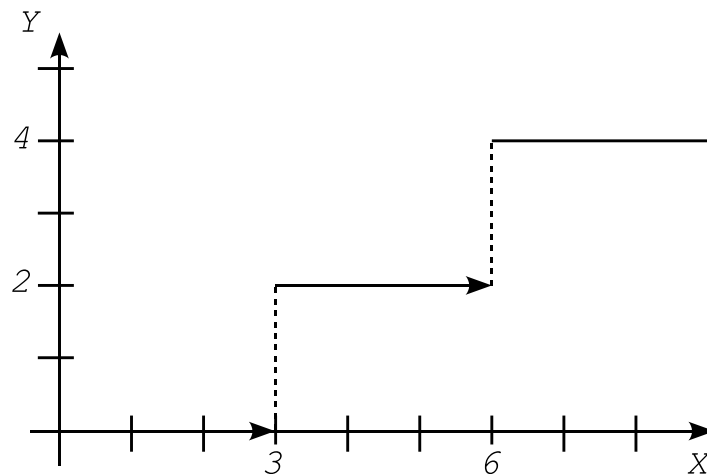
*Правило 1:* если  $X < 3$ , то  $Y = 0$ .

*Правило 2:* если  $3 \leq X < 6$ , то  $Y = 2$ .

*Правило 3:* если  $6 \leq X$ , то  $Y = 4$ .

На Прологе это можно выразить с помощью бинарного отношения  $f(X, Y)$  так:

```
f(X, 0) :- X < 3.           % Правило 1
f(X, 2) :- 3 =< X, X < 6.   % Правило 2
f(X, 4) :- 6 =< X.           % Правило 3
```



**Рис. 5.1.** Двухступенчатая функция

В этой программе предполагается, конечно, что к моменту начала вычисления  $f(X, Y)$   $X$  уже конкретизирован каким-либо числом; это необходимо для выполнения операторов сравнения.

Мы проделаем с этой программой два эксперимента. Каждый из них обнаружит в ней свой источник неэффективности, и мы устраним оба этих источника по очереди, применив отсечение.

**Эксперимент 1.** Проанализируем, что произойдет, если задать следующий запрос:

$$?- f(1, Y), 2 < Y.$$

При вычислении первой цели  $f(1, Y)$   $Y$  конкретизируется нулем. Поэтому вторая цель становится такой:

$$2 < 0.$$

Она терпит неудачу, и поэтому и весь список целей также терпит неудачу. Это очевидно, однако перед тем, как признать, что такому списку целей удовлетворить нельзя, Пролог-система при помощи возвратов попытается проверить еще две бесполезные в данном случае альтернативы.

Три правила, входящие в отношение  $f$ , являются взаимоисключающими, поэтому успех возможен, самое большее, в одном из них. Следовательно, мы (но не Пролог-система) знаем, что

как только успех наступил в одном из них, нет смысла проверять остальные, поскольку они все равно обречены на неудачу. Для предотвращения бессмысленного перебора мы должны явно указать Пролог-системе, что **не нужно** осуществлять возврат из этой точки. Мы можем сделать это при помощи конструкции отсечения. Отсечение записывается в виде символа «!», который вставляется между целями и играет роль некоторой псевдоцели. Вот наша программа, переписанная с использованием отсечения:

```
f(X, 0) :- X < 3, !.  
f(X, 2) :- 3 =< X, X < 6, !.  
f(X, 4) :- 6 =< X.
```

Символ «!» предотвращает возврат из тех точек программы, в которых он поставлен. Зададим вопрос:

```
?- f(1, Y), 2 < Y.
```

Пролог-система породит ветвь дерева. Эта ветвь потерпит неудачу на цели  $2 < 0$ . Система попытается сделать возврат, но вернуться она сможет не далее точки, помеченной в программе символом «!». Альтернативные ветви, соответствующие правилу 2 и правилу 3, порождены не будут.

Новая программа, снабженная отсечениями, во всех случаях более эффективна, чем первая версия, в которой они отсутствуют. Неудачные варианты новая программа распознает всегда быстрее, чем старая.

Вывод: добавив отсечения, мы повысили эффективность. Если их теперь убрать, программа породит тот же результат, только на его получение она потратит, скорее всего, больше времени. Можно сказать, что в нашем случае после введения отсечений мы **изменили только процедурную семантику** программы, оставив при этом ее декларативную семантику в неприкосновенности. В дальнейшем мы покажем, что использование отсечения может также затронуть и **декларативную семантику** Пролог-программы.



**Эксперимент 2.** Прделаем теперь еще один эксперимент со второй версией нашей программы. Предположим, мы задаем вопрос:

$$?- f(7, Y), Y = 4.$$

Проанализируем, что произошло. Перед тем, как был получен ответ, система пробовала применить все три правила. Эти попытки породили следующую последовательность целей:

*попытка применить правило 1:  $7 < 3$  терпит неудачу, происходит возврат и попытка применить правило 2 (точка отсечения достигнута не была);*

*попытка применить правило 2:  $3 \leq 7$  «успех», но  $7 < 6$  терпит неудачу; возврат и попытка применить правило 3 (точка отсечения снова не достигнута);*

*попытка применить правило 3:  $6 \leq 7$  — «успех».*

Приведенные этапы вычисления обнаруживают еще один источник неэффективности. Вначале выясняется, что  $X < 3$  не является истиной ( $7 < 3$  терпит неудачу). Следующая цель —  $3 \leq X$  ( $3 \leq 7$  — «успех»). Но нам известно, что если первая проверка неуспешна, то вторая обязательно будет успешной, так как второе — целевое утверждение является отрицанием первого. Следовательно, **вторая проверка лишняя** и соответствующую цель можно опустить. То же самое верно и для цели  $6 \leq X$  в правиле 3. Все эти соображения приводят к следующей, более экономной формулировке наших трех правил:

если  $X < 3$ , то  $Y = 0$ ;

иначе, если  $X < 6$ , то  $Y = 2$ ;

иначе  $Y = 4$ .

Теперь мы можем опустить в нашей программе те условия, которые обязательно выполняются при любом вычислении. Получается третья версия программы:

$f(X, 0) :- X < 3, !.$

$f(X, 2) :- X < 6, !.$

$f(X, 4).$

Эта программа дает тот же результат, что и исходная, но более эффективна, чем обе предыдущие. Однако что будет, если мы теперь удалим отсечения? Программа станет такой:

```
f(X, 0) :- X < 3.
f(X, 2) :- X < 6.
f(X, 4).
```

Она может порождать различные решения, часть из которых неверны. Например:

```
?- f( 1, Y).
    Y = 0,
    Y = 2,
    Y = 4.
```

Важно заметить, что в последней версии, в отличие от предыдущей, отсечения затрагивают не только процедурное поведение, но **изменяют** также и **декларативную семантику** Пролог-программы.

Более точный смысл механизма отсечений можно сформулировать следующим образом:

Назовем «целью-родителем» ту цель, которая сопоставилась с головой предложения, содержащего отсечение. Когда в качестве цели встречается отсечение, такая цель сразу же считается успешной и при этом заставляет систему принять те альтернативы, которые были выбраны с момента активизации цели-родителя до момента, когда встретилось отсечение. Все оставшиеся в этом промежутке (от цели-родителя до отсечения) альтернативы не рассматриваются.

Чтобы прояснить смысл этого определения, рассмотрим предложение вида

$$H :- B_1, B_2, \dots, B_m, !, \dots, B_n.$$

Будем считать, что это предложение активизировалось, когда некоторая цель  $G$  сопоставилась с  $H$ . Тогда  $G$  является целью-родителем. В момент, когда встретилось отсечение, успех уже наступил в целях  $B_1, \dots, B_m$ . При выполнении отсечения это (текущее) решение  $B_1, \dots, B_m$  **замораживается** и все возможные оставшиеся альтернативы (возможные решения) больше не рассматриваются. Далее, цель  $G$  связывается теперь с этим предложением: любая попытка сопоставить  $G$  с головой какого-либо другого предложения пресекается.

Применим эти правила к следующему примеру:

$C :- P, Q, R, !, S, T, U.$

$C :- V.$

$A :- B, C, D.$

?-  $A.$

Здесь  $A, B, C, D, P$  и т. д. имеют синтаксис предикатов. Отсечение повлияет на вычисление цели  $C$  следующим образом. Перебор будет возможен в списке целей  $P, Q, R$ ; однако как только точка отсечения будет достигнута, все альтернативные решения для этого списка изымаются из рассмотрения. Альтернативное предложение, входящее в  $C$ :

$C :- V.$

также не будет учитываться. Тем не менее, перебор будет возможен в списке целей  $S, T, U$ . Цель-родитель предложения, содержащего отсечения, — это цель  $C$  в предложении

$A :- B, C, D.$

Поэтому отсечение повлияет только на цель  $C$ . С другой стороны, оно будет невидимо из цели  $A$ . Таким образом, автоматический перебор все равно будет происходить в списке целей  $B, C, D$ , вне зависимости от наличия отсечения в предложении, которое используется для достижения  $C$ .

### 5.1.1. Примеры, использующие отсечение

**Вычисление максимума.** Процедуру нахождения наибольшего из двух чисел можно запрограммировать в виде отношения  $\text{max}(X, Y, M)$ , где  $M = X$ , если  $X$  больше или равен  $Y$ , и  $M$  есть  $Y$ , если  $X$  меньше  $Y$ . Это соответствует двум таким предложениям:

$$\begin{aligned}\text{max}(X, Y, X) &:- X \geq Y. \\ \text{max}(X, Y, Y) &:- X < Y.\end{aligned}$$

Эти правила являются взаимно исключающими. Если выполняется первое, второе обязательно потерпит неудачу. Если неудачу терпит первое, второе обязательно должно выполниться. Поэтому возможна более экономная формулировка, использующая понятие иначе:

если  $X \geq Y$ , то  $M = X$ ,  
иначе  $M = Y$ .

На Прологе это записывается при помощи отсечения:

$$\begin{aligned}\text{max}(X, Y, X) &:- X \geq Y, !. \\ \text{max}(X, Y, Y) &.\end{aligned}$$

**Проверка принадлежности списку, порождающая единственное решение.** Для того чтобы узнать, принадлежит ли  $X$  списку  $L$ , использовано отношением  $\text{in}/2$ , и программа была следующей:

$$\begin{aligned}\text{in}(R, [R \mid T]) &.\ \\ \text{in}(R, [H \mid T]) &:- \text{in}(R, T).\end{aligned}$$

Эта программа дает «недетерминированный» ответ: если  $R$  встречается в списке несколько раз, то будет найдено **каждое** его вхождение. Следующая программа исправляет этот «недостаток» и выдает только одно решение.

$$\begin{aligned}\text{in}(R, [R \mid T]) &:- !. \\ \text{in}(R, [H \mid T]) &:- \text{in}(R, T).\end{aligned}$$

### 5.1.2. Другие предикаты управления выводом

В языке программирования Пролог существуют еще несколько предикатов и логическая связка, используемые для управления процессом вывода.

$\backslash +$  Логическая операция отрицания « $\neg$ » имеет, с одной стороны, противоположное истинностное значение истинностному значению предиката, который следует за этой операцией, с другой стороны,  $\backslash +$  G ложен, если Пролог для G смог построить вывод.

`fail/0` Этот предикат обозначает «ложь», т. е. если он включен как подцель тела правила или запроса, то это правило и запрос никогда не будут доказаны. Предикат используется для организации перебора всех решений в части программы, где задана цель `init/0`, например, программа

```
number(1). number(2). number(3).
number(4). number(5). number(6).

init:-
    number(X), number(Y), write(X),
    write(Y), fail.
:- initialization(init).
```

выведет на экран все двузначные комбинации цифр от 1 до 6. Если убрать `fail` из запроса, то будет выведена только одна комбинация.

`nonvar/1` В ISO-Prolog предикат `nonvar/1` истинный, если переменная-аргумент к этому моменту уже приобрела свое значение, и ложный — в обратном случае.

`var/1` Предикат `var/1` истинный тогда и только тогда, когда предикат `nonvar/1` ложный.

**Пример 5.1.** Пусть дано уравнение

$$y = k \cdot x,$$

*причем комбинации значений переменных и их отсутствия могут быть любыми. Предложить программу, определяющую значения переменных, которые неизвестны, либо выдать на экран сообщение о том, что решение найти невозможно. Равенство должно сохраняться.*

Решение задачи (программа) будет строиться для каждого набора переменных, всего их  $2^3 = 8^1$ , но некоторые могут быть объединены.

```
solve(Y, K, X) :-  
    nonvar(K), nonvar(X), !,  
    % Y может быть как связанным, так и  
    % свободным.  
    Y is K * X.  
solve(Y, K, X) :-  
    nonvar(Y), nonvar(X),  
    X <> 0,  
    K is Y / X.  
solve(Y, K, X) :-  
    nonvar(Y), nonvar(X),  
    X = 0,  
    write('Нет решений'), nl.  
solve(Y, K, X) :-  
    nonvar(Y), nonvar(K),  
    % сократим себе программу!  
    solve(Y, X, K).  
solve(Y, K, X) :-  
    var(X), var(K), nonvar(X)  
    write('Множество решений'), nl,  
    format('X = Y / ~w', [K]), nl.  
.....% и т. д.
```

---

<sup>1</sup> Надо учесть также комбинации нулевых значений.

## 5.2. Лабораторная работа 2: Списки

**Задание.** В работе требуется реализовать **по крайней мере два отношения** из индивидуального задания в виде правил и фактов на языке Пролог. К программе требуется выполнить ряд запросов, объяснить выдаваемые системой результаты: дать процедурную и декларативную интерпретацию определенных отношений.

**Цель работы.** Приобрести навыки рекурсивной обработки рекурсивных структур данных; научиться интерпретировать (переводить на естественный язык) Пролог-программы.

**Время работы.** На выполнение работы отводится два академических часа.

### Индивидуальные задания

1. Сформировать новый список из всех четных элементов исходного списка.
2. Определить, является ли один список подсписком другого.
3. Удалить все вхождения заданного элемента из списка.
4. Написать программу пословного (подстрочного) перевода предложения, представленного в виде списка слов, с английского на французский (или любой другой) язык.
5. Сформировать новый список, в котором каждый элемент исходного списка входит в новый список два раза подряд.
6. Слить два упорядоченных списка в один, сохранив упорядоченность.
7. Определить, являются ли два заданных элемента соседними в списке.
8. Определить последний элемент списка.
9. Подсчитать количество элементов списка.
10. Упорядочить список методом пузырька.

11. Упорядочить список методом вставки.
12. Упорядочить список методом быстрой сортировки.
13. Заменить элемент списка на заданное значение.
14. Определить все перестановки элементов списка.
15. Найти сумму элементов списка, стоящих на нечетных местах в списке.
16. Инвертировать список.
17. Добавить элемент в конец списка.
18. Удалить два последних элемента списка.
19. Найти максимальный элемент списка.
20. В заданном списке выделить подсписок, содержащий  $N$  элементов с начала списка.
21. В заданном списке выделить подсписок, начиная с  $N$ -го элемента списка и кончая  $K$ -м элементом этого же списка.
22. Определить сумму отрицательных элементов списка, стоящих на четных местах.
23. Удалить из списка максимальный элемент.
24. Выполнить циклический сдвиг списка на заданное число элементов. Например, результат сдвига списка  $[a, b, c, d]$  на два элемента есть список  $[c, d, a, b]$ .
25. Определить предикат  $\text{code}(X, \text{List})$ , где  $X$  — целое неотрицательное число ( $0 \leq X \leq 9$ ), а  $\text{List}$  — это последовательность единиц, число которых равно  $X$ . Например,  $\text{code}(3, L)$  дает  $L = [1, 1, 1]$ , а  $\text{code}(0, L)$  дает  $L = []$ .
26. Определить двухаргументный предикат  $\text{translate}(C1, C2)$  для перевода списка цифр в список соответствующих слов. Например, истинным будет следующее высказывание  $\text{translate}([3, 4, 8], ['\text{три}', '\text{четыре}', '\text{восемь}'])$ .



## Методические указания к лабораторной работе

Сначала необходимо понять, между какими объектами задается отношение, его арифность (все то же самое, как и в предыдущей лабораторной работе). Далее:

1. Сначала определить частный случай (самый простой) отношения (в терминах доказательства теорем методом математической индукции)  $P(0, \dots)$ , базу индукции.
2. Затем в предположении, что условие базы индукции ложно, определить для остальных случаев  $P(n, \dots) \rightarrow P(n + 1, \dots)$ .

**Пример 5.2.** *Задача — определить последний элемент списка.*

Отношение задается между списками и элементами списка. Арифность отношения — 2.

Частный случай — одноэлементный список (пустой список не имеет последнего элемента). Предикат интерпретируется следующим образом: «Последний элемент одноэлементного списка и есть этот единственный элемент».

$$\text{last}([X], X).$$

Пусть список содержит не меньше одного элемента (отрицание предыдущего утверждения), тогда последний элемент списка — это последний элемент его хвоста.

$$\text{last}([_ | T], X) :- \text{last}(T, X).$$

*Декларативная интерпретация:* Последний элемент одноэлементного списка и есть требуемый элемент (решение). Если список содержит более одного элемента, то последний элемент этого списка — это последний элемент хвоста.

*Процедурная интерпретация:* Чтобы найти последний элемент списка, нужно: если список содержит лишь один элемент, то предъявить в качестве второго аргумента отношения (результата) `last/2` первый элемент списка. Иначе, если в списке содержится более одного элемента, то: (а) найти последний элемент хвоста и (б) предъявить его как результат.

### Вопросы для самопроверки

1. Что является причиной неэффективности системы программирования Пролог?
2. Какова семантика предиката «отсечение»?
3. Каково будет истинностное значение запроса `?- var(X).?`
4. Какова процедурная интерпретация правила `A :- Q,W,E,!,R,T,Y.?`
5. Возымеет ли действие отсечение в запросе `fail,!.`?

## 6. Предикаты с побочными действиями

Пролог обладает возможностью не только определять условия, когда значения предикатов истинны, но и выполнять некоторую «полезную» работу, такую как получать значение с клавиатуры, выводить сообщения на экран, складывать и вычитать и т. д. Предикаты, выполняющие эти действия, делятся на два класса: *арифметические* и так называемые *предикаты с побочными действиями*.

Для выполнения арифметических операций используется предикат «is/2».

**Пример 6.1.** *Решение квадратного уравнения «по-прологовски». Пусть даны  $a, b, c$  такие, что*

$$ax^2 + bx + c = 0.$$

*Найти такие значения  $x$ , чтобы равенство сохранялось<sup>1</sup>.*

```
sseq(A, B, C, X) :-  
    D is B * B + 4 * A * C,  
    sd(A, B, C, D, X).  
sd(A, B, C, D, X) :-  
    D < 0,  
    write('Решения в области'), nl,  
    write('комплексных чисел. '), nl,  
    !, fail.  
sd(A, B, C, D, X) :-  
    D = 0,  
    write('Четные корни'), nl,  
    X is - (B * B) / (2 * A).  
sd(A, B, C, D, X) :-  
    D > 0,  
    X is ( - (B * B) + sqrt(D) ) / (2 * A).  
sd(A, B, C, D, X) :-  
    D > 0,  
    X is ( - (B * B) - sqrt(D) ) / (2 * A).
```

---

<sup>1</sup> Предполагаем, что  $A, B, C$  заданы (конкретизированы).

Кроме арифметических операций в программе использованы предикаты с побочными действиями `write/1` и `nl/0`. Кроме этих необходимо выделить следующие предикаты для ввода данных в GNU-Prolog:

`read_token/1` вводит с клавиатуры лексическую единицу до нажатия клавиши «Enter». Вариант лексической единицы выдается в виде атома. Например, если пользователь с клавиатуры вводит «10», то переменная аргумента конкретизируется атомом «10», если пользователь вводит «F», то аргумент конкретизируется атомом «var('F')».

`read_term/1` или просто `read/1` позволяет вводить с клавиатуры терм: пользователь должен ввести синтаксически правильную конструкцию, заканчивающуюся точкой «.». Пример:

```
?- read(A).
a(s),b(k).      % Вводит пользователь.

A = (a(s),b(k))

yes.
```

Для отображения результатов вывода удобно использовать предикат `format/2`. Предикат выполняет аналогичную функцию формирования сообщения из индивидуальных составляющих и вывода на экран (или в файл), что и функция `printf`. Спецификация предиката — `format(Format, Arguments)`, где `Format` — это форматная строка, `Arguments` — список аргументов, которые будут подставлены в форматную строку. Например, запрос

```
?- A=8, B=c(Y), E='test',
   format('A=~w, B=~w, E=~w ~n !!! ',
         [A,B,E]).
A=8, B=c(_22), E=test
!!!
```

```
A = 8,  
B = c(Y),  
E = test.
```

```
yes.
```

Здесь ~w обозначает печать аргумента при помощи средств предиката `write/1`, ~n — перевод строки. В этом примере терм `B=c(Y)` распечатался как `B=C_22`: машине вывода Пролога все равно, как называются переменные, которые вводятся процедурой поиска логического вывода Пролог, имена порождаются последовательным перебором чисел.

## 6.1. Базы данных в Прологе

Очень коротко рассмотрим возможности Пролога вести базу данных. *Базой данных* Пролога называется **изменяемый** набор фактов (а также правил)<sup>1</sup>. Чтобы набор фактов был изменяемым, надо его таковым объявить:

```
% группа: обозначение, название  
:- dynamic(group/2).  
% студент: номер зачетки, группа, Ф. И. О.  
:- dynamic(student/3).
```

Как нетрудно догадаться, запросы к базе данных в Прологе делаются так же, как и к другим правилам и фактам (предполагается, что база данных Прологу известна):

```
% в какой группе учится студент  
% Иванов Иван  
?- group(X),  
   student(_, X, 'Иванов Иван').  
X = '02421',  
% Выдать всех студентов указанной группы  
?- group(X, '02421'),
```

---

<sup>1</sup> Вы уже знакомы с предикатами `consult/1` и `reconsult`.

```
student(_, X, Y).  
X = '02421', Y = 'Иванов Иван';  
. . . . .
```

Добавлять и удалять факты в базе данных можно с помощью предикатов с побочными действиями `assert/1` и `retract/1`:

```
add_student(Group, Num, Name) :-  
    assert(student(Num, Group, Name)).
```

Замечание: все аргументы предиката `assert/1` должны быть **конкретизированными** переменными.

```
del_student(Group, Num, Name) :-  
    retract(student(Num, Group, Name)).
```

Предыдущее замечание не имеет силы для `retract/1`, однако он будет истинным, если удалит первый попавшийся факт, удовлетворяющий ограничениям, указанным значениями переменных.

Существуют разновидности предиката добавления факта в базу данных: `asserta/1` и `assertz/1`, которые добавляют факты, соответственно, в начало или конец существующего списка фактов.

Базу данных можно загрузить из текстового файла при помощи предиката `consult/1`, причем выполнить его надо после объявления предиката динамическим.

## 6.2. Лабораторная работа 3: Базы данных

**Задание.** Необходимо разработать **интерактивную** программу ведения базы данных. База данных должна содержать **по крайней мере одно** отношение «один-ко-многим» ( $1:N$ ) или «многие-ко-многим» ( $N:M$ ). Кроме того, необходимо реализовать выполнение **запроса**, демонстрирующего эту связь, а также корректно реализовать функцию удаления записей.

**Цель работы.** Ознакомиться с вариантом реализации принципа реляционных баз данных в виде Пролог-программы. Научиться пользоваться предикатами с побочными действиями, а также управлять процессом поиска решения Пролог-системой.

**Время работы.** На выполнение работы отводится четыре академических часа.

### Методические указания к лабораторной работе

Непривычному к логическому программированию студенту довольно трудно реализовать меню на Прологе, однако, это сделать легче, чем в каком-либо другом языке программирования. Внимательно разберите следующий пример:

```
menu_do(1) :-
    write('Приступим к добавлению записи...'),
    .....
menu_do(0).

main :-
    repeat,
    write('Меню программы:'), nl,
    write('1 - добавление ...'), nl,
    .....
    write('0 - выход из программы'), nl,
    write(' > '), read_int(I),
    menu_do(I),
    I = 0, !; true.
```

**Индивидуальные задания.** Разработайте программу управления базой данных в следующих предметных областях<sup>1</sup>.

---

<sup>1</sup> Взято с <http://crec.mipt.ru/study/materials/db/DBvariants.doc>.

1. Институт (деканаты, кафедры, учебный отдел).

- Студенты: паспортные данные, адрес, дата зачисления, номер приказа, факультет, группа, является ли старостой, кафедра (специализация), изучаемые (изученные) предметы, оценки, задолженности, стипендия.
- Учебные курсы: название, факультет(ы), групп(ы), кафедра, семестр(ы), форма отчетности, число часов.
- Преподаватели: паспортные данные, адрес, телефон, фотография, кафедра, должность, ученая степень, начальник (зав. кафедрой), предмет(ы), число ставок, зарплата.

2. Библиотека института.

- Книги: авторы, название, раздел УДК, раздел (техническая, общественно-политическая и т. п.), место и год издания, издательство, количество страниц, иллюстрированность, цена, дата покупки, номер сопроводительного документа (чек, счет/накладная), вид издания (книги, учебники, брошюры, периодические издания), инвентарный номер (есть только для книг и некоторых учебников), длительность использования читателями (год, две недели, день), электронная версия книги или ее реферата (отсканированный текст).
- Читатели: номер читательского билета, ФИО, год рождения, адрес, дата записи, вид (студент, аспирант, преподаватель, сотрудник), курс, номер группы, названия взятых книг и даты их выдачи.

3. Отдел кадров и бухгалтерия некоторой компании.

- Сотрудники: ФИО, паспортные данные, фотография, дом. и моб. телефоны, отдел, комната, раб. телефоны (в том числе местный), подчиненные сотрудники, должность, тип(ы) работы, задание(я), проект(ы), размер зарплаты, форма зарплаты (почасовая, фиксированная).



- Отделы: название, комната, телефон(ы), начальник, размер финансирования, число сотрудников.
- Проекты: название, дата начала, дата окончания, размер финансирования, тип финансирования (периодический, разовый), задачи и их исполнители, структура затрат и статьи расходов.

4. Отдел поставок некоторого предприятия:

- Поставщики: название компании, ФИО контактного лица, расчетный счет в банке, телефон, факс, поставляемое оборудование (материалы), даты поставок (по договорам и реальные), метод и стоимость доставки.
- Сырье: тип, марка, минимальный запас на складе, время задержки, цена, продукты, при производстве которых используется, потребляемые объемы (необходимый, реальный, на единицу продукции).

5. Технологический отдел некоторого предприятия:

- Производственные процессы: продукты, объемы их производства, необходимые материалы, количества разных видов материалов на единицу продукции, отходы производства; используемое оборудование и его тип, даты ввода оборудования в строй, сроки амортизации, производительность оборудования; человеческие ресурсы (сколько всего и сколько по производству единицы продукции — сколько необходимо и сколько реально).
- Материалы: тип (категория), марка, является ли сырьем (или производится на предприятии), потребляемые объемы (в том числе на единицу конечной продукции), в рамках каких технологических процессов используется, цена.

6. Отдел продаж некоторой фирмы.

- Клиенты: название компании, ФИО контактного лица, адрес выставления счета, адрес доставки, телефон, факс.

- Заказы: тип заказа (покупка, гарантийный ремонт, негарантийный ремонт), общая стоимость, скидка, товар(ы), их изготовители, модели (марки), серийные номера, описание неисправностей, необходимые ресурсы, клиент, дата получения заказа, срок завершения, дата выставления счета и его оплаты, метод оплаты, дата поставки, метод и стоимость доставки.
- Ресурсы: ФИО, отдел(ы) и телефон(ы) исполнителя(ей), число рабочих часов для выполнения заказа, ставка зарплаты, ответственный за выполнение заказа, необходимое оборудование и расходные материалы, их количество и стоимость, а также наличие материалов.

#### 7. Магазин (внутренний учет).

- Клиенты: юридическое или физическое лицо, ФИО, адрес, телефон, адрес выставления счета, вид и номер карточки, факс.
- Продажи: наименования, модели (марки) и серийные номера товаров, поставка из магазина или со склада, количество и общая стоимость товаров, размер скидки, тип скидки, форма оплаты (наличными, оплата счета, по карточке), необходимость доставки, стоимость и тип доставки, адрес доставки.
- Товары: категория, модель, название производителя, адрес производителя, цена, количество в магазине и на складе.

#### 8. Электронный магазин (информация для клиентов).

- Товары: категория, модель, производитель, цены (в том числе средняя и минимальная), есть ли в наличии, описание, характеристики, внешний вид; магазины, где можно купить товар, их телефоны и адреса; аксессуары, их цены и где их купить.
- Магазины: название, компания-владелец, ее юридический адрес и home-site, контактные телефоны, адрес,

схема проезда, эмблема; товары и цены на них; рекламная информация: некоторые товары с фотографиями, описаниями и ценами, основные отделы.

9. Пункт проката видеозаписей (внутренний учет).

- Видеокассеты: идентификационный номер видеокассеты, тип видеокассет, дата его создания, компания-поставщик, число штук данного типа (общее, в магазине, выдано в настоящее время, выдано всего, выдано в среднем за месяц), общая длительность записей; записи видеокассет: название, длительность, категория, год выпуска и производитель (оригинала).
- Клиенты: ФИО, паспортные данные, адрес, телефон; заказы, т. е. взятые видеокассеты (сейчас и в прошлом): номер, дата выдачи, дата возвращения, общая стоимость заказа.

10. Пункт проката видеозаписей (информация для клиентов).

- Видеокассеты: краткое описание, внешний вид (этикетка), марка (пустой) видеокассеты, цена за единицу прокатного времени (например: 1 день, 3 дня, неделя), есть ли в наличии, общая длительность записей; записи на видеокассете: название, длительность, жанр (категория), тема, год и страна выпуска (оригинала), кинокомпания, описание, актеры, режиссер.
- Заказы: идентификационные номера и названия выданных видеокассет, дата выдачи, дата возвращения (продления), общая стоимость заказа, возвращены ли кассеты заказа.

11. Кинотеатры (информация для зрителей).

- Фильмы: название, описание, жанр (категория), длительность, популярность (рейтинг, число проданных билетов в России и в мире), показывается ли сейчас (сегодня, на текущей неделе), в каких кинотеатрах показывается, цены на билеты (в том числе средние).

- Кинотеатры: название, адрес, схема проезда, описание, число мест (в разных залах, если их несколько), акустическая система, широкоэкранность, фильмы и цены на них: детские и взрослые билеты в зависимости от сеанса (дневной, вечерний и т. п.) и от категории мест (передние, задние и т. п.); сеансы показа фильмов (дата и время начала).

12. Ресторан (информация для посетителей).

- Меню: дневное/вечернее, список блюд по категориям.
- Блюда: цена, название, вид кухни, категории (первое, второе и т. п.; мясное, рыбное, салат и т. п.), является ли вегетарианским, компоненты блюда, время приготовления, есть ли в наличии.
- Компоненты блюд: тип (гарнир, соус, мясо и т. п.), калорийность, цена, рецепт, время приготовления, есть ли в наличии, ингредиенты (продукты) и их расходы на одну порцию.

13. Аналитический отдел некоторой компании (поиск и анализ публикаций).

- Категории: название, тип (область исследований, область приложений и т. п.), родительская категория, дочерние категории, связанные по смыслу категории (с пояснениями о связях), найденные публикации.
- Публикации: название, тип (газетная, книжная, веб и т. п.), название, тип, адрес и телефон источника (газета, книга, сайт и т. п.), выходные данные (date-line), язык, реферат, ключевые слова, категории (с указанием степени уверенности отнесения к ним), текст и его тип (обычный, DOC, HTML, отсканированные картинки и т. п.), обзор.
- Задачи: тип задачи (классификация или поиск), сотрудник (создавший категорию или нашедший публикацию,

ответственный за категорию или публикацию и т. п.), завершена ли работа над задачей.

14. Аналитический отдел некоторой компании (анализ рынка технологий, например по публикациям, см. п. 13).

- Организация: название, тип (промышленная, финансовая, торговая, исследовательская и т. п.), категория(и), организация-владелец (акционеры), страна, контактная информация; договорные отношения с другими организациями.
- Технология (продукт): название, категория(и), организация-разработчик и производитель(и), использующие организации.
- Человек: фамилия, имя, тип (начальник, менеджер, создатель технологии и т. п.), организация(и), в которой работает, контактная информация.

### Вопросы для самопроверки

1. В чем основное отличие предикатов «is» и «=»?
2. Какая директива объявляет предикат динамическим?
3. В каком случае выполнение `retract(_)` будет успешным?
4. Дайте декларативную интерпретацию запроса `retract(_), fail.`

## 7. Поиск решения на основе перебора

Иногда приходится сталкиваться с задачами, эффективный алгоритм решения которых очевидным образом реализовать не удастся, либо нет достаточно времени на анализ свойств задачи и поиск подходящего алгоритма. К таким задачам относятся, например, *задачи с удовлетворением ограничений*<sup>1</sup>. Такие задачи являются математическими проблемами, определенными на конечном наборе объектов, чьи значения должны удовлетворять ряду ограничений, выраженных в виде неравенств и логических выражений. Исследования в области решения задач CSP ведутся достаточно давно, и по сей день актуальность этих исследований только повышается.

К задачам CSP сводятся многие задачи искусственного интеллекта, в частности планирование действий. Задачи удовлетворения ограничений довольно часто демонстрируют большую комбинаторную сложность, и практически для каждой индивидуальной задачи строятся собственные варианты эвристических алгоритмов<sup>2</sup> их решения. Примеры известных задач — «Восемь ферзей», «Раскраска карты», «Судoku», «Поиск выполняющего набора», «Составление расписания вуза».

Формально задачи CSP определяются следующим образом. Заданы вектор переменных  $\vec{V} = \langle v_1, v_2, \dots, v_n \rangle$ , где  $n$  — количество переменных (натуральное); множество множеств  $D = \{d_1, d_2, \dots, d_n\}$ , где  $d_i$  — непустое множество значений (домен), которые может принимать переменная  $v_i$ ,  $i = 1, 2, \dots, n$ . задается также логическое условие  $P(\vec{V}) = P(v_1, v_2, \dots, v_n)$ , которое истинно, если значения, присвоенные переменным  $v_i$ , соответствуют условиям правильной комбинации. Например, для задачи «Восемь ферзей»  $P(\vec{V})$  истинно, если все ферзи, расставленные на доске, не бьют друг друга. Иногда говорят о системе ограничений и о таких значениях переменных, при которых все ограничения выполняются (истинны). Систему ограничений можно

<sup>1</sup> Англ. *Constraint satisfaction problems, CSP*.

<sup>2</sup> Алгоритмов, где на этапах выбора очередного направления поиска решения из нескольких альтернатив используется дополнительная (по отношению к исходной) информация для задания этим альтернативам некоторого предпочтения.

записать как конъюнкцию индивидуальных ограничений, т. е. свести опять же к единому логическому условию  $P(\vec{V})$ .

## 7.1. Алгоритм «Британского музея»

Одним из простых алгоритмов, решающих задачи CSP, является алгоритм «Британского музея»<sup>1</sup>. Алгоритм реализует самый общий подход в задачах поиска решения на основе последовательной проверки всех возможностей (одну за одной), начиная с самых простых решений.

Алгоритм реализует концептуальный, а не практический подход, оперируя огромным количеством возможных альтернатив. В частности, в теории он представляет способ найти самую короткую программу, которая решает конкретную задачу. Например, можно сгенерировать все возможные программы длиной в один символ. Проверить каждую программу, решает ли она эту задачу<sup>2</sup>. Если среди односимвольных программ не найдено программы, решающей задачу, перейти к просмотру программ длиной в два символа, затем в три и т. д. В теории такой подход позволяет найти самую короткую программу, однако на практике такой перебор занимает недопустимо большое время вычислений (для многих задач больше, чем возраст Вселенной).

Название данный алгоритм получил ввиду высказывания Аллена Ньюэлла, Дж. С. Шоу и Герберта А. Симона в 1958 году: «... Вполне уместно предположить, что если посадить обезьян за печатные машинки, можно через некоторое время воспроизвести все книги в известном Британском музее» в Лондоне.

Несмотря на всю идеалистичность подхода для задач небольшой размерности, алгоритм вполне пригоден. Для начала, конечно, нет необходимости порождать программы, достаточно порождать варианты значений переменных  $v_i$ , затем проверять выполнимость  $P(\vec{V})$ . Рассмотрим пример задачи:

<sup>1</sup> В советской литературе вариант этого подхода известен как метод «Отобразить и проверить» (источник информации, к сожалению, потерян). В настоящее время широко используется термин «Метод грубой силы» (Brute Force Approach) для обозначения данного подхода.

<sup>2</sup> Существует фундаментальная проблема остановки, которая делает такую проверку, в общем случае, невозможной.

**Пример 7.1.** Разработать программу поиска списка счастливых билетов, состоящих из шести цифр. Подсчитать их количество.

```
num(X) :- member(X, [0,1,2,3,4,5,6,7,8,9]).  
gen([]).  
gen([X|T]) :- num(X), gen(T).
```

```
p([A,B,C, D,E,F]) :-  
    A + B + C == D + E + F.
```

```
lucky([A,B,C, D,E,F]) :-  
    gen([A,B,C, D,E,F]),  
    p([A,B,C, D,E,F]).
```

Программа при помощи предиката `gen/1` порождает идентификаторы билетов. Предикат `p/1` проверяет, является ли билет счастливым. Процедура порождения списка счастливых билетов оформлена в виде предиката `lucky/1` и в комментариях не нуждается. Для запуска программы порождения списка надо выполнить команду:

```
?- lucky(L).
```

```
L = [0,0,0,0,0,0] ? ;  
L = [0,0,1,0,0,1] ? ;  
L = [0,0,1,0,1,0] ? ;  
L = [0,0,1,1,0,0] ? ;  
L = [0,0,2,0,0,2] ?
```

```
yes.
```

Для подсчета количества счастливых билетов создадим еще одно вспомогательное правило:

```
count(N) :- findall(Ticket, lucky(Ticket),  
    Tickets), length(Tickets, N).
```



Данное правило позволяет подсчитывать количество счастливых билетов, но не выводить их полный список на экран.

Выполним запрос (GNU-Prolog):

```
?- count(N).
```

```
N = 55252
```

```
(630 ms) yes.
```

Приведенные программы являются также примерами использования стандартных `member/2` и `length/2` предикатов обработки списков.

**Ускорение решения задачи.** Программа перебирает  $10^6$  вариантов, из которых, как мы только что увидели, только около  $5.5 \cdot 10^4$  относятся к решению задачи. То есть примерно один из двадцати билетов — счастливый. Возникает вопрос: Можно ли усовершенствовать программу, чтобы уменьшить количество неправильных вариантов<sup>1</sup> и сэкономить время решения задачи на проверке этих неправильных вариантов?

Первым делом давайте попробуем вычислить значение переменной `F is A+B+C-D-E`. Добавим к программе следующий код:

```
lucky2([A,B,C, D,E,F]) :-
    gen([A,B,C, D,E]),
    F is A+B+C-D-E,
    num(F),
    p([A,B,C, D,E,F]).
```

```
count2(N) :- findall(Ticket, lucky2(Ticket),
    Tickets), length(Tickets, N).
```

```
?- count2(N).
```

```
N = 55252
```

```
(133 ms) yes.
```

<sup>1</sup> Часто требуется как можно быстрее найти первое решение или самое короткое решение. В этом случае можно рассматривать и сокращение перебора и в области решений.

Получено такое же количество решений, но за время в пять раз меньшее. Вычисленное значение  $F$  может быть отрицательным и больше 9, что противоречит условиям задачи, поэтому в новую процедуру порождения билетов необходимо добавить дополнительную проверку  $\text{num}(F)$ , которая выполняется, если  $F$  находится в требуемом диапазоне. Теперь порождается в 10 раз меньше билетов, даже с учетом тех, где  $F$  находится вне диапазона. То есть каждый второй сгенерированный билет — счастливый. Если убрать уже ненужную повторную проверку  $p/1$ , то скорость исполнения программы увеличится еще на 30 % до 106 микросекунд, т. е. уже более, чем в 6 раз быстрее первоначальной.

?- count2(N).

N = 55252

(103 ms) yes.

**Дополнительное ускорение.** Теперь попробуем найти два последних числа. Выражение  $A+B+C-D$  изменяется в пределах  $-9, -8, \dots, 0, 1, \dots, 26, 27$ : от  $0+0+0-9$  до  $9+9+9-0$ . Варианты, кода результат выражения — отрицательный заведомо, неподходящие, так же как если этот результат больше 18,  $9+9+9-9$ . Можно еще усовершенствовать алгоритм, но оставим это в качестве упражнения. Теперь надо разработать подпрограммы, которые будут для диапазона  $0, 1, \dots, 18$ . Будем решать просто отдельную переборную задачу: Задано число  $S \in 0, 1, \dots, 18$ , найти два слагаемых  $E$  и  $F$ , дающих в сумме  $N$ . Дополним программу следующим кодом:

```
lucky3([A,B,C, D,E,F]) :-
    gen([A,B,C, D]),
    S is A+B+C-D,
    S >= 0, S <= 18,
    gen2(S, E,F).
```

```
count3(N) :- findall(Ticket, lucky3(Ticket),
    Tickets), length(Tickets, N).
```

```

gen2(0,0,0):-!. % Выделим отдельно наглядные
gen2(18,9,9):-!. % тривиальные варианты.
gen2(N,A,B):-N<10, !, igen(N,A), B is N - A.
gen2(N,A,B):-D is N - 9, Z is 9 - D,
            igen(Z, A1), A is A1 + D, B is N - A.

% igen(N, A) для A порождает последовательности
% 0,1,2,...,N
igen(N, A) :- N>=1, M is N - 1, igen(M, A).
igen(N, N).

```

Запускаем запрос:

```

?- count3(N).
N = 55252

```

(47 ms) yes.

Теперь программа работает в 13 с лишним раз быстрее первоначальной и в 2 раза быстрее предыдущей, т. е. примерно один из трех билетов не является счастливым. Конечно, программу можно совершенствовать дальше: перейти к порождению первых трех цифр, и, отталкиваясь от полученной суммы трех первых цифр, по аналогии с последним примером порождать соответствующие последовательности. Однако необходимо заметить, что программа<sup>1</sup> постепенно становится сложной, а текст все меньше и меньше воспринимаемым. Многие задачи требуют с каждым усовершенствованием увеличения программы в 2 раза для сокращения перебора ненужных вариантов также в 2 раза. Поэтому важно вовремя остановиться, либо искать какой-либо новый подход.

## 7.2. Лабораторная работа 4: Перебор

**Задание.** В работе требуется реализовать и, по возможности, усовершенствовать переборный алгоритм для **одной из задач**

<sup>1</sup> Автор пособия не ставил целью **найти** эффективную и короткую программу для решения этой задачи. Задача — продемонстрировать ход рассуждений.

из списка индивидуальных заданий. В процессе усовершенствования программы анализируйте трудоемкость очередного этапа и полученное ускорение алгоритма. Отметим, что к категории переборных и оптимизационных относятся большинство олимпиадных задач, несколько из них приведены в качестве варианта задания.

**Цель работы.** Приобрести навыки поиска решений задач с удовлетворением ограничений при помощи полного и частичного перебора; приобрести навыки анализа алгоритма и сокращения пространства поиска.

**Время работы.** На выполнение работы отводится четыре академических часа.

### Индивидуальные задания

1. Реализовать программу поиска счастливых билетов из материалов лекций и провести дальнейшее усовершенствование переборного алгоритма.
2. Решение диофантова уравнения  $4x + 5y = 0$  для значений переменных  $x$  и  $y$  из некоторого диапазона, значения переменных — целые числа.
3. Сгенерировать списки-палиндромы, состоящие из чисел из заданного диапазона.
4. Реализовать процедуру упорядочения списка методом перебора перестановок.
5. Решить задачу о выполнимости функции  $x_1x_2\bar{x}_3 \vee \bar{x}_1x_2x_3$ , подсчитать количество выполняющих наборов.
6. Решить задачу о восьми ферзях полным перебором.
7. Решить задачу о раскраске планарного графа (карты). Провести эксперименты со временем решения задачи в зависимости от количества доступных цветов для раскраски.

8. Известно, что пароль состоит из трех букв и цифр, в системе хранится хэш-номер как сумма ASCII-кодов пароля. Сгенерировать возможные пароли. Можно усложнить задачу: в системе хранится MD5-хэш паролей.
9. Задан набор из  $N$  слов, из которых требуется составить связный кроссворд. Слова в кроссворде должны располагаться либо вертикально, либо горизонтально, причем каждое слово, записанное по вертикали, должно пересекаться с каждым словом, записанным по горизонтали. Слова, записанные в одном направлении, отделяются друг от друга как минимум одним пустым рядом. Каждое слово в кроссворде должно встречаться в точности столько раз, сколько раз оно присутствует в наборе.
10. Заданы  $N$  различных точек плоскости и натуральное число  $M$ . Требуется найти максимальный по площади невырожденный  $M$ -угольник без самопересечений и самокасаний, вершинами которого являются некоторые из этих  $N$  точек.
11. Троллейбусы одного маршрута проходят через остановку каждые  $k$  ( $1 \leq k \leq 500$ ) минут. Известны времена прихода пассажиров на эту остановку. Если пассажир приходит на остановку в момент прихода троллейбуса, то он успевает уехать на нем.  
  
Напишите программу, которая бы определяла, во сколько должен пройти первый троллейбус (это время от 0 до  $k - 1$ ), чтобы: 1) суммарное время ожидания троллейбуса для всех пассажиров было минимально; 2) максимальное из времен ожидания троллейбуса было минимально.
12. Расшифровать ребус, полученный в результате замены одинаковых букв одинаковыми цифрами: БЛОК  $\times$  7 = СТЕНА.
13. Найти гамильтонов путь в графе.
14. Найти эйлеров путь в графе.
15. Расставить минимальное число белых коней, чтобы пробились все свободные позиции.

16. Расставить минимальное число белых ладей, чтобы пробивались все свободные позиции.
17. Расставить минимальное число белых ферзей, чтобы пробивались все свободные позиции.
18. Расставить минимальное число белых слонов, чтобы пробивались все свободные позиции.
19. Расставить максимальное число белых коней, чтобы они не били друг друга.
20. Расставить максимальное число белых ладей, чтобы они не били друг друга.
21. Расставить максимальное число белых ферзей, чтобы они не били друг друга.
22. Расставить максимальное число белых слонов, чтобы они не били друг друга.
23. Найти все кратчайшие маршруты коня между двумя заданными позициями.
24. Найти все кратчайшие маршруты ладьи между двумя заданными позициями.
25. Найти все кратчайшие маршруты ферзя между двумя заданными позициями.
26. Составить из костяшек набора домино все магические квадраты размера  $4 \times 4$ . Костяшки можно класть только горизонтально, костяшка занимает 2 позиции.
27. Расставить на клеточном поле всеми возможными способами фишки таким образом, чтобы в каждой линии (горизонтальной, вертикальной, диагональной) располагалось четное число фишек.
28. Имеется  $n$  деталей и  $m$  станков. Каждая деталь характеризуется временем обработки. Станок обрабатывает любую деталь сразу, все станки одинаковы. Определить порядок обработки деталей на станках, когда все детали будут обработаны за минимальное время.

## Методические указания к лабораторной работе

Выполняйте аналогично примеру в материалах лекции, начиная с формальной постановки задачи как задачи CSP. Для задачи о счастливых билетах вектор переменных  $\vec{V}$  — это набор переменных  $[A, B, C, D, E, F]$ , Области значений всех переменных в первоначальной постановке — числа из диапазона  $[0, 1, 2, \dots, 9]$ , поэтому достаточно было запрограммировать всего один генератор gen/1. В процессе совершенствования нами получены зависимости возможных диапазонов изменения значений от состояния назначения переменных  $A, B, C, D$ .

## Вопросы для самопроверки

1. В чем суть алгоритма «Британского музея»? Насколько он эффективен?
2. Как организуется генератор данных на проверку?
3. Приведите общую схему алгоритма решения диофантова уравнения.
4. Какова общая математическая постановка задач на удовлетворение ограничений?
5. Разработайте генератор чисел от 1 до 100 без использования списка значений.

## Заключение

Изучение логического программирования полезно для понимания того, что любая правильная<sup>1</sup> программа «устроена логично». Исследование правильности программы позволяет создавать логичный (т. к. изучение, как правило, всегда позволяет найти «дыры в логике» в плохой программе), короткий (чем длиннее код программы, тем длиннее его доказательство) и эффективный (изучение способствует пониманию того, как функционируют реализованные в программе алгоритмы) программный код.

В книге [9] дан пример классификации языков (точнее принципов) программирования (таблица 7.1) по соотношению признаков «условие» и «действие». Таким образом, вы только что ознакомились с новым принципом программирования, где программирование представляет собой одновременную обработку некоторого набора объектов, обладающих заданным свойством, в различных объектах данных.

Таблица 7.1

Классификация языков программирования

	Условие локально	Условие глобально
Действие локально	Структурное программирование	Объектно-ориентированное программирование + Event-driven или Process programming
Действие глобально	Программирование конечных автоматов	Сентенциальное программирование (Пролог, Рефал)

<sup>1</sup> Под *правильностью* программы подразумевается, что (а) будучи запущенной, программа рано или поздно остановится, и (б) что для всех корректных входных данных выходные данные будут тоже корректны. Ознакомиться с основными принципами **доказательства правильности** программ можно в книге [1].



## Рекомендуемая литература

1. Андерсон Р. Доказательство правильности программ : пер. с англ. / Р. Андерсон. – М. : Мир, 1982. – 168 с.: ил.
2. Братко И. Программирование на языке ПРОЛОГ для искусственного интеллекта : пер. с англ. / И. Братко. – М. : Мир, 1990. – 560 с.: ил.
3. Васильев С. Н. Интеллектуальное управление динамическими системами / С. Н. Васильев, А. К. Жерлов, Е. А. Федосов, Б. Е. Федунцов. – М. : Физматлит, 2000. – 352 с: ил.
4. Искусственный интеллект : в 3 кн. / под ред. Э. В. Попова. – М. : Радио и связь, 1990. – 464 с.: ил.
5. Лорьер Ж.-Л. Системы искусственного интеллекта : пер. с франц. / Ж.-Л. Лорьер. – М. : Мир, 1991. – 568 с.: ил.
6. Малпас Дж. Реляционный язык Пролог и его применение / Дж. Малпас. – М. : Наука, 1990. – 464 с.
7. Математический энциклопедический словарь / гл. ред. Ю. В. Прохоров. – М. : Сов. энциклопедия, 1988. – 847 с.
8. Непейвода Н. Н. Прикладная логика : учеб. пособие / Н. Н. Непейвода. – 2-е изд. – Новосибирск : Изд-во Новосибир. ун-та, 2000. – 521 с.: ил.
9. Непейвода Н. Н. Основания программирования / Н. Н. Непейвода, И. Н. Скопин. – Москва; Ижевск : Институт компьютерных исследований, 2003 – 880 с.: ил.
10. Рассел С. Искусственный интеллект: современный подход : пер. с англ. / С. Рассел, П. Новриг. 2-е изд. – М. : Изд. дом «Вильямс», 2006. – 1408 с.: ил.
11. ДС-грамматика [Электронный ресурс] // Wikipedia, The Free Encyclopedia: сайт. – URL:[https://en.wikipedia.org/wiki/Definite\\_clause\\_grammar](https://en.wikipedia.org/wiki/Definite_clause_grammar). (дата обращения: 28.11.2013).
12. The GNU Prolog web site [Электронный ресурс]: сайт. URL:<http://www.gprolog.org/>. (дата обращения: 28.11.2013).
13. SWI-Prolog's home [Электронный ресурс]: сайт. URL:<http://www.swi-prolog.org/>. (дата обращения: 28.11.2013).

*Учебное издание*

**Черкашин** Евгений Александрович

**Рекурсивно-логическое программирование**

ISBN 978-5-9624-0938-2

Редактор *Г. А. Борисова*

Верстка *Е. А. Черкашин*

Макет подготовлен при помощи системы Lua<sup>A</sup>T<sub>E</sub>X

Темплан 2013 г. Поз. 186

Подписано в печать 28.12.2013. Формат 60×90 1/16.  
Уч.-изд. л. 6,4. Усл. печ. л. 6,8. Тираж 100 экз. Заказ 170

Издательство ИГУ  
664003, г. Иркутск, бульвар Гагарина, 36

Е. А. Черкашин

# РЕКУРСИВНО-ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Е. А. Черкашин РЕКУРСИВНО-ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ