



*Томский межвузовский центр
дистанционного образования*

В. М. Зюзьков

ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

Томск - 2000

Министерство образования Российской Федерации

**ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)**

Кафедра автоматизированных систем управления (АСУ)

В. М. Зюзьков

ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

2000

Зюзьков В. М.

Логическое программирование: Учебное пособие. - Томск: Томский межвузовский центр дистанционного образования, 2000. - 62 с.

Предназначено для студентов, обучающихся на всех формах обучения с использованием дистанционных образовательных технологий.

© Зюзьков В. М., 2000
© Томский межвузовский центр
дистанционного образования, 2000

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ	7
1 МАТЕМАТИЧЕСКИЕ ОСНОВЫ ЛОГИЧЕСКОГО ПРОГРАММИРОВАНИЯ.....	10
1.1 История	10
1.2 Логический язык первого порядка	13
1.3 Формальные теории первого порядка.....	17
1.3.1 Определение формальной теории	17
1.3.2 Исчисление высказываний.....	20
1.3.3 Определение теорий первого порядка	21
1.4 Доказательство от противного.....	22
1.5 Задача об автоматическом доказательстве теорем	23
1.6 Предваренная нормальная форма.....	24
1.7 Сколемизация	27
1.8 Конъюнктивная нормальная форма	28
1.9 Сведение к дизъюнктам.....	29
1.10 Правило резолюции для исчисления высказываний	30
1.11 Унификация	31
1.12 Правило резолюции для исчисления предикатов	32
1.13 Алгоритм резолюций	32
1.14 Опровержение методом резолюций	34
2 ВВЕДЕНИЕ В ПРОЛОГ	38
2.1 Хорновская логическая программа	38
2.2 Сеанс работы с интерпретатором Пролога.....	41
2.3 Пример Пролог-программы: родственные отношения	44
2.3.1 Используем только факты.....	44
2.3.2 Используем факты и правила	50
2.3.3 Рекурсивные правила.....	53
2.4 Общие принципы поиска ответов на вопросы системой Пролог	54

2.5 Декларативная и процедурная семантика программ.....	58
2.5.1 Алгоритм работы интерпретатора Пролога	59
2.6 Синтаксис языка SWI-Prolog	60
2.7 Порядок предложений и целей	66
3 СТРУКТУРЫ ДАННЫХ	70
3.1 Предикат унификации	70
3.2 Арифметические выражения	72
3.2.1 Примеры программ с числами.....	76
3.2.2 Дифференцирование.....	79
3.3 Списки	81
3.3.1 Синтаксис и семантика списков.....	81
3.3.2 Некоторые предикаты для списков	83
3.4 Структуры	90
3.4.1 Выборка структурированной информации из базы данных	90
3.4.2 Рекурсивные структуры	93
3.5 Модификация синтаксиса (операторная запись)	96
4 УПРАВЛЕНИЕ ПОВТОРЕНИЕМ В ПРОЛОГЕ.....	101
4.1 Отсечение.....	101
4.1.1 Определение отсечения	101
4.1.2 Примеры программ с отсечением	106
4.2 Отрицание как неудача.....	110
4.3 Трудности с отсечением и отрицанием	113
4.4 Рекурсия	116
5 ВНЕЛОГИЧЕСКИЕ ПРЕДИКАТЫ ПРОЛОГА	121
5.1 Анализ и синтез термов	121
5.1.1 Проверка типа термов.....	121
5.1.2 Создание и декомпозиция термов.....	123
5.2 Ввод и вывод.....	127
5.3 Метапрограммирование	129
5.3.1 Эквивалентность программ и данных.....	129
5.3.2 Предположение об открытости мира.....	130

5.3.3 Программирование второго порядка	132
5.4 Операции с базой данных.....	134
5.4.1 Добавить в базу данных и удалить.....	134
5.4.2 Пример: база данных «Достопримечательности»	137
5.4.3 Пример: запоминающая функция.....	140
ЛИТЕРАТУРА.....	144
ПРИЛОЖЕНИЕ	145

ПРЕДИСЛОВИЕ

При изучении нового языка программирования требуется значительная затрата сил и времени. В этой связи необходимо обозначить причины, которые могут побудить программиста к изучению языка Пролог.

Любой язык программирования навязывает пользователям определенный взгляд на окружающий мир. Это проявляется в том, что программист, длительное время пользующийся некоторым языком и усвоивший соответствующий этому языку взгляд на мир, будет стремиться находить новые сферы применения для тех типов вычислений, к которым данный язык приспособлен, и будет избегать задач, для решения которых этот язык не годится.

По сравнению с процедурными языками язык Пролог освобождает пользователей от необходимости обдумывания правил вычисления значений и планирования действий, реализуемых в процессе выполнения программы. При должном пользовании языком Пролог взгляд программиста на мир может подняться до уровня логической спецификации подлежащей решению задачи.

Программа на Прологе больше является описанием того, что нужно вычислить («что есть истина»), чем того, как надо это сделать. В сущности, этот же подход реализуется и в функциональном программировании — лишь с тем уточнением, что речь в нем идет о свойствах функций. В связи с этим Дж. Робинсон предложил назвать такой стиль программирования *декларативным*.

Интуитивно любой программист отличит язык программирования высокого уровня от языка программирования низкого уровня. В чем состоит различие? Чем определяется уровень? Программирование представляет собой отображение в программах объектов, понятий и явлений природной области задачи. Чем более адекватно можно выполнить это отображение, тем выше уровень языка программирования.

Декларативное программирование требует высокого уровня абстрагирования. Но повышение уровня абстрагирования — необходимое требование для программиста. Эдсгер Дейкстра (Edsger Dijkstra) [5] подчеркивал, что программист должен обладать умением абстрагировать: «Язык программирования — это лишь средство описания абстрактных конструкций. Программист

должен иметь способность полностью абстрагироваться от несущественных деталей, думая на нескольких уровнях абстракции одновременно».

Декларативные языки при классификации по степени абстракции от аппаратуры относятся к языкам *сверхвысокого уровня*. Команды исполняются на полностью абстрактной машине, полностью скрыт доступ к памяти, и возможно скрыть поток управления.

Наиболее убедительный аргумент в пользу необходимости изучения языка Пролог заключается в том, что данный язык позволяет работать специалисту в актуальных областях информатики на высоком концептуальном уровне.

Остановимся на вопросе — как должно происходить обучение программированию вообще. Методологический подход Г.С. Цейтина [7, с. 46] исходит из того, что целью курса программирования следует считать переход от непонимания программирования к его пониманию. Тогда для формирования навыков программирования нужно изучать их структуру у подготовленных программистов и определить содержание курса в соответствии с выделенными элементами. Предполагая, что программирование — это переход от знания о задаче, выраженного в обычной, не программной форме, к выражению этого знания (точнее, части его) в форме программы, предлагается выделить три элемента программирования по такому преобразованию знания.

- Первый элемент — *переход от знания в математической форме* к знанию в форме программы. Ему соответствует программирование математических формул, программирование рекуррентно решаемых задач посредством рекурсивных процедур (во всех этих случаях не требуется даже присваивания).
- Второй элемент — *моделирующий*, включает построение последовательности действий в программе по образцу последовательности действий в моделируемой реальной системе или моделируемом поведении человека, решающего задачу стандартным способом.

- Третий элемент программирования — *преобразования программ*. Они включают сборку сложной программы из элементарных программ и эквивалентные преобразования.

Оказывается, что сама парадигма декларативного программирования способствует при обучении программированию следовать методологическому подходу Цейтина.

...Но труды учения нужны не только для того, чтобы добыть известный информационный капитал. Они играют и другую роль, не зависящую от природы этого капитала: они пробуждают страсть к соревнованию, учат преодолевать препятствия, укрепляют «сопротивление стрессам» и таким образом формируют структуру личности.

Станислав Лем. Сумма технологий

1 МАТЕМАТИЧЕСКИЕ ОСНОВЫ ЛОГИЧЕСКОГО ПРОГРАММИРОВАНИЯ

1.1 История

По миру распространяется огромное число лживых историй, и хуже всего то, что добрая половина из них — правда.

Уинстон Черчилль

Логическое программирование является, пожалуй, наиболее впечатляющим примером применения идей и методов математической логики (точнее, одного из ее разделов — теории логического вывода) в программировании.

Идея использования языка логики предикатов первого порядка в качестве языка программирования возникла еще в 60-е годы, когда создавались многочисленные системы автоматического доказательства теорем и основанные на них вопросно-ответные системы. Суть этой идеи заключается в том, чтобы программист не указывал машине последовательность шагов, ведущих к решению задачи, как это делается во всех процедурных языках программирования, а описывал на логическом языке свойства интересующей его области, иначе говоря, описывал мир своей задачи. Другие свойства и удовлетворяющие им объекты машина находила бы сама путем построения логического вывода.

Первые компьютерные реализации систем автоматического доказательства теорем появились в конце 50-х годов, а в 1965 г.

Дж. Робинсон (J.A. Robinson) предложил метод резолюций [1], который и по сей день лежит в основе большинства систем поиска логического вывода.

Робинсон пришел к заключению, что правила вывода, которые следует применять при автоматизации процесса доказательства с помощью компьютера, не обязательно должны совпадать с правилами вывода, используемыми человеком. Он обнаружил, что общепринятые правила вывода, например правило *modus ponens*, специально сделаны «слабыми», чтобы человек мог интуитивно проследить за каждым шагом процедуры доказательства. Правило резолюции более сильное, оно трудно поддается восприятию человеком, но эффективно реализуется на компьютере.

После открытия метода резолюций он, по предложению Грина (Cordell Green), вскоре был использован в качестве основы нового языка программирования. К концу 60-х годов выявились принципиальные трудности, препятствующие широкому применению таких систем. Главная проблема заключается в практической неэффективности известных методов построения логического вывода. Стремление обойти эту преграду привело к созданию различных линейных стратегий метода резолюций, которые, в сущности, являются прообразами современных интерпретаторов языка Пролог. В 1971 г. один из специалистов в области автоматического доказательства теорем Роберт Ковальски (Robert Kowalski) из Эдинбурга прочитал несколько лекций по автоматическому доказательству теорем в лаборатории Искусственного интеллекта Марсельского университета. Работающий там Ален Колмероэ (Alain Colmerauer) и его коллеги вскоре поняли, каким образом можно использовать резолюцию в качестве основы нового языка программирования. Так, в 1972 году родился язык Prolog («programming in logic» — программирование в терминах логики, по-русски, Пролог), быстро завоевавший популярность во всем мире.

Популяризации языка Пролог во многом способствовала эффективная реализация этого языка в середине 1970-х годов Дэвидом Д.Г. Уорреном (David D. H. Warren) из Эдинбурга. К числу новейших достижений в этой области относятся средства про-

граммирования на основе логики ограничений (Constraint Logic Programming — CLP), которые обычно реализуются в составе системы Пролог. Средства CLP показали себя на практике как исключительно гибкий инструмент для решения задач составления расписаний и планирования материально-технического снабжения. В 1996 году был опубликован официальный стандарт ISO языка Prolog.

Следуя [2], отметим наиболее заметные тенденции в истории развития языка Пролог. Этот язык быстро приобрел популярность в Европе как инструмент практического программирования. Логическое программирование пережило пик популярности в середине 80-х годов 20 века, когда оно было положено японцами в основу проекта разработки программного и аппаратного обеспечения вычислительных систем пятого поколения [9]. С другой стороны, в США этот язык в целом был принят с небольшим опозданием в связи с некоторыми историческими причинами. Одна из них состояла в том, что Соединенные Штаты вначале познакомились с языком Microplanner, который был также близок к идее логического программирования, но неэффективно реализован. Определенная доля низкой популярности Пролога в этой стране объясняется также реакцией на существующую «ортодоксальную школу» логического программирования, представители которой настаивали на использовании чистой логики и требовали, чтобы логический подход не был «запятнан» практическими средствами, не относящимися к логике. В прошлом это привело к широкому распространению неверных взглядов на язык Пролог. Например, некоторые считали, что на этом языке можно программировать только рассуждения с выводом от целей к фактам. Но истина заключается в том, что Пролог — универсальный язык программирования и на нем может быть реализован любой алгоритм. Далекая от реальности позиция «ортодоксальной школы» была преодолена практиками языка Пролог, которые приняли более прагматический подход, воспользовавшись плодотворным объединением нового декларативного подхода с традиционным процедурным.

1.2 Логический язык первого порядка

Терпеть не могу логики. Она всегда банальна и нередко убедительна.

Оскар Уайльд

Изложим неформально математические основы логического программирования. Для этого понадобятся некоторые знания из курса математической логики в пределах технического университета.

Любое математическое утверждение в конечном счете говорит о предметах (объектах). Каждая математическая теория имеет свою *предметную область*, или *универсум*, — совокупность всех предметов, которые она изучает.

Например, универсумом теории чисел является множество натуральных чисел, а ее объектами — сами натуральные числа.

Математическая теория может иметь несколько универсумов. В этом случае теория является многосортной, объекты делятся на *типы*, или *сорта*, и для каждого сорта задается свой универсум. Примерами могут служить современные языки программирования, или математический анализ — два универсума: универсум чисел и универсум функций.

Простейшие из выражений, обозначающих предметы, — *константы*, т.е. имена конкретных объектов. Например, константами служат числа (2, -5 , π и т.д.). Считается, что для каждой константы однозначно задан предмет, который она обозначает. Далее для каждой константы четко указывается сорт, которому она принадлежит. Аналогией этого могут служить описания типизированных констант в языках программирования.

Столь же просты с виду и *переменные*, например x , y ,... Но для переменной неизвестен предмет, который она обозначает; в принципе она может обозначать какой угодно предмет из нашего универсума. Например, если наш универсум — люди, то x может обозначать в данный момент любого конкретного человека. Чтобы наши рассуждения не стали ошибочными, нужно следить, чтобы однажды выбранное значение x далее внутри данного рассуждения не изменялось, как говорят, оно должно быть *фиксированным* (обратите внимание на противоположное понимание пе-

ременной в программировании). Для того чтобы у нас был неограниченный запас имен переменных, часто пользуются индексами, например x_1 .

Более сложные выражения образуются применением символов операций к более простым. Операция, соответствующая символу, применяется к предметам и в результате дает тоже предмет. Например, символу \times сопоставляется операция над числами, дающая по двум числам их произведение. В общем случае n -местную операцию f , примененную к выражениям t_1, \dots, t_n , будем обозначать $f(t_1, \dots, t_n)$, такую форму записи называют *функциональной*.

Выражение, обозначающее предмет, называется *термом*. Операции называются еще *функциональными символами*, или просто *функциями*.

Чтобы образовать высказывание из предметов, нужно соединить их отношением; *n -местное отношение* — операция, сопоставляющая n предметам высказывание. Например, двуместное отношение « $=$ » сопоставляет двум числам x и y высказывание « $x = y$ », в частности $2 = 2$ — истинное высказывание, а $2 = 5$ — ложное высказывание. Одноместное отношение «... — положительное число» для числа 5 является истинным высказыванием «5 — положительное число». В «теории человеческих отношений» двуместное отношение «любить» сопоставляет паре (Ромео, Джульетта) истинное высказывание, а паре (Демон, Тамара) — ложное высказывание.

В математике чаще всего встречаются одноместные и двуместные (бинарные) отношения. Бинарные отношения обычно записываются между своими аргументами, например $4 < 7$, $x^2 + 2x + 1 > 0$ и т.д. Одноместные отношения в математике часто записываются при помощи символа \in и символа для множества объектов, обладающих данным свойством. Например, утверждение « π — действительное число» записывается в виде $\pi \in \mathbf{R}$, где \mathbf{R} обозначает множество действительных чисел.

В логике для единообразия мы пользуемся записью

$$P(t_1, \dots, t_n),$$

чтобы обозначить высказывание, образованное применением n -местного отношения P к предметам t_1, \dots, t_n . Символ P , изобра-

жающий отношение, называется *предикатом*. «Предикат» и «отношение» соотносятся как имя и предмет, им обозначаемый. Но в математике эти два понятия употребляются почти как синонимы. В логических материалах пользуются строгим термином «предикат», а в конкретных приложениях, когда это вошло в математическую традицию, используют и слово «отношение» (например, говорить об отношении « $>$ » в формуле $a > b$).

В такой записи $2 = 4$ выглядит следующим образом:

$$=(2, 4).$$

Элементарные (или *атомарные*) *формулы* имеют вид $P(t_1, \dots, t_n)$, где P — n -местный предикат, t_1, \dots, t_n — термы. В обычной математике элементарные формулы называются просто *формулами*.

Сложные формулы строятся из элементарных. Задавая язык конкретной математической теории, непосредственно определяют именно элементарные формулы и их смысл.

Для того чтобы задать элементарные формулы, необходимо определить предикаты, используемые в нашей теории, и ее термы. А чтобы задать термы, нужно определить сорта объектов, константы и операции. В совокупности предикаты, сорта, константы, операции составляют *словарь* (или *сигнатуру*) теории.

Интерпретация

Задав словарь теории, необходимо *проинтерпретировать* все понятия, перечисленные в нем. При этом константам сопоставляются конкретные объекты, задаются правила вычисления функций, сопоставленных операциям, и правила, по которым определяются логические значения предикатов. После интерпретации элементарные формулы, не содержащие переменных, оказываются либо истинны, либо ложны, а формулы, содержащие переменные, становятся истинными либо ложными после задания (фиксации) значений переменных.

Выражения, с помощью которых записываются высказывания в нашем формальном языке, называются *логическими формулами*. С чисто формальной точки зрения предикаты (отношения) можно рассматривать как функции, сопоставляющие своим аргу-

ментам истинностные значения, т.е. функции, принимающие всего два значения: **истина** и **ложь**.

Как только задана интерпретация и фиксированы значения всех встречающихся в элементарной формуле переменных, становится известно и логическое значение элементарной формулы.

Мы тогда можем определить истинностные значения и всех более сложных формул, так как значения их элементарных частей уже заданы. Но для этого нужно знать, какими способами более сложные формулы строятся из более простых. Для образования новых формул из имеющихся используются логические связки. Логические связки применяются к высказываниям и в результате дают высказывания.

Общеприняты следующие логические связки:

Название	Обозначение	Каким предложением обычно передается в русском языке
конъюнкция	$A \& B$	« A и B »
дизъюнкция	$A \vee B$	« A или B или оба вместе»
отрицание	$\neg A$	«не A »
импликация	$A \supset B$	« A влечет B »
квантор общности	$\forall x A(x)$	«для всех x верно $A(x)$ »
квантором существования	$\exists x A(x)$	«существует такое x , что $A(x)$ »

Выразительные средства языка, который мы описали, принципиально ограничены в одном важном отношении: нет возможности говорить о произвольных свойствах объектов теории, т.е. о произвольных подмножествах множества всех объектов. Синтаксически это отражается в запрете формулировать выражения, скажем, вида $\forall P(P(x))$, где P — предикат. Предикаты обозначают фиксированные, а не переменные свойства. Поэтому данный язык называется *языком логики предикатов первого порядка* (или просто *языком первого порядка*).

Используя введенный язык, можно теперь различные утверждения записать в виде формул. Например: «Прапорщики

любят порядок, но не только они». В качестве универсума берем множество людей и на нем определяем два одноместных предиката: $O(x) \equiv$ « x любит порядок» и $P(x) \equiv$ « x — прапорщик». Тогда искомая формула имеет вид

$$\forall x(P(x) \supset O(x)) \& \exists x(\neg P(x) \& O(x)).$$

1.3 Формальные теории первого порядка

Формальная теория представляет собой множество чисто абстрактных объектов (не связанных с внешним миром), в которой представлены правила оперирования множеством символов в чисто синтаксической трактовке без учета смыслового содержания (или семантики).

Исторически понятие формальной теории было разработано в период интенсивных исследований в области оснований математики для формализации собственно логики и теории доказательства. Сейчас этот аппарат широко используется при создании специальных исчислений для решения конкретных прикладных задач. Одним из таких приложений является логическое программирование.

Формальную теорию иногда называют *аксиоматикой* или *формальной аксиоматической теорией*. Родоначальником аксиоматических теорий можно считать «Начала» Евклида.

1.3.1 Определение формальной теории

Формальная теория T считается определенной, если:

- выделено некоторое множество формул, называемых *аксиомами* теории T ;
- имеется конечное множество $\{R_1, R_2, \dots, R_m\}$ отношений между формулами, называемых *правилами вывода*. Правила вывода позволяют получать из некоторого конечного множества формул другое множество формул.

Множество аксиом B может быть конечным или бесконечным. Если множество аксиом бесконечно, то, как правило, оно задается с помощью конечного множества *схем аксиом* и правил

порождения конкретных аксиом из схемы аксиом. Обычно аксиомы делятся на два вида: *логические* аксиомы (общие для целого класса формальных теорий) и *нелогические* (или *собственные*) аксиомы (определяющие специфику и содержание конкретной теории).

Пусть A_1, A_2, \dots, A_n, A — формулы теории T . Если существует такое правило вывода R , что $\langle A_1, A_2, \dots, A_n, A \rangle \in R$, то говорят, что формула A *непосредственно выводима* из формул A_1, A_2, \dots, A_n по правилу вывода R . Обычно этот факт записывают следующим образом:

$$\frac{A_1, A_2, \dots, A_n}{A} R,$$

где формулы A_1, A_2, \dots, A_n называются *посылками*, а формула A — *заключением*.

Выводом формулы A из множества формул Γ в теории T называется такая последовательность F_1, F_2, \dots, F_k , что $A = F_k$, где любая формула F_i ($i < k$) является либо аксиомой, либо $F_i \in \Gamma$, либо непосредственно выводима из ранее полученных формул F_{j_1}, \dots, F_{j_n} ($j_1, \dots, j_n < i$). Если в теории T существует вывод формулы A из множества формул Γ , то это записывается следующим образом:

$$\Gamma \vdash_T A,$$

где формулы из Γ называются *гипотезами* вывода. Если теория T подразумевается, то её обозначение обычно опускают.

Если множество Γ конечно: $\Gamma = \{B_1, B_2, \dots, B_n\}$, то вместо $\{B_1, B_2, \dots, B_n\} \vdash A$ пишут $B_1, B_2, \dots, B_n \vdash A$. Если Γ есть пустое множество \emptyset , то A называют теоремой (или *доказуемой* формулой) и в этом случае используют сокращенную запись $\vdash A$ (« A есть теорема»).

Понятие формальности можно определить в терминах теории алгоритмов: теорию T можно считать формальной, если построен алгоритм (механически применяемая процедура вычисления) для проверки правильности рассуждений с точки зрения принципов теории T . Это значит, что если некто предлагает математический текст, являющийся, по его мнению, доказательством некоторой теоремы в теории T , то, механически применяя

алгоритм, мы можем проверить, действительно ли предложенный текст соответствует стандартам правильности, принятым в T . Таким образом, стандарт правильности рассуждений для теории T определен настолько точно, что проверку его соблюдения можно передать вычислительной машине (следует помнить, что речь идет о **проверке правильности** готовых доказательств, а не об их поиске!). Если проверку правильности доказательств в какой-либо теории нельзя передать вычислительной машине и она доступна в полной мере только человеку, значит, еще не все принципы теории аксиоматизированы (то, что мы не умеем передать машине, остается в нашей интуиции и «оттуда» регулирует наши рассуждения).

Формулы теории имеют смысл только тогда, когда имеется какая-нибудь интерпретация входящих в них символов.

Если T — теория, то можно рассматривать различные интерпретации этой теории. Чтобы определить *интерпретацию*, мы должны задать:

- универсум M , называемый *областью интерпретации*;
- соответствие, относящее каждому n -местному предикату некоторое n -местное отношение в M , каждой функциональной букве, требующей n аргументов, — некоторую функцию $M^n \rightarrow M$ и каждой константе — некоторый элемент из M ;
- предметные переменные мыслятся пробегающими область интерпретации M ;
- логическим связкам придается их обычный смысл.

Для заданной интерпретации всякая формула P без свободных переменных представляет собой высказывание, которое истинно или ложно. Если высказывание является истинным, то говорят, что формула P *выполняется* в данной интерпретации.

Всякая формула со свободными переменными выражает некоторое отношение на области интерпретации; это отношение может быть выполнено (истинно) для одних значений переменных из области интерпретации и не выполнено (ложно) для других.

Формализация задается не только синтаксисом и семантикой формального языка (эти компоненты как раз чаще всего берутся традиционными, из хорошо известного крайне ограниченного набора), но и множеством утверждений, которые считаются истинными. Именно эта формулировка базисных свойств, аксиом, описывающих некоторую предметную область, обычно рассматривается как математическое описание объектов. Таким образом, практически нас интересуют не все интерпретации данной теории, а лишь те из них, на которых выполнены аксиомы.

Одним из первых вопросов, которые возникают при задании формальной теории, является вопрос о том, возможно ли, рассматривая какую-нибудь формулу формальной теории, определить, является ли она доказуемой или нет. Другими словами, речь идет о том, чтобы определить, является ли данная формула теоремой или *не-теоремой* и как это доказать.

- Формальная теория T называется **разрешимой**, если существует алгоритм, который для любой формулы теории определяет, является ли это формула теоремой теории.
- Формальная теория T называется **полуразрешимой**, если существует алгоритм, который для любой формулы P теории выдает ответ «да», если P является теоремой теории, и выдает «нет» или, может быть, не выдает никакого ответа, если P не является теоремой (то есть алгоритм применим не ко всем формулам).

1.3.2 Исчисление высказываний

Исчислением высказываний называется формальная теория с языком логики высказываний, со схемами аксиом

$A1) A \supset (B \supset A);$

$A2) (A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C));$

$A3) (\neg B \supset \neg A) \supset ((\neg B \supset A) \supset B);$

и правилом вывода MP (*Modus Ponens* — обычно переводится как *правило отделения*):

$$\frac{A, A \supset B}{B} \quad MP$$

Здесь A , B и C — *любые* формулы. Таким образом, множество аксиом исчисления высказываний бесконечно, хотя задано тремя схемами аксиом. Множество правил вывода также бесконечно, хотя оно задано только одной схемой.

Интерпретация формул исчисления высказываний проста — область интерпретации состоит из двух значений «истина» и «ложь»; поэтому пропозициональная переменная принимает только значения И и Л и интерпретация составной формулы вычисляется по известным законам с помощью логических операций над истинностными значениями. Поскольку любая формула содержит только конечное число пропозициональных переменных, то формула обладает только конечным числом различных интерпретаций. Следовательно, исчисление высказываний является, очевидно, разрешимой формальной теорией.

Напомним, что формулы исчисления высказываний являются теоремами тогда и только тогда, когда они являются тавтологиями (т.е. истинными во всех интерпретациях).

1.3.3 Определение теорий первого порядка

Формальная теория называется *теорией первого порядка*, если она имеет следующие аксиомы и правила вывода.

Аксиомы теории первого порядка T разбиваются на два класса: логические аксиомы и собственные (или нелогические) аксиомы.

Логические аксиомы: каковы бы ни были формулы A , B и C теории T , следующие формулы являются логическими аксиомами теории T .

A_1 . $A \supset (B \supset A)$;

A_2 . $(A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))$;

A_3 . $(\neg B \supset \neg A) \supset ((\neg B \supset A) \supset B)$;

A_4 . $\forall x A(x) \supset A(x)$;

A_5 . $\forall x (A \supset B(x)) \supset (A \supset \forall x B(x))$, где A не содержит свободных вхождений переменной x .

Собственные аксиомы: таковые не могут быть сформулированы в общем случае, ибо меняются от теории к теории. Правилами вывода во всякой теории первого порядка являются

1. *Modus ponens*: $\frac{A, A \supset B}{B} \text{ MP}$
2. Правило обобщения: $\frac{A(x)}{\forall x A(x)} \text{ Gen}$

Теория первого порядка, которая не содержит собственных аксиом, называется *исчислением предикатов первого порядка*. *Чистым исчислением предикатов* называется исчисление предикатов первого порядка, не содержащее предметных констант и функторов.

Логические аксиомы выбраны таким образом, что множество логических следствий аксиом теории в точности совпадает с множеством теорем теории. В частности, для исчисления предикатов первого порядка множество его теорем совпадает с множеством логически общезначимых формул.

Теорема 1. (Теорема Чёрча о неразрешимости исчисления предикатов, 1936). Не существует алгоритма, который для любой формулы исчисления предикатов первого порядка устанавливает, теорема она или нет.

1.4 Доказательство от противного

Частным случаем косвенных методов доказательства является приведение к противоречию (от противного). Метод доказательства основывается на следующем утверждении.

Если $\Gamma, \neg S \vdash F$, где F – любое противоречие (тождественно ложная формула), то $\Gamma \vdash S$.

В этом методе используются следующие равносильности:

$$A \supset B \equiv \neg (A \supset B) \supset (C \& \neg C) \equiv (A \& \neg B) \supset (C \& \neg C),$$

$$A \supset B \equiv (A \& \neg B) \supset \neg A,$$

$$A \supset B \equiv (A \& \neg B) \supset B.$$

Используя вторую из приведенных равносильностей для доказательства $A \supset B$, мы допускаем одновременно A и $\neg B$, т.е. предполагаем, что заключение ложно:

$$\neg (A \supset B) \equiv \neg (\neg A \vee B) \equiv A \& \neg B.$$

Теперь мы можем двигаться и вперед от A , и назад от $\neg B$. Если B выводимо из A , то, допустив A , мы доказали бы B . Поэтому, допустив $\neg B$, мы получим противоречие. Если же мы выведем $\neg A$ из $\neg B$, то тем самым получим противоречие с A . В общем случае мы можем действовать с обоих концов, выводя некоторое предложение C , двигаясь вперед, и его отрицание $\neg C$, двигаясь назад. В случае удачи это доказывает, что наши посылки *несовместимы* или *противоречивы*. Отсюда мы выводим, что дополнительная посылка $A \& \neg B$ должна быть ложна, а, значит, противоположное ей утверждение $A \supset B$ — истинно. Метод доказательства от противного — один из самых лучших инструментов математика. «Это гораздо более «хитроумный» гамбит, чем любой шахматный гамбит: шахматист может пожертвовать пешку или даже фигуру, но математик жертвует *партию*» (Г. Харди).

1.5 Задача об автоматическом доказательстве теорем

Алгоритм, который проверяет отношение

$$\Gamma \mid\!-\!_T S$$

для формулы S , множества формул Γ , и теории T , называется алгоритмом *автоматического доказательства теорем*. В общем случае такой алгоритм невозможен, т.е. не существует алгоритма, который для любых S , Γ и T выдавал бы ответ «Да», если $\Gamma \mid\!-\!_T S$, и ответ «Нет», если неверно $\Gamma \mid\!-\!_T S$. Более того, известно, что нельзя построить алгоритм автоматического доказательства теорем даже для большинства конкретных достаточно сложных формальных теорий T . В некоторых случаях удастся построить алгоритм автоматического доказательства теорем, который применим не ко всем формулам теории (т.е. частичный алгоритм).

Для некоторых простых формальных теорий (например, исчисления высказываний) и некоторых простых классов формул (например, теорий первого порядка с одним одноместным предикатом) алгоритмы автоматического доказательства теорем известны.

Пример. Поскольку для исчисления высказываний известно, что теоремами являются тавтологии, можно воспользоваться простым методом проверки общезначимости формулы с помо-

щью таблиц истинности. А именно достаточно вычислить истинностное значение формулы при всех возможных интерпретациях (их конечное число). Если во всех случаях получится значение И, то проверяемая формула — тавтология, и, следовательно, является теоремой исчисления высказывания. Если же хотя бы в одном случае получится значение Л, то проверяемая формула не является тавтологией и, следовательно, не является теоремой теории высказываний.

Приведенный выше пример является алгоритмом автоматического доказательства теорем в теории исчисления высказываний, хотя и не является алгоритмом автоматического поиска вывода теорем из аксиом теории исчисления высказываний.

Для любой теории первого порядка T , любой формулы S и множества формул Γ теории T метод резолюций выдает ответ «Да», если $\Gamma \vdash_T S$, и выдает ответ «Нет» или не выдает никакого ответа (т.е. заикливается), если неверно $\Gamma \vdash_T S$.

В основе метода резолюций лежит идея «доказательства от противного»: пытаемся доказать $\Gamma, \neg S \vdash F$, где F — любое противоречие. Отсюда будет следовать $\Gamma \vdash_T S$.

Пустая формула не имеет никакого значения ни в какой интерпретации, в частности, не является истинной ни в какой интерпретации и, по определению, является противоречием. В качестве формулы F при доказательстве от противного по методу резолюций принято использовать пустую формулу, которая обозначается \square .

1.6 Предваренная нормальная форма

Приведение формулы к виду, пригодному для применения метода резолюций, требует рассмотрения некоторых нормальных форм формул.

Формула $Q_1x_1 \dots Q_nx_n A$, где символ Q_i означает либо \forall , либо \exists , x_i и x_j различны для $i \neq j$ и A — не содержит кванторов, называется формулой в **предваренной нормальной форме**. (Сюда включается и случай $n = 0$, когда вообще нет никаких кванторов.)

Без ограничения общности можно потребовать, чтобы квантифицированы могли быть только переменные, имеющие свободное вхождение. Действительно, по правилам интерпретации если формула A не содержит вхождений переменной x , то формулы A , $\exists xA$ и $\forall xA$ имеют одно и то же значение истинности при всех интерпретациях.

Интерес к предваренным формам связан со следующей теоремой, справедливой для всякой теории первого порядка.

Теорема 2. Существует алгоритм, преобразующий всякую формулу A к такой формуле B в предваренной нормальной форме, что $A=B$.

Доказательство. Алгоритм получения предваренной формулы очень прост. Этапы таковы:

- **Исключения связок эквивалентности и импликации** по правилам:

$$A \sim B \equiv (A \supset B) \& (B \supset A) \equiv (A \& B) \vee (\neg A \& \neg B);$$

$$A \supset B \equiv \neg A \vee B \equiv \neg (A \& \neg B).$$

- **Переименование** (если необходимо) связанные переменные таким образом, чтобы никакая переменная не имела бы одновременно свободных и связанных вхождений. Это условие требуется не только для рассматриваемой формулы, но также для всех ее подформул.
- **Разделение связанных переменных.** Логически эквивалентное преобразование: $Q_1x A(\dots Q_2x B(\dots x \dots) \dots) = Q_1x A(\dots Q_2y B(\dots y \dots) \dots)$, где Q_1 и Q_2 — любые кванторы и y — любая переменная, не входящая в формулу $B(\dots x \dots)$.
- **Удаление тех квантификаций**, область действия которых не содержит вхождений квантифицированной переменной. Такие квантификации в действительности не нужны.
- **Протаскивание отрицаний.** Преобразовать все вхождения отрицания в стоящие непосредственно перед элементарными подформулами в соответствии со следующими правилами:

$$\neg \forall x A = \exists x \neg A;$$

$$\neg \exists x A = \forall x \neg A;$$

$$\neg \neg A = A;$$

$$\neg (A \vee B) = \neg A \& \neg B;$$

$$\neg(A \& B) = \neg A \vee \neg B.$$

- **Перемещение кванторов.** С помощью логически эквивалентных преобразований перемещаем все квантификации в начало формулы. Для этого используется следующий набор правил преобразования, которые справедливы во всякой теории первого порядка. Здесь мы приведем эти правила для конъюнкции. Парные правила для дизъюнкции выглядят аналогично.

$$(\forall x A \& \forall x B) = \forall x (A \& B).$$

$$(\forall x A \& B) = \forall x (A \& B), \text{ если } B \text{ не содержит } x.$$

$$(A \& \forall x B) = \forall x (A \& B), \text{ если } A \text{ не содержит } x.$$

$$(\exists x A \& B) = \exists x (A \& B), \text{ если } B \text{ не содержит } x.$$

$$(A \& \exists x B) = \exists x (A \& B), \text{ если } A \text{ не содержит } x.$$

Эти правила позволяют в каждой формуле постепенно передвигать все кванторы влево.

Для полноты этого набора правил необходимо добавить возможность переименования некоторых связанных переменных. Например, формула $\exists x A(x) \& \forall x B(x)$ будет сначала преобразована в $\exists x A(x) \& \forall y B(y)$ (перед применением правил преобразования). Заметим также, что для упрощения формул в любой момент можно применять свойства коммутативности, ассоциативности и идемпотентности связок $\&$ и \vee . Итак, теорема конструктивно доказана.

Пример. Найдем предваренную форму, эквивалентную формуле

$$\forall x(P(x) \& \forall y \exists x(\neg Q(x,y) \supset \forall z R(a,x,y))).$$

Этапы этого поиска последовательно перечислены ниже.

Исключение связки импликации:

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

Разделение связанных переменных:

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

Удаление бесполезной квантификации:

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

Протаскивание отрицаний:

$$\forall x(P(x) \& \forall y \exists u(Q(u,y) \vee R(a,u,y))).$$

Перемещение кванторов:

$$\forall x \forall y (P(x) \& \exists u(Q(u,y) \vee R(a,u,y))),$$

$$\forall x \forall y \exists u (P(x) \& (Q(u,y) \vee R(a,u,y))).$$

Замечание. Одна формула может допускать много эквивалентных предваренных форм. Вид полученного результата зависит от порядка применения правил, а также от произвола при переименовании.

1.7 Сколемизация

Рассмотрим один из математических результатов, послуживших идейной основой метода резолюций. Интуитивно сочетание кванторов $\forall x \exists y$ может пониматься как утверждение о существовании функции, строящей y по x . Рассмотрим несколько примеров. Если замкнутая формула $\exists y P(y)$ выполнима в некоторой интерпретации, то существует некоторая предметная константа c из универсума, для которой формула $P(c)$ истинна. Другой случай. Пусть, например, имеется формула P с двумя параметрами x и y . Тогда замкнутая формула $\forall x \exists y P(x,y)$ выполнима тогда и только тогда, когда выполнима формула $\forall x P(x, f(y))$, где f — новый одноместный функциональный символ.

Аналогичное преобразование выполнимо и для произвольных предваренных формул. Например, формула

$$\forall x \forall y \exists z \forall u \exists v P(x,y,z,u,v)$$

выполнима тогда и только тогда, когда выполнима формула

$$\forall x \forall y \forall u P(x,y, f(x,y), u, g(x,y,u))$$

(здесь f и g — функциональные символы, не встречающиеся в формуле P).

Элиминация кванторов существования (сколемизация)

Сколемовской формой называется предваренная нормальная форма, не содержащая кванторы существования.

Сколемизация осуществляется преобразованиями:

$$\exists x_1 Q_2 x_2 \dots Q_n x_n A(x_1, x_2, \dots, x_n) \rightarrow Q_2 x_2 \dots Q_n x_n A(a, x_2, \dots, x_n);$$

$$\forall x_1 \dots \forall x_{i-1} \exists x_i Q_{i+1} x_{i+1} \dots Q_n x_n A(x_1, \dots, x_i, \dots, x_n) \rightarrow$$

$$\forall x_1 \dots \forall x_{i-1} Q_{i+1} x_{i+1} \dots Q_n x_n A(x_1, \dots, f(x_1, \dots, x_{i-1}), \dots, x_n),$$

где a — новая предметная константа, f — новый функтор, а Q_2, \dots, Q_n — любые кванторы, символ \rightarrow показывает результат преобразования.

Функции для замены кванторов существования впервые стал использовать Т. Сколем.

Теорема 3. Пусть даны замкнутая формула A и формула B — результат сколемизации A . Тогда формулы A и B одновременно выполнимы или невыполнимы.

1.8 Конъюнктивная нормальная форма

Литерал — это элементарная формула или отрицание элементарной формулы.

Дизъюнкт — это литерал или дизъюнкция конечного числа литералов.

Конъюнктивная нормальная форма — это формула, которая является дизъюнктом или конъюнкцией конечного числа дизъюнктов.

Теорема 4. Для любой формулы A , не содержащей кванторы, существует формула B , являющаяся конъюнктивной нормальной формой (КНФ), и B логически эквивалентна A .

Доказательство. Укажем алгоритм построения формулы B .

- Исключаем связки эквивалентности и импликации по правилам:

$$X \sim Y = (X \supset Y) \& (Y \supset X);$$

$$X \supset Y = \neg X \vee Y.$$

- **Протаскивание отрицаний.** Преобразуем все вхождения отрицания в стоящие непосредственно перед элементарными подформулами в соответствии со следующими правилами:

$$\neg \neg X = X;$$

$$\neg (X \vee Y) = \neg X \& \neg Y;$$

$$\neg (X \& Y) = \neg X \vee \neg Y.$$

- Преобразуем полученную формулу, используя правило дистрибутивности \vee относительно $\&$.

Пример. Приведем к КНФ формулу $(X \& Y) \sim (\neg X \& Z)$.

- После первого этапа получаем

$$(X \& Y) \sim (\neg X \& Z) = (X \& Y \supset \neg X \& Z) \& (\neg X \& Z \supset X \& Y) = \\ = (\neg(X \& Y) \vee (\neg X \& Z)) \& (\neg(\neg X \& Z) \vee (X \& Y)).$$

- После второго этапа получаем

$$(\neg(X \& Y) \vee (\neg X \& Z)) \& (\neg(\neg X \& Z) \vee (X \& Y)) = ((\neg X \vee \neg Y) \vee (\neg X \& Z)) \\ \& ((\neg \neg X \vee \neg Z) \vee (X \& Y)) = ((\neg X \vee \neg Y) \vee (\neg X \& Z)) \& ((X \vee \neg Z) \vee (X \& Y)).$$

- После третьего этапа получаем

$$((\neg X \vee \neg Y) \vee (\neg X \& Z)) \& ((X \vee \neg Z) \vee (X \& Y)) = \\ = (((\neg X \vee \neg Y) \vee \neg X) \& ((\neg X \vee \neg Y) \vee Z)) \& ((X \vee \neg Z) \vee (X \& Y)) = \\ = (((\neg X \vee \neg Y) \vee \neg X) \& ((\neg X \vee \neg Y) \vee Z)) \& (((X \vee \neg Z) \vee X) \& ((X \vee \neg Z) \vee Y)).$$

- Используя идемпотентность и ассоциативность связок, упрощаем результат:

$$(\neg X \vee \neg Y) \& (\neg X \vee \neg Y \vee Z) \& (X \vee \neg Z) \& (X \vee \neg Z \vee Y).$$

Заметим, что в силу ассоциативности операций $\&$ и \vee , как бы мы не расставляли скобки в выражениях $A_1 \& A_2 \& \dots \& A_n$ и $A_1 \vee A_2 \vee \dots \vee A_n$ ($n > 2$), всегда будем приходить к логически эквивалентным формулам. Кроме того, КНФ не являются однозначно определенными. Формула может иметь несколько равносильных друг другу КНФ.

1.9 Сведение к дизъюнктам

Метод резолюций работает с дизъюнктами. Любая формула A исчисления предикатов может быть преобразована во множество дизъюнктов следующим образом (здесь знак \rightarrow используется для обозначения способа преобразования формул).

1. **Приводим к предваренной форме.** См. алгоритм в доказательстве теоремы 2. Получаем формулу A_1 , логически эквивалентную формуле A .

2. **Проводим сколемизацию** формулы A_1 и получаем формулу A_2 . По теореме 3 формулы A_1 и A_2 одновременно выполняемы или невыполнимы.

3. **Элиминация кванторов всеобщности.** Выполняем преобразования вида: $\forall x P(x) \rightarrow P(x)$. Это преобразование не является логически эквивалентным, но формула $\forall x P(x) \supset P(x)$ является

логически общезначимой. После третьего этапа получаем формулу A_3 , которая не содержит кванторов, и формулы A_2 и A_3 одновременно выполнимы или невыполнимы.

4. Приведение к конъюнктивной нормальной форме. Выполняем логически эквивалентные преобразования вида: $X \vee (Y \& Z) = (X \vee Y) \& (X \vee Z)$. После четвертого этапа получаем формулу A_4 , которая находится в конъюнктивной нормальной форме.

5. Элиминация конъюнкции. Преобразование: $X \& Y \rightarrow X, Y$. После пятого этапа формула распадается на множество дизъюнктов.

Теорема 5. Если Γ — множество дизъюнктов, полученных из формулы A , то A является противоречием тогда и только тогда, когда множество Γ невыполнимо.

(Множество формул Γ невыполнимо — это означает, что не существует интерпретации, в которой бы все формулы Γ имели бы значение И.)

Доказательство. Формулы A и A_4 одновременно выполнимы или невыполнимы в любой интерпретации. Формула A_4 является противоречием тогда и только тогда, когда не существует интерпретации, в которой бы все формулы Γ имели бы значение И.

1.10 Правило резолюции для исчисления высказываний

Пусть C_1 и C_2 — два дизъюнкта в исчислении высказываний, и пусть $C_1 = P \vee A$, а $C_2 = \neg P \vee B$, где P — пропозициональная переменная, а A, B — любые дизъюнкты (в частности, может быть, пустые или состоящие только из одного литерала). Правило вывода

$$\frac{C_1, C_2}{A \vee B} R$$

называется **правилом резолюции**. Дизъюнкты C_1 и C_2 называются **резольвируемыми** (или **родительскими**), дизъюнкт $A \vee B$ — **резольвентой**, а формулы P и $\neg P$ — **контрарными** литералами.

Правило резолюции — это очень мощное правило вывода. Многие правила вывода являются частными случаями правила резолюции:

$$\begin{array}{c}
\frac{A, A \supset B}{B} \text{Modus ponens} \quad \frac{A, \neg A \vee B}{B} R \\
\frac{A \supset B, B \supset C}{A \supset C} \text{Транзитивность} \quad \frac{\neg A \vee B, \neg B \vee C}{\neg A \vee C} R
\end{array}$$

Доказательство следующей теоремы тривиально.

Теорема 6. Резольвента является логическим следствием резольвируемых дизъюнктов.

1.11 Унификация

Пусть имеются две элементарные формулы A и B языка первого порядка. Эти формулы могут содержать индивидные (предметные) переменные. В некоторых случаях возможна такая подстановка термов вместо переменных в этих формулах, что формулы A и B (вообще говоря, различные) становятся тождественными. Такая подстановка называется **унификатором**. Например, формулы $F(x, p(x, y))$ и $F(t(z, c), u)$ можно унифицировать: для этого достаточно переменную x заменить на $t(z, c)$, а вместо переменной u подставить терм $p(t(z, c), y)$; в результате унификации получаем формулу $F(t(z, c), p(t(z, c), y))$.

В общем случае под **подстановкой** мы понимаем множество $\theta = \{t_1/X_1, \dots, t_n/X_n\}$, где все X_i ($i=1, \dots, n$) — различные переменные, а t_i — термы, отличные от всех X . Результат применения подстановки θ к элементарной формуле E (т.е. результат одновременной замены всех переменных X_1, \dots, X_n на термы t_1, \dots, t_n соответственно) обозначается $E\theta$. Композиция $\theta\lambda$ подстановок θ и λ определяется естественным образом и удовлетворяет условию $(E\theta)\lambda = E(\theta\lambda)$ для любой формулы E .

Унификатором формул E_1 и E_2 называется такая подстановка θ , что $E_1\theta = E_2\theta$. Вообще говоря, у двух формул E_1 и E_2 может быть бесконечно много унификаторов, однако среди них всегда имеется **наиболее общий** (или самый общий) унификатор σ , такой, что все остальные унификаторы получаются в результате действия на σ некоторыми подстановками.

1.12 Правило резолюции для исчисления предикатов

Для применения правила резолюции нужны контрарные литералы в резолювируемых дизъюнктах. Пусть C_1 и C_2 — два дизъюнкта в исчислении предикатов. Правило вывода

$$\frac{C_1, C_2}{(A \vee B)\sigma} R$$

называется правилом резолюции в исчислении предикатов, если в дизъюнктах C_1 и C_2 существуют унифицируемые контрарные литералы P_1 и P_2 , то есть $C_1 = P_1 \vee A$, а $C_2 = \neg P_2 \vee B$, причем атомарные формулы P_1 и P_2 являются унифицируемыми наиболее общим унификатором σ . В этом случае резольвентой дизъюнктов C_1 и C_2 является дизъюнкт $(A \vee B)\sigma$, полученный из дизъюнкта $A \vee B$ применением унификатора σ .

1.13 Алгоритм резолюций

Мы можем попробовать доказать невыполнимость конечного множества дизъюнктов Γ с помощью следующего алгоритма резолюций (\square обозначает пустой дизъюнкт).

while $\square \notin \Gamma$ **do**

begin

- выбираем дизъюнкты C и D из Γ , содержащие унифицируемые контрарные литералы;
- вычисляем резольвенту R ;
- заменяем Γ на $\Gamma \cup \{R\}$.

end

Теорема 7. Если множество дизъюнктов является невыполнимым, то алгоритм резолюций за конечное число шагов придет к противоречию (пустому дизъюнкту).

Пример. Проверим невыполнимость множества дизъюнктов

$$\Gamma = \{P \vee Q, P \vee R, \neg Q \vee \neg R, \neg P\}.$$

Дизъюнкты удобно пронумеровать. Получится следующий список:

1. $P \vee Q$

2. $P \vee R$
3. $\neg Q \vee \neg R$
4. $\neg P$

Затем вычисляются и добавляются к Γ резольвенты. В списке, который приводится ниже, каждый дизъюнкт – резольвента некоторых из предшествующих дизъюнктов. Номера их приведены в скобках справа от соответствующей резольвенты.

5. $P \vee \neg R$ (1,3)
6. Q (1,4)
7. $P \vee \neg Q$ (2,3)
8. R (2,4)
9. P (2,5)
10. $\neg R$ (3,6)
11. $\neg Q$ (3,8)
12. $\neg R$ (4,5)
13. $\neg Q$ (4,7)
14. \square (4,9)

При работе алгоритма возможны три случая.

1. Среди текущего множества дизъюнктов нет резольвируемых. Это означает, что множество формул Γ выполнимо.

2. В результате очередного применения правила резолюции получен пустой дизъюнкт. Это означает, что множество формул Γ невыполнимо

3. Процесс не заканчивается, то есть множество дизъюнктов пополняется все новыми резольвентами, среди которых нет пустых. Выполнение алгоритма не завершится, например, для множества $\{P, \neg P \vee Q\}$. Резольвентный дизъюнкт Q будет порождаться неограниченное число раз.

Алгоритм проверки невыполнимости – недетерминированный: вообще говоря, возможен не один выбор для резольвируемых дизъюнктов и контрарных литералов. В приведенном примере мы выбирали дизъюнкты C и D в лексикографическом порядке номеров. Такая стратегия неоптимальная. Некоторые резольвенты оказались ненужными, а также вычислялись в некоторых случаях более одного раза. Для сравнения продемонстрируем теперь применение этого же алгоритма с минимумом резолюций:

- 5. Q (1, 4)
- 6. R (2, 4)
- 7. $\neg Q$ (3, 6)
- 8. \square (5, 7)

Конкретные реализации выбора резольвируемых дизъюнктов и контрарных литералов называются *стратегиями* метода резолюции.

1.14 Опровержение методом резолюций

Опровержение методом резолюций – это алгоритм автоматического доказательства теорем в прикладном исчислении предикатов, который сводится к следующему. Пусть нужно установить выводимость $\Gamma \vdash G$.

Каждая формула множества Γ и формула $\neg G$ (отрицание целевой теоремы) независимо преобразуется во множества дизъюнктов. Пусть S — полученное совокупное множество дизъюнктов.

Теорема 8. Множество дизъюнктов S невыполнимо тогда и только тогда, когда $\Gamma \vdash G$.

С помощью алгоритма резолюций, примененному к множеству дизъюнктов S , мы можем установить выводимость $\Gamma \vdash G$.

При работе алгоритма возможны три случая.

1. На каком-то этапе отсутствуют резольвируемые дизъюнкты. Это означает, что теорема опровергнута, то есть формула G не выводима из множества формул Γ .

2. В результате очередного применения правила резолюции получен пустой дизъюнкт. Это означает, что теорема доказана, то есть $\Gamma \vdash G$.

3. Процесс не заканчивается, то есть множество дизъюнктов пополняется все новыми резольвентами, среди которых нет пустых. Это ничего не означает.

Замечание. Таким образом, исчисление предикатов является **полуразрешимой** теорией, а метод резолюций является **частичным алгоритмом** автоматического доказательства теорем.

Пример. Докажем методом резолюций теорему $\vdash (((A \supset B) \supset A) \supset A)$. Сначала нужно преобразовать в дизъюнкты отрицание целевой формулы $\neg(((A \supset B) \supset A) \supset A)$. Используя правила из

пункта «Сведения к дизъюнктам», получаем множество дизъюнктов $\{A \vee A, \neg B \vee A, \neg A\}$.

После этого проводится резольвирование имеющихся предложений 1–3.

1. $A \vee A$.
2. $\neg B \vee A$.
3. $\neg A$.
4. A (1, 3)
5. \square (3, 4).

Таким образом, теорема доказана.

В настоящее время предложено множество различных стратегий метода резолюций. Среди них различаются **полные** и **неполные** стратегии. Полные стратегии – это такие, которые гарантируют нахождение доказательства теоремы, если оно вообще существует. Неполные стратегии могут в некоторых случаях не находить доказательства, зато они работают быстрее. Следует иметь в виду, что автоматическое доказательство теорем методом резолюций имеет по существу переборный характер, и этот перебор столь велик, что может быть практически неосуществим за приемлемое время.

При автоматическом доказательстве теорем методом резолюций основная доля вычислений приходится на поиск унифицируемых дизъюнктов. Таким образом, эффективная реализация алгоритма унификации критически важна для практической применимости метода резолюций.

Рассмотрим следующий пример из [4], показывающий, как можно получить ответы на вопросы с помощью метода резолюций. Предположим, что у нас есть предикат $F(x, y)$, означающий, что « x – отец y », и истинна следующая формула

$$F(john, harry) \& F(john, sid) \& F(sid, liz).$$

Таким образом, у нас есть три дизъюнкта. Они не содержат переменных или импликаций, а просто представляют базисные факты.

Введем еще три предиката $M(x)$, $S(x, y)$ и $B(x, y)$, означающие соответственно, что x – мужчина, что он единокровен с y , что он брат y . Мы можем записать следующие аксиомы о семейных отношениях:

$$\forall x, y (F(x, y) \supset M(x));$$

$$\begin{aligned} &\forall x,y,w ((F(x,y) \& F(x,w)) \supset S(y,w)); \\ &\forall x,y ((S(x,y) \& M(x)) \supset B(x,y)). \end{aligned}$$

Они утверждают следующее: 1) все отцы – мужчины; 2) если у детей один отец, то они единокровны; 3) брат – это единокровный мужчина.

Пусть мы задали вопрос $\exists z B(z, \text{harry})$? (Есть ли у *harry* брат?) Чтобы найти ответ с помощью метода резолюции, мы записываем отрицание вопроса $\forall z \neg B(z, \text{harry})$. Затем приводим аксиомы к нормальной форме и записываем каждый дизъюнкт в отдельной строке (так как каждый дизъюнкт истинен сам по себе):

1. $\neg F(x,y) \vee M(x)$;
2. $\neg F(x,y) \vee \neg F(x,w) \vee S(y,w)$;
3. $\neg S(x,y) \vee \neg M(x) \vee B(x,y)$;
4. $F(\text{john}, \text{harry})$;
5. $F(\text{john}, \text{sid})$;
6. $F(\text{sid}, \text{liz})$;
7. $\neg B(z, \text{harry})$.

Для применения резолюции необходимо найти для данной пары дизъюнктов такую подстановку термов вместо переменных, чтобы после нее некоторый литерал одного из дизъюнктов стал отличаться от некоторого литерала другого дизъюнкта лишь отрицанием. Если, например, мы подставим *john* вместо *x* и *sid* вместо *y*, то получим следующее:

$$\neg F(\text{john}, \text{sid}) \vee \neg F(\text{john}, w) \vee S(\text{sid}, w).$$

Мы можем применить правило резолюции к этому дизъюнкту и к (5), что дает новый дизъюнкт

$$8. \neg F(\text{john}, w) \vee S(\text{sid}, w) \quad (5, 2) \{x \rightarrow \text{john}, y \rightarrow \text{sid}\}$$

Сокращенная запись в правой части предыдущей строки означает, что правило резолюции применялось к дизъюнктам с номерами 2 и 5, причем была произведена подстановка $x \rightarrow \text{john}$ и $y \rightarrow \text{sid}$. Продолжая, получим

9. $S(\text{sid}, \text{harry})$ (4, 8) $\{w \rightarrow \text{harry}\}$
10. $M(\text{sid})$ (6, 1) $\{x \rightarrow \text{sid}, y \rightarrow \text{liz}\}$
11. $\neg S(\text{sid}, y) \vee B(\text{sid}, y)$ (10, 3) $\{x \rightarrow \text{sid}\}$
12. $B(\text{sid}, \text{harry})$ (9, 11) $\{y \rightarrow \text{harry}\}$
13. \square (12, 7) $\{z \rightarrow \text{sid}\}$

Таким образом, мы вывели дизъюнкт (12), выражающий, что *sid* — брат *harry*, используя аксиомы и факты (4), (5) и (6). Это противоречит отрицанию нашего вопроса, которое утверждает, что *harry* не имеет братьев.

2 ВВЕДЕНИЕ В ПРОЛОГ

2.1 Хорновская логическая программа

Программа на Прологе представляет собой, грубо говоря, описание некоторой прикладной теории первого порядка. Как мы знаем, такая теория отличается от исчисления предикатов первого порядка наличием собственных аксиом. Так вот программа на Прологе есть запись собственных аксиом и ничего больше. Причем эти аксиомы пишутся в таком виде, что программировать может даже человек, не знающий математической логики. После этого программист обращается к интерпретатору Пролога с запросом. Запрос — это формула созданной теории, относительно которой требуется установить: является она теоремой данной теории или нет. И интерпретатор Пролога выдает ответ «Yes» или «No».

Покажем, как прикладная формальная теория, в рамках которой решается задача автоматического доказательства, реализуется в языке Пролог.

- **Хорновским дизъюнктом** называется дизъюнкт, содержащий не более одного позитивного литерала.
- Дизъюнкт, состоящий только из одного позитивного литерала, называется **фактом**.
- Дизъюнкт, имеющий позитивный и негативные литералы называется **правилом**.
- Логическая или, точнее, **хорновская логическая программа** состоит из набора хорновских дизъюнктов.

Таким образом, хорновский дизъюнкт есть элементарная формула, либо он имеет вид $A_0 \vee \neg A_1 \vee \dots \vee \neg A_n$, где A_i — атомарные формулы и $n \geq 1$.

Структура хорновских дизъюнктов такова, что, с одной стороны, с их помощью довольно естественно описываются многие задачи, а с другой стороны, они допускают простую процедурную интерпретацию.

Факты в программе описываются в виде, привычном для элементарной формулы, например $P(t_1, t_2, t_3)$.

Наличие в программе факта $P(t_1, \dots, t_n)$, содержащего (в своих аргументах) переменные X_1, \dots, X_m , означает, что для любых объектов X_1, \dots, X_m имеет место некоторое отношение $P(t_1, \dots, t_n)$.

Формула $A_0 \vee \neg A_1 \vee \dots \vee \neg A_n$ равносильна формуле $A_1 \& \dots \& A_n \supset A_0$, и в обозначениях Пролога данное принято записывать по-иному:

$A_0 :- A_1, \dots, A_n.$

Если в таком правиле встречаются переменные X_1, \dots, X_m , то читается оно следующим образом: для всех объектов X_1, \dots, X_m из A_1, \dots, A_n следует A_0 (или, что эквивалентно, для всех X_1, \dots, X_m , если верны утверждения A_1, \dots, A_n , то верно также A_0).

Запросом к логической программе называется формула вида $A_1 \& \dots \& A_n$, где все A_i — атомарные формулы. Следуя синтаксису Пролога, вместо этой формулы мы пишем

$?- A_1, \dots, A_n.$

Запрос, не содержащий переменных, читается так: верно ли, что A_1, \dots и A_n ? Если же в атомарных формулах содержатся переменные X_1, \dots, X_m , то его следует читать иначе: для каких объектов X_1, \dots, X_m верно A_1, \dots и A_n ? При этом, разумеется, считается, что машине «известна» только та информация, которая представлена в программе (и, быть может, свойства некоторых встроенных предикатов и функций, таких, как $<$ или $+$), и ответ на запрос должен логически вытекать из этой информации.

Таким образом, логические программы имеют простую и естественную семантику: запрос

$?- A_1, \dots, A_n.$

(с переменными X_1, \dots, X_m) к программе P понимается как требование вычислить все значения переменных X_1, \dots, X_m , при которых утверждения A_1, \dots и A_n логически следуют из утверждений, содержащихся в P .

Пример из раздела 1.14 запишется в виде следующей программы на Прологе.

```

m(X) :-
    f(X, Y) .
s(Y, W) :-
    f(X, Y) , f(X, W) .
b(X, Y) :-
    s(X, Y) , m(X) .
f(john, harry) .
f(john, sid) .
f(sid, liz) .

```

Обычно Пролог-система работает в форме диалога с пользователем. Утверждение, которое требуется доказать, вводится с клавиатуры.

Для данной программы можно задавать различные вопросы Пролог-системе. Вызов

```
?- b(Z, harry) .
```

означает вопрос «Существует ли такой Z, который является братом harry?».

Пролог ответит

```

Z = sid
Yes

```

Опишем более систематически процесс программирования на языке Пролог.

Решение задач. Конечной целью составления компьютерной программы является создание инструмента, предназначенного для решения задачи из некоторой прикладной области. *Прикладная область* — это некоторая абстрактная часть мира или область знаний. *Структура* прикладной области состоит из значимых для этой области сущностей, функций и отношений. В удачной компьютерной программе структура прикладной области моделируется таким образом, что поведение этой программы во время выполнения будет отражать какие-то существенные аспекты данной структуры. К примеру, отношение, существующее между некоторыми сущностями прикладной области, должно, по аналогии, соблюдаться и для обозначений, которые представляют эти сущности в программе, моделирующей эту область.

Программирование. Процесс составления программы на Прологе в основном сходен с процессом построения теории в логике предикатов.

- Программист анализирует значимые сущности, функции и отношения из прикладной области и выбирает обозначения для них.
- Программист семантически определяет каждую значимую функцию и каждое значимое отношение (предикат). Для предикатов указывается, какие конкретные их реализации дают *истину*, а какие — *ложь*.
- Программист аксиоматически определяет каждый предикат при помощи правил и фактов Пролога. (Общее название для правил и фактов — *предложения*, иногда употребляется термин *клауза*.) Аксиоматическое определение предиката будет удачным, если оно отразит смысл семантического определения. Множеством аксиоматических определений всех значимых для заданной предметной области предикатов является программа, моделирующая структуру этой области.
- Для выполнения запросов к множеству правил и фактов программист или пользователь применяет интерпретатор языка Пролог. Совокупность, состоящую из запроса, множества правил и фактов программы и интерпретатора языка, можно рассматривать как алгоритм решения задач из прикладной области. При этом запрос, правила и факты программы представляют собой начальные формулы алгоритма, а интерпретатор содержит правила преобразования этих формул. Интерпретатор играет роль активной силы, которая выполняет выводы из правил программы и тем самым *реализует* или *развертывает* предикаты, определенные правилами программы.

2.2 Сеанс работы с интерпретатором Пролога

Существует большое количество реализаций языка Пролог, как коммерческих, так и свободно распространяемых. Мы будем ориентироваться на SWI-Prolog (версия 3.4.5 и старше), разрабо-

танный в университете города Амстердама. Возможностей данной реализации вполне достаточно для логического программирования.

SWI-Prolog распространяется¹ под лицензией GPL, что обеспечивает возможность его использования без нарушений чьих-либо коммерческих интересов. Эта версия языка Пролог доступна как пользователям ОС Linux, так и пользователям Windows. Отметим, что классические и лучшие учебники по Прологу практически согласованы с языком SWI-Prolog [2, 8]. Мы считаем, что при обучении декларативному программированию надо использовать интерпретаторы — тем самым уже на первом этапе обучения студенты могут оценить декларативное программирование по достоинству. При решении коммерческих задач надо переходить к компиляторам.²

После вызова на выполнение интерпретатор выдает приглашение «?–», которое свидетельствует о том, что работа ведется в режиме выполнения запросов. После ввода запроса интерпретатор обращается к своей базе данных, находит в ней факты и правила, необходимые для ответа на запрос, формирует и выводит ответ. Непосредственно после загрузки в базе данных интерпретатора находятся только стандартные (встроенные) предикаты, обеспечивающие работу системы и выполнение вспомогательных функций.

Запрос (цель) вводится после приглашения и обязательно заканчивается точкой, например,

```
?– 5+4<3.
No
```

Пролог анализирует запрос и выдает ответ Yes (да) в случае истинности утверждения и No (нет) в противном случае или когда ответ не может быть найден.

Хранят программы на языке Пролог в текстовых файлах, чаще всего имеющих расширение pl, например example.pl. Для того чтобы Пролог мог оперировать информацией, содержа-

¹ www.swi-prolog.org

² Visual-Prolog, web-сайт: www.visual-prolog.com

щейся в файле, он должен ознакомиться с его содержимым (*проконсультироваться* с ним). Это можно сделать несколькими способами. При использовании первого варианта в квадратных скобках записывается имя файла (без `pl`, если файл находится в рабочем каталоге), например,

```
?- [example].
```

В случае удачного завершения этой операции будет выдано сообщение, аналогичное следующему:

```
% example compiled 0.00 sec, 612 bytes
Yes
```

В противном случае будет выдан список ошибок (ERROR) и/или предупреждений (Warning).

Второй способ состоит в вызове встроенного предиката `consult`, которому в качестве аргумента передается имя файла (также без расширения), например:

```
?- consult(example).
```

Расширение `pl` часто используется для файлов, содержащих программы на языке программирования Perl, поэтому можно встретить и другие расширения для файлов с программами на Прологе. Для загрузки файлов с расширениями, отличными от `pl`, все имя файла следует обязательно заключать в апострофы:

```
?- consult('example2.prolog').
```

```
?- ['example2.prolog'].
```

Обе эти команды добавляют факты и правила из указанного файла в базу данных Пролога. Можно загружать несколько файлов одновременно. В этом случае они перечисляются через запятую, например

```
?- [example1, 'example2.prolog'].
```

Если загружаемый файл находится не в рабочем каталоге, тогда нужно указывать полное имя файла вместе с каталогом:

```
?- ['d:\prolog\work\example.pl'].
```

Важно помнить, что все запросы должны заканчиваться точкой. Если вы забудете ее поставить, то Пролог выведет символ ' | ' и будет ожидать дальнейшего ввода. В этом случае надо ввести точку и нажать клавишу Enter:

```
?- [example]
| .
Yes
```

2.3 Пример Пролог-программы: родственные отношения

*Авраам родил Исаака;
Исаак родил Иакова;
Иаков родил Иуду и братьев его;*

...

Евангелие от Матвея

2.3.1 Используем только факты

Данный пример взят из [2, стр. 26 и далее]. В качестве предметной области выбрано множество людей. Некоторые из них находятся в различных родственных отношениях. Одно из таких отношений — отношение «родитель». Тот факт, что Том является одним из родителей Боба, можно записать следующим образом:

```
родитель(том, боб) .
```

В данном случае в качестве имени двухместного предиката выбрано слово «родитель»; том и боб — аргументы этого предиката. Заметим, что язык Пролог чувствителен к выбору регистра написания слов — имена предикатов пишутся со строчной буквы,

и имена объектов (константы) предметной области пишутся также со строчной буквы.³

Рис. 1 представляет дерево родственных отношений.

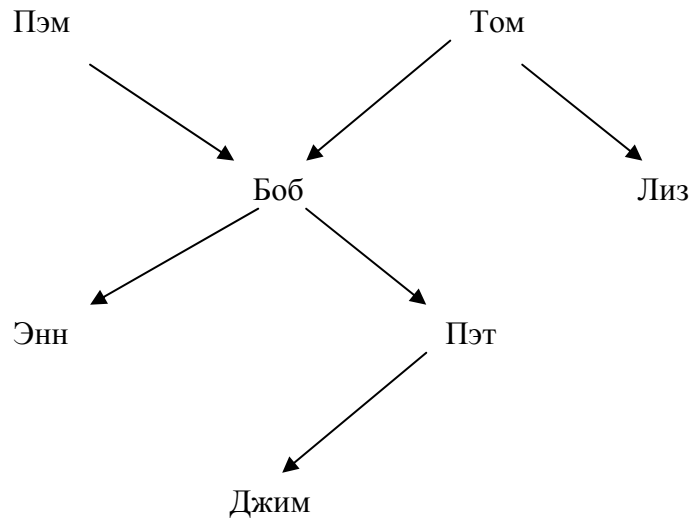


Рис. 1 — Семейное дерево

Эти родственные отношения можно представить в виде программы, состоящей из шести предложений, каждое из которых объявляет один факт о предикате родитель.

```

родитель ( пэм, боб) .
родитель ( том, боб) .
родитель ( том, лиз) .
родитель ( боб, энн) .
родитель ( боб, пэт) .
родитель ( пэт, джим) .

```

Например, факт родитель (том, боб) представляет собой конкретный экземпляр отношения «родитель». В целом отношение (предикат) определяется как множество всех своих экземпляров. В данном — аргументами предиката являются константы, называемые **атомами**.

После загрузки данной программы можно задать некоторые вопросы об отношении родитель, например является ли Боб одним из родителей Пэт? Этот вопрос можно передать системе Пролог, введя его с клавиатуры:

³ Более точно синтаксис описан далее.

?- родитель (боб, пэт) .

Обнаружив, что это — факт, о существовании которого утверждается в программе, Пролог отвечает:

Yes

После этого можно задать еще вопрос:

?- родитель (лиз, пэт) .

Система Пролог ответит:

No

поскольку в программе нет упоминания о том, что Лиз является одним из родителей Пэт. Система ответит также «No» на вопрос

?- родитель (том, бэн) .

Поскольку в программе даже не встречалось имя Бэн.

Кроме того, системе можно задать более интересные вопросы. Например, кто является родителями Лиз?

?- родитель (X, лиз) .

На этот раз Пролог ответит не просто «Yes» или «No», а сообщит такое значение X, при котором приведенное выше утверждение является истинным. Поэтому ответ будет таковым:

X = том

И система будет ждать, пока мы не нажмем клавишу <Enter>, а после этого ответит Yes .

В данном случае одним из аргументов предиката выступает **переменная** (отметим, что переменная указывается прописной буквой).

Вопрос о том, кто является детьми Боба, можно сообщить системе Пролог следующим образом:

?- родитель (боб, X) .

На этот раз имеется больше одного возможного ответа. Интерпретатор вначале выдаст в ответ одно решение:

X = энн

Теперь можно потребовать у системы сообщить еще одно решение (введя точку с запятой), и Пролог найдет следующий ответ:

X = пэт

Если после этого будут затребованы дополнительные решения, Пролог ответит «No», поскольку все решения уже исчерпаны.

Этой программе может быть задан еще более общий вопрос о том, кто является чьим родителем? Этот вопрос можно также сформулировать иным образом: «Найти X и Y, такие, что X является одним из родителей Y». Этот вопрос может быть оформлен на языке Пролог следующим образом:

?- родитель (X, Y) .

После этого Пролог начнет отыскивать все пары родителей и детей одну за другой. Решения отображаются на экране по одному до тех пор, пока Пролог получает указание найти следующее решение (в виде точки с запятой) или пока не будут найдены все решения. Ответы выводятся следующим образом:

X = пэм
 Y = боб ;
 X = том
 Y = боб ;
 X = том
 Y = лиз ;
 X = боб
 Y = энн ;
 X = боб
 Y = пэт ;

```
X = пэт
Y = джим ;
No
```

Чтобы прекратить вывод решений, достаточно нажать клавишу <Enter> вместо точки с запятой.

Этой программе, рассматриваемой в качестве примера, можно задать еще более сложный вопрос, например спросить о том, кто является родителями родителей Джима (дедушками и бабушками). Но у нас в программе нет предиката родитель_родителя, поэтому этот запрос необходимо разбить на следующие два этапа.

1. Кто является одним из родителей Джима? Предположим, что это — некоторый объект Y.

2. Кто является одним из родителей Y? Предположим, что это — некоторый объект X.

Подобный сложный вопрос записывается на языке Пролог как последовательность двух простых:

```
?- родитель (Y, джим) , родитель (X, Y) .
```

Ответ будет следующим:

```
Y = пэт
X = боб ;
No
```

Этот составной запрос можно прочесть таким образом: найти такие X и Y, которые удовлетворяют следующим двум требованиям: родитель (Y, джим) и родитель (X, Y). Поскольку это конъюнкция двух целей, то она коммутативна, и, следовательно, ответ на запрос останется прежним, если мы переставим подцели:

```
?- родитель (X, Y) , родитель (Y, джим) .
```

Аналогичным образом программе можно задать вопрос о том, кто является внуками Тома:

?- родитель (том, X) , родитель (X, Y) .

Система Пролог ответит следующим образом:

X = боб
Y = энн ;
X = боб
Y = пэт ;
No

Могут быть заданы и другие вопросы, например имеют ли Энн и Пэт общих родителей. Соответствующий запрос в языке Пролог выглядит следующим образом:

?- родитель (X, энн) , родитель (X, пэт) .

Ответом на него является:

X = боб

В нашей программе только один предикат, но можно добавить и другие. Определим пол людей. Это можно сделать, по крайней мере, двумя способами. Можно использовать двуместный предикат пол:

пол (пэм, женщина) .
пол (пэт, женщина) .
пол (лиз, женщина) .
пол (энн, женщина) .
пол (боб, мужчина) .
пол (том, мужчина) .
пол (джим, мужчина) .

Второй способ — это ввести два одноместных предиката, указывающих свойство человека быть мужчиной или женщиной.

женщина (пэм) .
 женщина (пэт) .
 женщина (лиз) .
 женщина (энн) .
 мужчина (боб) .
 мужчина (том) .
 мужчина (джим) .

Какой способ выбрать — дело вкуса. Но в первом случае в программу вводится новый универсум, состоящий из двух объектов: мужчина и женщина.

2.3.2 Используем факты и правила

Все эти предикаты представляют элементарные формулы (в терминологии Пролога — факты). Но хорновские дизъюнкты в программе могут быть и более сложными. С точки зрения Пролога мы определяем новые предикаты через уже существующие предикаты. Например, можно описать предикат *мать*:

```
мать (X, Y) :-
    родитель (X, Y) ,
    пол (X, женщина) .
```

Словесно это можно выразить так: «Если X является родителем Y и пол X есть женский, то X — мать Y».

Но этот предикат можно определить и по-другому:

```
мать (X, Y) :-
    родитель (X, Y) ,
    женщина (X) .
```

Мы привели два примера правил, т.е. утверждения, которые принято считать истинными, если выполнены некоторые условия. Поэтому принято считать, что правила состоят из следующих частей:

- условие (правая часть правила);
- заключение (левая часть правила).

Часть, соответствующая заключению, называется также головой правила, а часть, соответствующая условию, — телом правила. Если тело правила является истинным, то его логическим следствием является голова правила. Предполагается, что на переменные в правилах (да и в фактах) распространяется действие квантора всеобщности («для всех»).

Тело правила состоит из нескольких элементарных формул (предикатов), разделенных запятыми. Запятые в данном случае обозначают операцию конъюнкции. Поскольку конъюнкция коммутативна, то элементарные формулы в теле правила, вообще говоря, можно переставить (о ситуациях, когда это делать нельзя — узнаете позже), например:

мать (X, Y) :—
 женщина (X) ,
 родитель (X, Y) .

В правой части правила может стоять и один предикат, например определим отношение отпрыск:

отпрыск (X, Y) :—
 родитель (Y, X) .

Применение правил в языке Пролог иллюстрируется в следующем примере. Зададим программе вопрос, является ли Лиз отпрыском Тома:

?— отпрыск (лиз, том) .

Поскольку в программе отсутствуют факты о дочерях и сыновьях, единственным способом поиска ответа на этот вопрос является использование правила с определением предиката отпрыск. Данное правило является общим в том смысле, что его можно применить к любым объектам X и Y; но оно также применимо и к таким конкретным объектам, как лиз и том, потому что выражение отпрыск (X, Y) можно унифицировать с выражением отпрыск (лиз, том). Для этого вместо X необходимо подставить лиз, а вместо Y — том. Это действие называется конкре-

тизацией переменных (в данном случае — X и Y), которая выполняется подстановкой:

X=лиз, Y=том.

После конкретизации будет получен частный случай общего правила, который выглядит следующим образом:

```
отпрыск (лиз, том) :-  
    родитель (лиз, том) .
```

Таким образом, исходная цель отпрыск (лиз, том) теперь заменена на новую цель родитель (лиз, том), истинность которой теперь должен установить Пролог. Но задача достижения этой цели является тривиальной, поскольку ее можно найти как факт в рассматриваемой программе. Это означает, что голова последнего правила отпрыск (лиз, том) также удовлетворяется и Пролог-система выдает ответ Yes.

Введем теперь отношение «сестра». Соответствующий предикат можно определить через уже существующие предикаты. Словесно это отношение можно представить так:

Для любого X и Y
X является сестрой Y, если

- (1) X и Y имеют общего родителя и
- (2) X — женщина.

На язык Пролог это переводится в виде правила:

```
сестра (X, Y) :-  
    родитель (Z, X) ,  
    родитель (Z, Y) ,  
    пол (X, женщина) .
```

Теперь системе можно задать вопрос:

?- сестра (энн, пэт) .

Ответом будет «Yes», как и следовало ожидать. Поэтому можно сделать вывод, что отношение «сестра» в том виде, в каком оно определено, действует правильно. Но в рассматриваемой программе имеется незаметный на первый взгляд недостаток, который обнаруживается при получении ответа на вопрос о том, кто является сестрой Пэт:

?- сестра (X, пэт) .

Пролог находит два ответа, и один из них оказывается неожиданным.

X = энн;
X = пэт

Второй ответ является также правильным, поскольку в нашем определении предиката сестра нигде не отражено свойство, что X и Y должны иметь различные значения. Это легко добавить в программу, если дополнительно потребовать, что значения переменных X и Y нельзя унифицировать ($X \neq Y$).

сестра (X, Y) :-
 родитель (Z, X) ,
 родитель (Z, Y) ,
 пол (X, женщина) ,
 X \neq Y .

2.3.3 Рекурсивные правила

Введем отношение предок: X является предком Y, если X есть родитель Y, или есть родитель родителя, или есть родитель родителя родителя и т.д. Мы должны определить предикат предок, так чтобы он позволял находить предков до любого колена, а не только до определенного уровня в генеалогическом древе. Нам поможет рекурсивное определение: X является предком Y, если 1) X есть родитель Y или 2) существует такой Z, что X — родитель Z, а Z есть предок Y. Такое определение предиката через самого себя можно задать с помощью двух правил:

предок (X, Y) :- % правило 1
 родитель (X, Y) .

предок (X, Y) :-
 родитель (X, Z) ,
 предок (Z, Y) .

% правило 2

Для задания любого рекурсивного предиката требуется не менее двух правил. По крайней мере, одно правило требуется для конца рекурсии, и, по крайней мере, одно правило требуется для рекурсивного вызова. Рекурсия является одним из фундаментальных принципов программирования на языке Пролог и применяется вместо циклов.

Теперь можно задать вопрос: для кого Пэм является предком?

```
?- предок(пэм, X) .
X = боб ;
X = энт ;
X = пэт ;
X = джим ;
No
```

2.4 Общие принципы поиска ответов на вопросы системой Пролог

Вопрос к системе Пролог всегда представляет собой последовательность из одной или нескольких целей. Чтобы ответить на вопрос, Пролог пытается достичь всех целей. Выражение *достичь цели* означает: продемонстрировать, что цель логически следует (как теорема) из фактов и правил, заданных в программе. Если вопрос содержит переменные, система Пролог должна также найти конкретные объекты (вместо переменных), при использовании которых цели достигаются. Для пользователя отображаются варианты конкретизации переменных, полученные при подстановке конкретных объектов вместо переменных. Если система Пролог не может продемонстрировать ни для какого варианта конкретизации переменных, что цели логически следуют из программы, то выдает в качестве ответа No.

Существует два направления последовательности шагов логического вывода при доказательстве теоремы:

1) от фактов (которые истины по определению), используя правила, переходим к новым утверждениям, пока не получим искомую цель (такое движение приводит, как правило, к информационному взрыву; кроме того, не ясно — когда нужно остановиться);

2) от цели к фактам (используя метод резолюции).

С точки зрения пользователя система Пролог начинает с искомой целью и с помощью правил заменяет текущие цели новыми до тех пор, пока не обнаружится, что новые цели являются простыми фактами.

Рассмотрим вопрос:

?- предок(том, пэт) .

Система Пролог пытается достичь данной цели. Для этого она предпринимает попытки найти в программе предложение, из которого непосредственно может следовать указанная выше цель. Оказывается, что в данном случае подходящими являются только правила 1 и 2, описывающие предикат предок. Головы этих правил можно унифицировать с целью — вот критерий, по которому Пролог-система выбирает подходящие предложения.

Два правила (1 и 2) представляют собой два альтернативных способа дальнейших действий системы. Она вначале проверяет правило, которое находится на первом месте в программе:

```
предок(X, Y) :-                               % правило 1
    родитель(X, Y) .
```

При унификации головы этого правила с целью осуществляется конкретизация переменных:

X = том, Y = пэт

И первоначальная цель предок(том, пэт) заменяется следующей целью: родитель(том, пэт) .

В программе нет факта, который можно было унифицировать с данной целью, поэтому текущая цель не достижима. Теперь система Пролог возвращается к первоначальной цели, чтобы проверить альтернативный способ вывода главной цели предок(том, пэт). Эта цель теперь унифицируется с головой второго правила и конкретизация переменных дает:

$X = \text{том}, Y = \text{пэт}.$

Теперь главная цель заменяется на сложную цель: родитель(том, Z), предок(Z, пэт). Система сталкивается с необходимостью заниматься сразу двумя целями и пытается их достичь в том порядке, в каком они записаны. Первая цель достигается легко, поскольку она унифицируется с одним из фактов в программе. Эта унификация приводит к конкретизации $Z = \text{боб}.$ Таким образом, достигается первая цель, и оставшаяся цель принимает вид предок(боб, пэт).

Для достижения данной цели снова используется правило 1. Следует отметить, что это (второе) применение того же правила не имеет ничего общего с его предыдущим применением. Поэтому система Пролог использует в правиле новый набор переменных при каждом его применении. Чтобы продемонстрировать это, переименуем переменные в правиле 1 для этого этапа следующим образом:

```
предок(X1, Y1) :-                                     % правило 1
    родитель(X1, Y1) .
```

Голова данного правила унифицируется с целью предок(боб, пэт) при подстановке $X1 = \text{боб}, Y1 = \text{пэт}.$ Текущая цель заменяется следующей: родитель(боб, пэт), и данная цель достигается сразу же, поскольку она представлена в программе в виде факта.

Графическая схема выполнения программы (см. рис. 2) имеет форму дерева. Узлы дерева соответствуют целям или спискам целей, которые должны быть достигнуты. Дуги между узлами со-

ответствуют этапам применения (альтернативных) предложений программы, на которых цели одного узла преобразуются в цели другого узла. Верхняя цель достигается после того, как будет найден путь от корневого узла (верхней цели) к лист-узлу, обозначенному как «Yes». Лист носит метку «Yes», если он представляет собой простой факт. Процесс выполнения Пролог-программ состоит в поиске путей, оканчивающихся такими простыми фактами. В ходе поиска система Пролог может войти в одну из ветвей, не позволяющих достичь успеха. При обнаружении того, что ветвь не позволяет достичь цели, система Пролог автоматически возвращается к предыдущему узлу и пытается использовать в этом узле альтернативное предложение.

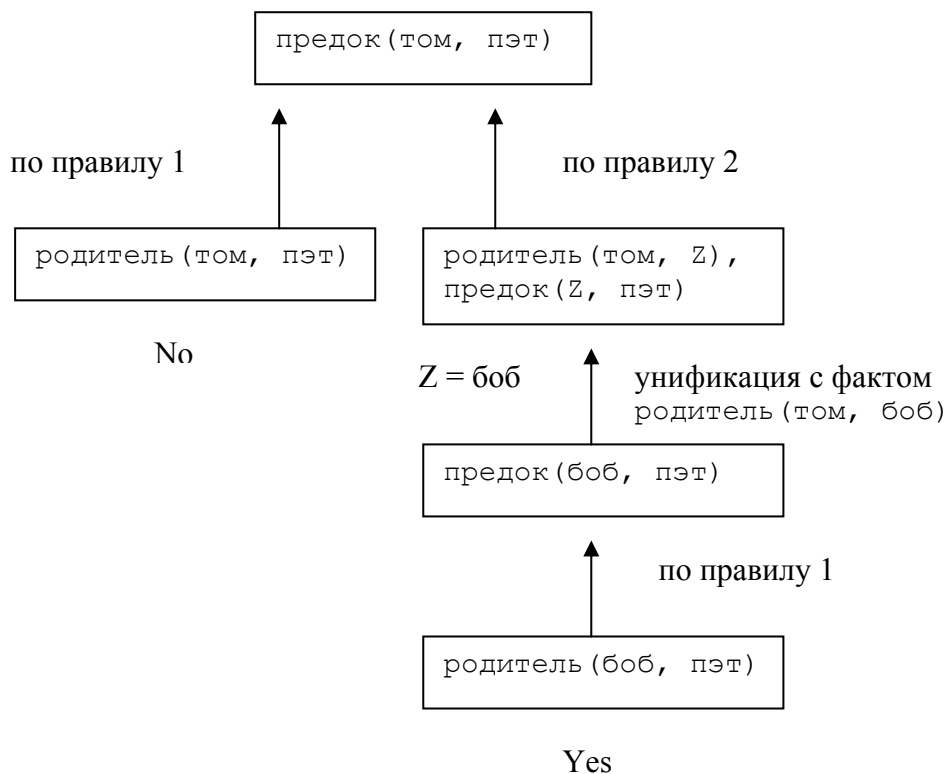


Рис. 2 — Дерево достижения цели предок (том, пэт)

2.5 Декларативная и процедурная семантика программ

И поэтому в логике никогда не может быть ничего неожиданного.

Людвиг Витгенштейн

В рассмотренных выше примерах всегда была возможность понять результаты программы, не зная точно, как система фактически нашла эти результаты. Но иногда важно учитывать, как именно происходит поиск ответа в системе, поэтому имеет смысл проводить различие между двумя семантиками программ Пролога, а именно между

- декларативной семантикой,
- процедурной семантикой.⁴

Декларативная семантика в Прологе касается только отношений, определенных в программе. Поэтому декларативная семантика регламентирует то, каким должен быть результат работы программы. С другой стороны, процедурная семантика определяет также способ получения этого результата, иными словами, показывает, как фактически проводится обработка этих отношений системы Пролог.

Способность системы Пролог самостоятельно отрабатывать многие процедурные детали считается одним из её особых преимуществ. Пролог побуждает программиста в первую очередь рассматривать декларативную семантику программ, в основном независимо от их процедурной семантики. А поскольку результаты программы в принципе определяются их декларативной семантикой, этого должно быть (по сути) достаточно для написания программы. Такая возможность важна и с точки зрения практики, так как декларативные аспекты программы обычно проще для

⁴ В языках программирования процедурной (операционной) семантикой называют описание последствий отдельных шагов вычислений, которые имеют место при выполнении программы. Декларативная (функциональная) семантика — описание функций программы, т.е. установление отношения между входными и выходными данными (экстенциональное или наблюдаемое отношение).

понимания по сравнению с процедурными деталями. Но чтобы полностью воспользоваться этими преимуществами, программист должен сосредоточиваться в основном на декларативной семантике и, насколько это возможно, избегать отвлекаться на детали выполнения. Последние необходимо оставлять в максимально возможной степени для самой системы Пролог.

Такой декларативный подход фактически часто позволяет гораздо проще составлять программы на языке Пролог по сравнению с такими типичными процедурно-ориентированными языками программирования, как Си или Паскаль. Но, к сожалению, декларативный подход не всегда позволяет решить все задачи, иногда процедурные аспекты не могут полностью игнорироваться по практическим причинам, связанным с обеспечением вычислительной эффективности. Желательно, чтобы программист имел правильное представление о том, как работает интерпретатор языка Пролог.

2.5.1 Алгоритм работы интерпретатора Пролога

Рекурсивный алгоритм ответа на запрос $G1, G2, \dots, Gm$ (список целей) следующий.

- Если список целей пуст — завершает работу *успешно*.
- Если список целей не пуст, продолжает работу, выполняя операцию ПРОСМОТР.
- ПРОСМОТР. Просматривает предложения программы от начала к концу до обнаружения первого предложения C , такого, что голова (левая часть) C унифицируема с первой целью $G1$. Если такого предложения обнаружить не удастся, то работа заканчивается *неуспехом*.

Если C найдено и имеет вид

$H :- B1, \dots, Bn,$

то переменные в C переименовываются, чтобы получить такой вариант C' предложения C , в котором нет общих переменных со списком $G1, \dots, Gm$.

Пусть C' — это $H' :- B1', \dots, Bn'$.

Унифицируется $G1$ с H' ; пусть S — результирующая конкретизация переменных. В списке целей $G1, G2, \dots, Gm$ цель $G1$

заменяется списком $B1', \dots, Bn'$, что порождает новый список целей: $B1', \dots, Bn', G2, \dots, Gm$.

Заметим, что 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 $B1'', \dots, Bn'', G2', \dots, Gm'$.

- Вычисляется (используя тот же самый алгоритм) 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 (**бэктрекинг**) к просмотру программы. Этот просмотр продолжается, начиная с предложения, непосредственно следующего за предложением C (C — предложение, использовавшееся последним), и делается попытка достичь успешного завершения с помощью другого предложения.

Вычисление целей интерпретатор Пролога осуществляет с помощью **поиска в глубину с возвратом** (в дереве целей): правило вычислений всегда выбирает первую слева подцель в текущем списке целей, а правило поиска выбирает из программы первое предложение, голова которого унифицируема с данной подцелью. Если вычисление заходит в тупик, т.е. ни одно из утверждений программы не применимо к текущему списку целей, то происходит возврат назад по построенной ветви и для предыдущего состояния пробуются первое из еще не применявшихся к нему утверждений программы.

2.6 Синтаксис языка SWI-Prolog

Программа на языке Пролог обычно описывает некую действительность. Объекты (элементы) описываемого мира пред-

ставляются с помощью термов. **Терм** — это имя *объекта*. Существует 4 вида термов: *атомы*, *числа*, *переменные* и *составные термы*. На рис. 3 представлена классификация объектов данных в языке Пролог. Система SWI-Prolog распознает тип данных в программе по его синтаксической форме. Это возможно благодаря тому, что в синтаксисе языка SWI-Prolog определены разные формы для объектов данных каждого типа. Системе SWI-Prolog (в отличие от некоторых других версий Пролога) не требуется сообщать какую-то дополнительную информацию (наподобие объявления типа данных) для того, чтобы она распознала тип объекта.

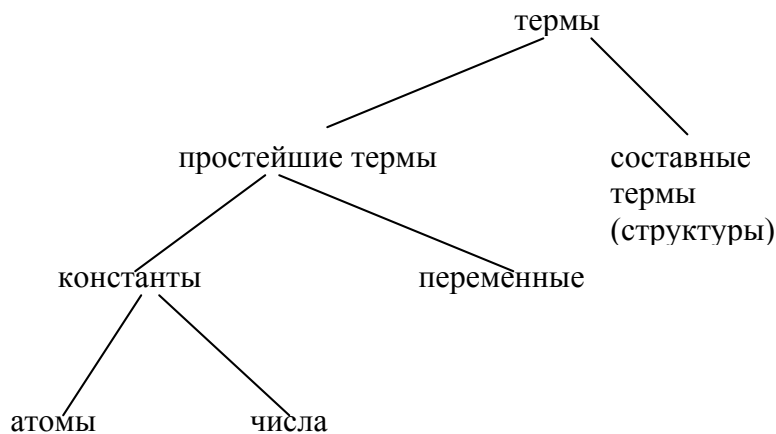


Рис. 3 — Классификация объектов данных в языке Пролог

Переменные записываются с помощью произвольных латинских и русских букв, цифр и знаков подчеркивания, но первый символ должен быть всегда прописной латинской буквой или знаком подчеркивания («_»). Примеры: `X`, `_4711`, `X_1_2`, `Результат`, `_x23`, `_`. Последний пример (единственный символ подчеркивания) является особым случаем — **анонимной переменной** (переменной без имени). Анонимная переменная применяется, когда ее значение не используется в программе. Возможное неоднократное употребление безымянной переменной в одном выражении применяется для того, чтобы подчеркнуть наличие переменных при отсутствии их специфической значимости. Например, следующая ситуация: переменная появляется в предложении только один раз, как в правиле

```
мать (X) :-
    родитель (X, Y) ,
    мужчина (X) .
```

Это правило можно переписать с использованием анонимной переменной:

```
мать (X) :-
    родитель (X, _) ,
    мужчина (X) .
```

Каждый раз при появлении в предложении символа подчеркивания он представляет новую анонимную переменную. Например, правило утверждающее, что в мире есть некто, имеющий ребенка, можно записать так:

```
'кто-то имеет ребенка' :-
    родитель (_, _) .
```

Если анонимная переменная появляется в вопросе, ее значение не выводится, когда система Пролог отвечает на вопросы. Если нас интересуют имена людей, имеющих детей, но не имена детей, то системе можно задать такой вопрос:

```
?- родитель (X, _) .
```

Атомы могут формироваться тремя перечисленными ниже способами.

1. Строки букв (латинских или русских), цифр и знаков подчеркивания, начинающиеся со строчной буквы.

2. Строки некоторых специальных символов (не содержащие в себе пробелов):

```
<<    >>    >>>    >+<    +++    =>    <=    <=>    <->
::+    ::    ...
```

При использовании атомов в этой форме следует соблюдать осторожность: некоторые комбинации специальных символов уже используются в языке в специальных целях, кроме того, какие комбинации таких символов считаются допустимыми, может зависеть от разных версий языка SWI-Prolog.

3. Строки любых символов, заключенные в ординарные апострофы. Внутри могут быть пробелы. Например: 'X', '.....', 'А роза упала на лапу Азора'.

Числа в Прологе бывают целыми и вещественными. Синтаксис целых чисел прост, как это видно из следующих примеров: 1, 1313, 0, -97. Не все целые числа могут быть представлены в машине, их диапазон ограничен интервалом между некоторыми минимальным и максимальным значениями, определенными конкретной реализацией Пролога. SWI-Prolog допускает использование целых чисел в диапазоне от -2^{31} до $2^{31}-1$.

Синтаксис вещественных чисел также зависит от конкретной реализации. Мы будем придерживаться простых правил, понятных из следующих примеров: 3.14, -.0035, 100.2. При обычном программировании на Прологе вещественные числа используются редко. Причина этого кроется в том, что Пролог — язык, предназначенный в первую очередь для обработки символьной, а не числовой информации. При символьной обработке часто используются целые числа, нужда же в вещественных числах невелика. Везде, где можно, Пролог старается привести число к целому виду.

Есть два способа как можно задавать **комментарий** в языке SWI-Prolog. Первый способ: специальные символьные скобки «/*» и «*/», например

```
/* это — комментарий */
```

Еще один метод, более удобный для оформления коротких комментариев, предусматривает использование знака процента. Все, что находится между знаком «%» и концом строки, интерпретируется как комментарий.

```
% Это — также комментарий
```

Составные термы (или **структуры**) состоят из имени структуры (представленного атомом) и списка аргументов (термов Пролога, то есть атомов, чисел, переменных или других составных термов), заключенных в круглые скобки и разделенных

запятыми. Составные термы можно рассматривать как имена каких-то сложных объектов из предметной области. Имя структуры называется **функтором**. Нельзя помещать символ пробела между функтором и открывающей круглой скобкой. В других позициях, однако, пробелы могут быть полезны для создания более читаемых программ. Аргументы составного терма называются также компонентами. Этот метод структурирования данных является простым и мощным. Он является одной из причин того, почему Пролог можно так естественно применять для решения проблем, связанных с символическими манипуляциями. Все структурированные объекты можно представить графически в виде деревьев. Корнем дерева является функтор, а ветвями — компоненты. Если компонент представляет собой структуру, он становится поддеревом этого дерева, которое соответствует всему структурированному объекту.

Примеры структур: `date(16,1,2005)`, `книга(Пушкин, 'Руслан и Людмила')`. Первую структуру можно использовать для представления дат (первого января две тысячи пятого года), а вторую структуру — для представления информации о книгах.

Отметим, что синтаксически **предикаты** имеют такое же строение, как составные термы: предикат состоит из имени предиката (представленного атомом) и списка аргументов (термов Пролога, то есть атомов, чисел, переменных или других составных термов), заключенных в круглые скобки и разделенных запятыми. В качестве имен предикатов можно использовать любые атомы.

Нельзя помещать символ пробела между именем предиката и открывающей круглой скобкой. Количество аргументов предиката называется его **арностью** (местностью). В программе могут присутствовать предикаты с одинаковыми именами и разной арностью. Система Пролог считает такие предикаты различными. Могут быть и 0-местные предикаты; один из таких предикатов мы уже определяли:

```
'кто-то имеет ребенка' :-  
    родитель(_, _).
```


Пролог отличает предикаты от термов, представляющих данные, по той роли, какую они играют в программе. Голова любого правила и любой факт являются предикатами, а аргументы предикатов есть термы. Тело любого правила состоит из предикатов, разделенных запятыми (или представлено одним предикатом). Поэтому синтаксическое выражение книга (Пушкин, 'Руслан и Людмила') может быть термом — именем данных, а может быть предикатом, представляющим отношение между людьми и книгами, которое истинно тогда и только тогда, когда первый аргумент именуется автором книги. Синтаксическая неразличимость термов и предикатов позволяет использовать один и тот же алгоритм унификации наиболее часто повторяющейся операции при работе системы Пролог.

При описании предикатов (например, в help'е системы) часто применяется такое соглашение: в качестве ссылки на предикат используется имя предикатов и его арность (количество формальных параметров), например, 'кто-то имеет ребенка'/0, родитель/2. Формальные параметры предикатов можно различать как входные и/или выходные. Входными называются такие параметры, которые при вызове предиката всегда имеют полностью заданные значения, без неконкретизированных переменных (иначе параметры называются выходными). Тип входных/выходных параметров обозначаются путем указания перед именами параметров префикса «+» (входной) или «-» (выходной). Символ «?» перед параметром обозначает, что он может быть как входным, так и выходным, например, родитель (?X, ?Y).

Составные термы и предикаты представлены в программах, как уже говорилось, в **префиксной** форме: сначала имя, а потом аргументы, но в том случае, когда всего два аргумента, удобно использовать также **инфиксную** запись: имя помещается между аргументами. Есть несколько встроенных атомов, состоящих из специальных символов, используемых в качестве инфиксных имен структур и предикатов. О них речь будет идти позже.

2.7 Порядок предложений и целей

*Наводить порядок надо тогда,
когда еще нет смуты.*

Лао-Цзы

Логически непротиворечивое определение предиката может быть процедурно неправильно. Вызов

?- p.

предиката, определенного правилом

p :- p.

приводит к бесконечному циклу.

Рассмотрим различные варианты программы «родственные отношения», полученные путем переупорядочивания предложений и целей. С точки зрения логики программы в этих вариантах эквивалентны: на любой запрос они должны давать одинаковые результаты.

```
родитель ( пэм, боб) .
родитель ( том, боб) .
родитель ( том, лиз) .
родитель ( боб, энн) .
родитель ( боб, пэт) .
родитель ( пэт, джим) .
```

Вариант 1

```
предок (X, Y) :-                               % pr1
    родитель (X, Y) .
```

```
предок (X, Y) :-                               % pr2
    родитель (X, Z) ,
    предок (Z, Y) .
```

Следующий вызов дает верный ответ.

```
?- предок(том,пэт) .
Yes
```

Вариант 2

Переставим правила pr1 и pr2 местами:

```
предок2(X,Y) :-
    родитель(X,Z) ,
    предок2(Z,Y) .
```

```
предок2(X,Y) :-
    родитель(X,Y) .
```

Прежний вызов дает прежний ответ.

```
?- предок2(том,пэт) .
Yes
```

На самом деле в этом варианте вычисления продолжаются дольше, чем в первом варианте. Чтобы это заметить, можно сделать трассировку вызываемых предикатов с помощью встроенного предиката `trace`.⁵ Сначала вызываем

```
?- trace(предок) , trace(родитель) .
```

А потом вызвать

```
?- предок(том,пэт) .
?- предок2(том,пэт) .
```

Вариант 3

Вернемся к первому варианту, но в правой части правила pr2 две цели переставим местами.

⁵ Более подробно о тестировании и трассировке смотрите в приложении.

```
предок3 (X, Y) :-
    родитель (X, Y) .
```

```
предок3 (X, Y) :-
    предок3 (Z, Y) ,
    родитель (X, Z) .
```

Прежний вызов дает прежний ответ.

```
?- предок3 (том, пэт) .
Yes
```

Но вызов `предок3 (лиз, джим)` переполняет стек Пролога при рекурсии. Хотя эта цель благополучно вычисляется в варианте 1 и 2.

Вариант 4

Теперь в варианте 1 переставим и правила, и цели в теле правила `pr2`.

```
предок4 (X, Y) :-
    предок4 (Z, Y) ,
    родитель (X, Z) .
```

```
предок4 (X, Y) :-
    родитель (X, Y) .
```

Теперь даже первоначальный вызов `предок4 (том, пэт)` переполняет стек Пролога при рекурсии. И это понятно. Вызов цели `предок4 (том, пэт)` приводит к цели `предок4 (Z, пэт)`, эта цель, в свою очередь, вызывает цель `предок4 (Z1, пэт)`, которая отличается от предыдущей только новым именем переменной, и подобные вызовы далее повторяются без конца: `предок4 (Z2, пэт)`, `предок4 (Z3, пэт)`,...

Данный пример показывает, как Пролог-система может пытаться решить задачу таким способом, при котором решение никогда не будет достигнуто, хотя оно существует. Такая ситуация

не является редкостью при программировании на Прологе (как и на других языках).

Рекомендации

- в первую очередь применяйте самое простое правило (где отсутствует рекурсия);
- избегайте левой рекурсии⁶.

⁶ Левая рекурсия — это рекурсия, когда предикат сначала вызывает себя, а только потом другие предикаты (как в вариантах 3 и 4 в правиле `pr2` предикат `предок` в первую очередь вызывает себя).

3 СТРУКТУРЫ ДАННЫХ

3.1 Предикат унификации

Есть несколько встроенных предикатов в Прологе, которые используются в инфиксной записи. Наиболее употребительный такой предикат — **унификация**, для его записи используется символ «= \Rightarrow ». Следующий вызов выдает истину

?- <терм 1> = <терм 2>.

если эти два терма можно унифицировать. В соответствии с определением унификации в разделе 1.11 два терма, не содержащие переменных, унифицируемы, если они тождественны. Если же хотя бы один из термов содержит переменную, то эти термы унифицируемы только тогда, когда существует такая подстановка значений вместо имеющихся переменных, под действием которой эти термы становятся тождественными.

С любой успешной унификацией термов, содержащих переменные, связан побочный эффект. Переменные после успешной унификации термов становятся конкретизированными, т.е. они получают значения в виде некоторых термов.

Лексическая область определения имен переменных представляет собой одно предложение. Это означает, например, что если имя X встречается в двух предложениях, то оно обозначает две разные переменные⁷. Но каждое вхождение X в одном и том же предложении (или в одной и той же цели) соответствует одной и той же переменной. Поэтому если одно из вхождений какой-то переменной в цели получило какое-то значения, то эта переменная и во всех других вхождениях имеет то же самое значение.

Приведем примеры унификации:

⁷ Это означает, что в Прологе нет глобальных переменных, часто используемых в процедурном программировании. А как же передавать информацию из одного правила в другое? Для этого и используется унификация.

?- X=5.

X = 5

Yes

?- g(X)=g(f(3,a)).

X = f(3, a)

Yes

?- g(5,X)=g(Y,f(a,b)).

X = f(a, b)

Y = 5

Yes

?- 6=6.

Yes

?- 6=2+4.

No

?- X=Y.

X = _G156 % это внутреннее обозначение переменной для
Пролога

Y = _G156

Yes

?- X=5,Y=6,X=Y.

No

?- X=1,X=2.

No

?- _ = 5, _ = 6. % анонимная переменная унифицирует-
ся с любым термом

Yes

Предикат, который является отрицанием унификации, обозначается «\=». Вызов

?- <терм 1> \= <терм 2>.

выдает истину, если термы не унифицируемы. Например,

?- a \= b.

Yes

Отметим, что предикаты $=$ и \neq можно использовать и в префиксной форме:

```
?- =(X, 5) .
```

```
X = 5
```

```
Yes
```

3.2 Арифметические выражения

В Прологе имеются функторы (имена составных термов), которые используются в инфиксной форме. К ним относятся имена основных арифметических операций:

+	сложение
-	вычитание
*	умножение
/	деление
^ (или **)	возведение в степень
//	целочисленное деление
mod	остаток от деления

Все эти функторы можно использовать и в префиксной форме, например `// (X, 5)`.

Есть также функции с числовыми аргументами и числовым результатом:

```
abs(X)
max(X, Y)
min(X, Y)
random(N) => 0 ≤ i ≤ N (N, i - целые)
integer(X) - округление X до ближайшего целого
floor(R) => N (N ≤ R < N+1, пол - наибольшее целое ≤ R)
ceil(R) => N (N-1 < R ≤ N, потолок - наименьшее целое ≥ R)
sqrt(X)
sin(X) - углы в радианах
cos(X)
tan(X)
asin(X)
acos(X)
atan(X)
```



```

log(X)  ≡ ln X
log10 (X)
exp(X)  ≡ e^X
pi = 3.14159
e = 2.71828

```

Термы π и e можно рассматривать как нуль-местные функции.

Применяя арифметические операции и функции к переменным и константам (числам и атомам) и используя при необходимости скобки, можно сконструировать составные термы, которые называются **арифметическими выражениями**. Примерами арифметических выражений являются следующие термы:

```

-2 +sqrt (b^2-4*a*c) ;
(X-1) // ( (X+Y-1) ^2+1) .

```

Для некоторых арифметических выражений можно вычислить численное значение, скажем, если переменные X и Y конкретизированы целыми числами, то выражение $(X-1) // ((X+Y-1)^2+1)$ имеет вполне определенное целое значение. Если же переменные, входящие в арифметическое значение, еще не имеют значений, то и значение арифметического выражения не определено. Численного значения выражение $-2+\sqrt{b^2-4*a*c}$ не имеет, поскольку содержит атомы. Это выражение просто сложный терм и структуры данных такого вида можно использовать при программировании символьных вычислений.

Как вычислить значение арифметического выражения (в тех случаях, когда это возможно)? «Наивный» вызов

```

?- X=1+2.
X = 1+2
Yes

```

ничего не дает, поскольку в этом случае выражение $1+2$ рассматривается просто как терм, который успешно унифицируется с переменной X . Точно так же вычисляются и следующие вызовы:

```
?- pi=X.
X = pi
Yes
```

```
?- X= max(1,1) .
X = max(1, 1)
Yes
```

```
?- X=log10(10) .
X = log10(10)
Yes
```

Для вычисления арифметических значений специально используется предопределенная операция `is`. Операция `is` вынуждает систему выполнить вычисление, поэтому правильный способ вычисления арифметического выражения состоит в следующем:

```
?- X is 1+2.
X = 3
Yes
```

При этом операция сложения была выполнена с помощью специальной процедуры, которая связана с операцией `is`. Более сложные вычисления:

```
?- X is 5/2, Y is 5//2, Z is (X+Y+0.5)*2.
X = 2.5
Y = 2
Z = 10
Yes
```

Порядок выполнения операций при вычислении арифметических выражений определяется приоритетами операций и возможными скобками, входящими в выражение. Приоритет операций (а также коммутативность, ассоциативность и дистрибутивность) соответствует обычным правилам математики.

Нельзя рассматривать операцию `is` как оператор присваивания (необходимый оператор в процедурных языках). На самом деле — это предикат, вычисление которого связано с побочным эффектом. Но и побочный эффект нельзя описать в терминах присваивания.

Во-первых, слева от `is` может стоять не только переменная, но и число. В этом случае вычисленное значение правого операнда (арифметического выражения) сравнивается с данным числом и выдается Yes или No.

```
?- 5 is 5+1.
```

```
No
```

```
?- 5 is 4+1.
```

```
Yes
```

```
?- 3+2 is 4+1.
```

```
No
```

Во-вторых, невозможно переменной, имеющей значение, с помощью `is` присвоить новое значение (и любым другим способом). Примеры:

```
?- X = 2, X is X+1.
```

```
No
```

```
?- X is 3, X is 0.
```

```
No
```

Первый пример показывает, как не надо программировать на Прологе (а это традиционный способ в процедурных языках).

Таким образом, предикат `is` сначала вызывает вычисление правого операнда, а потом производит унификацию полученного значения и левого операнда.

При выполнении арифметических операций возникает необходимость сравнивать числовые значения. Например, с помощью следующей цели может быть выполнена проверка того, больше ли произведение чисел 277 и 37 по сравнению с числом 10000:

```
?- 277*37 > 10000.
```

```
Yes
```

Обратите внимание на то, что операция «>», как и операция `is`, вынуждают систему вычислять арифметические выражения (но в этом случае вычисляются оба операнда операции — левый и правый).

Ниже перечислены встроенные предикаты сравнения, которые обычно употребляются в инфиксной форме.

- $X > Y$, X больше Y .
- $X < Y$, X меньше Y .
- $X \geq Y$, X больше или равно Y .
- $X \leq Y$, X меньше или равно Y .
- $X =:= Y$, Значения X и Y равны.
- $X \neq Y$, Значения X и Y не равны.

Следует отметить, что предикат унификации и предикат проверки на равенство « $=:=$ » отличаются друг от друга; например, они по-разному ведут себя в целях $X=Y$ и $X=:=Y$. Первая цель вызывает унификацию термов X и Y , и если объекты унифицируются, то это может привести к конкретизации некоторых переменных в термах X и Y . В этом случае вычисление не выполняется. С другой стороны, операция $=:=$ вызывает вычисление арифметических выражений, но не может стать причиной какой-либо конкретизации переменных. Это отличие можно проиллюстрировать на следующих примерах:

```
?- 1+2 =:= 2+1.
Yes
```

```
?- 1+2 = 2+1.
No
```

```
?- 1+A = B+2.
A = 2
B = 1
Yes
```

3.2.1 Примеры программ с числами

Пример 1. Составьте предикат гипотенуза, который по двум катетам прямоугольного треугольника вычисляет его гипо-

тенузу. Воспользуемся теоремой Пифагора и встроенной функцией `sqrt` для вычисления квадратного корня:

```
гипотенуза(X,Y,Z):- Z is sqrt(X^2 + Y^2).
```

Программа корректно вычисляет гипотенузу, но если мы попробуем при ее помощи найти один из катетов, то убедимся, что процедура работает не вполне правильно. Чтобы избежать этого, добавим проверку того, что первые два аргумента предиката — положительные числа, для чего используем встроенный предикат `number` и сравнение с нулем:

```
гипотенуза(X,Y,Z):- number(X), X>0, number(Y), Y>0,
                     Z is sqrt(X**2 + Y**2).
```

Теперь наша программа работает корректно:

```
?- гипотенуза(3,4,X).
X = 5
Yes
```

```
?- гипотенуза(3,a,X).
No
```

```
?- гипотенуза(3,X,5).
No
```

Пример 2. Напишите предикат, который по двум парам чисел — длинам катетов прямоугольных треугольников — определяет величину меньшей из гипотенуз. Воспользуемся предикатом `гипотенуза`, разобранным выше, и встроенной функцией `min`:

```
мин_гип(A1,B1,A2,B2,Min):-
    гипотенуза(A1,B1,C1),
    гипотенуза(A2,B2,C2),
    Min is min(C1,C2).
```

Запросы к интерпретатору Пролога могут выглядеть так:

```
?- мин_гип(3, 4, 8, 6, X) .
X = 5
Yes
```

```
?- мин_гип(3, 4, Y, 6, X) .
No
```

Пример 3. Факториалом натурального числа n называют произведение всех целых чисел от 1 до n включительно. Для записи факториала числа n используют обозначение $n!$. По определению считается, что $0! = 1$. Имеем рекурсивное определение:

$$n! = n \times (n-1) \times (n-2) \times \dots \times 2 \times 1 = n \times (n-1)!$$

Следующий предикат вычисляет факториал числа (`integer` — встроенный предикат; проверяет: является ли аргумент целым числом):

```
факториал(0, 1) .
Факториал(N, R) :- integer(N), N > 0, N1 is N-1,
                    факториал(N1, R1), R is N*R1.
```

Первое правило (так называемый терминальный случай, то есть тот момент выполнения предиката, когда он перестает вызывать сам себя) гласит, что факториал нуля равен единице. Второе правило есть просто запись определения факториала: результат R получается умножением числа N на факториал числа, на единицу меньшего. Оно будет срабатывать при всех $n > 1$ потому, что интерпретатор Пролога просматривает программу сверху вниз и переходит к следующему правилу или факту только в том случае, когда он не может выполнить текущее правило.

Замечание 1. Для трех предикатов `N1 is N-1`, `факториал(N1, R1)`, `R is N*R1` во втором правиле нельзя изменить порядок их расположения (хотя с точки зрения логики это было бы допустимо). Изменение порядка выполнения данных предикатов приводит к ситуации, когда при вычислениях арифметического выражения требуется значение еще не конкретизированной переменной. Вместе со встроенным предикатом `is` в программу было введено отношение, возможность выполнения

которого зависит от порядка обработки, и поэтому процедурные соображения выступили на первый план.

Замечание 2. Второе правило в следующем виде не будет работоспособно, так как аргумент $N-1$ предиката факториал (при конкретном числе вместо N) сам по себе без `is` вычисляться не будет.

```
Факториал(N,R) :- integer(N), N>0,
                  факториал(N-1,R1), R is N*R1.
```

3.2.2 Дифференцирование

Сейчас рассмотрим, как можно в Прологе выполнять символьные преобразования над арифметическими выражениями. Отметим, что в Прологе это осуществляется очень просто.

Задача. Написать предикат `dv(+Term1,+Atom,-Term2)`, где `Term2` есть результат дифференцирования `Term1` по математической переменной `Atom`. `Term1` — арифметическое выражение, составленное из атомов и чисел с использованием скобок, знаков операций `+`, `-`, `*`, `/`, `^` и функций `sin(x)`, `cos(x)`, `ln(x)` и `exp(x)`.

Логическая программа дифференцирования — просто набор соответствующих правил дифференцирования.

```
dv(X,X,1).
dv(Y,_,0) :-
    number(Y).
dv(Y,X,0) :-
    atom(Y),
    X \= Y.
dv(X^N,X,N*X^N1) :-
    number(N),
    N1 is N-1.
dv(X^A,X,A*X^(A-1)).
dv(A^X,X,ln(A)*A^X).
dv(sin(X),X,cos(X)).
dv(cos(X),X,-sin(X)).
dv(exp(X),X,exp(X)).
dv(ln(X),X,1/X).
dv(Y+Z,X,DY+DZ) :-
```

```

      dv (Y, X, DY) ,
      dv (Z, X, DZ) .
dv (Y-Z, X, DY-DZ) :-
      dv (Y, X, DY) ,
      dv (Z, X, DZ) .
dv (Y*Z, X, DY*Z+DZ*Y) :-
      dv (Y, X, DY) ,
      dv (Z, X, DZ) .
dv (Y/Z, X, (DZ*Y-DY*Z) / (Z*Z)) :-
      dv (Y, X, DY) ,
      dv (Z, X, DZ) .
dv (A, X, dv (A, X)) .

```

Последнее правило добавлено для того, чтобы Пролог не выдавал No, если он не сможет продифференцировать. В этом правиле $dv(A, X)$ — просто терм, который обозначает производную A по X. Проверим, как это работает:

```

?- dv (x^2+y^3+sin(x)+2^x+ln(x*x)+(x+1)*x, x, R) .
R=2*x^1+dv(y^3, x)+cos(x)+ln(2)*2^x+dv(ln(x*x), x)+((1+0)*x+1*(x+1))
Yes

```

Программа работает не вполне удовлетворительно. Во-первых, она не умеет дифференцировать сложные функции, в частности $\ln(x*x)$. Во-вторых, не может найти производную от y^3 по x. В-третьих, слишком сложное выражение получилось в результате дифференцирования.

Первые два обстоятельства требует более сложного анализа термов (а не просто задания их строения с помощью шаблонов левой части правил). Мы вернемся к этой задаче в разделе 5.1, когда познакомимся с встроенными предикатами, анализирующими структуру и значение терма.

Упрощение выражений после дифференцирования — это отдельная задача, которая должна решаться независимо от дифференцирования. Надо определять специальный предикат *simple*, который упрощал бы термы, в соответствии с математическими законами. Правила для *simple* должны соответствовать различным ситуациям: сложение с 0, умножение на 1, приведение подобных слагаемых и т.д.

3.3 Списки

3.3.1 Синтаксис и семантика списков

Список — это простая структура, широко используемая в нечисловом программировании. Список представляет собой последовательность, состоящую из любого количества элементов. Можно дать следующее неформальное рекурсивное определение списка:

- список может быть пустым (тогда он не содержит элементов) или
- список состоит из головы (это просто терм) и хвоста (это снова список).

Синтаксически список описывается с помощью рекурсивной структуры, функтор которой изображается точкой «.». Примеры:

`[]` — пустой список;

`.(a, [])` — список, состоящий из одного элемента — атома `a`, голова списка — элемент `a`, хвост — пустой список;

`.(a, .(b, .(c, [])))` — список, состоящий из трех элементов `a`, `b` и `c`, голова списка — элемент `a`, хвост — список `.(b, .(c, []))`.

Такое представление используется в системе Пролог в основном для внутренних целей — внешнее представление списка `.(a, .(b, .(c, [])))` выглядит как `[a, b, c]`. Оно предпочтительнее для представления последовательности.

Примеры:

```
?- .(1, .(2, .(3, []))) = X.
X = [1, 2, 3]
Yes
```

```
?- [a, [a], [[a]]] = [X, Y, Z].
X = a
Y = [a]
Z = [[a]]
Yes
```

```
?- [a,b,c, 4,5] = .(X,Y) .
X = a
Y = [b, c, 4, 5]
Yes
```

Второй пример говорит о том, что элементы списка могут быть сами списками. Третий пример показывает, как с помощью унификации из списка можно выделить (выбрать) голову и хвост.

Чтобы можно было выбирать голову и хвост списка, записанного в квадратных скобках, и не использовать функтор «.», в языке Пролог предусмотрено еще одно дополнение к списковой записи: вертикальная черта, которая разделяет голову и хвост. Это показано в следующем примере:

```
?- [a,b,c] = [X|Y] .
X = a
Y = [b, c]
Yes
```

Обозначение с помощью вертикальной черты фактически является более общим, поскольку вертикальная черта может стоять в списке после любого количества элементов, а не только после головы. Следующий пример показывает возможное использование вертикальной черты:

```
?-
L=[1,2,3,4], L=[H|T], L=[X,Y|T1], L=[X,Y,Z|T2], L=[_,_,_,
_|T3] .

L = [1, 2, 3, 4]
H = 1
T = [2, 3, 4]
X = 1
Y = 2
T1 = [3, 4]
Z = 3
T2 = [4]
T3 = []

Yes
```

Приведем еще несколько примеров унификации со списками:

Список 1	Список 2	Конкретизация переменных
[X,Y,Z]	['орел','курица','утка']	X='орел', Y='курица', Z='утка'
[7]	[X Y]	X=7, Y=[]
[1,2,3,4]	[X,Y Z]	X=1, Y=2, Z=[3,4]
[1,2]	[3 X]	нет унификации
[1 [2]]	[X Y]	X=1, Y=[2]
[1,2]	[X Y]	X=1, Y=[2]
[1,[2]]	[X,Y]	X=1, Y=[2]

Отметим один из видов списков — список кодов ASCII символов. Такие списки могут быть представлены в виде **строк**, например,

```
?- Y = "Борис".
Y = [193, 238, 240, 232, 241]
Yes
```

```
?- "Борис" = [X,Y|"рис"] .
X = 193
Y = 238
Yes
```

3.3.2 Некоторые предикаты для списков

Рассмотрим основные предикаты для списков: проверка на вхождение элемента в список, добавление элемента в список, соединение двух списков и др.

Является ли данный терм списком?

Объект данных является списком, если его можно унифицировать с пустым списком или с непустым списком. Получается два правила:

```
listp([]).
listp([_|_]).
```

Пример использования данного предиката будет приведен немного позже. В языке SWI-Prolog существует встроенный предикат

`is_list`, который осуществляет указанную проверку; отличие работы этих двух предикатов видно из следующих вызовов:

```
?- listp(X) .
X = [] ;
X = [_G231|_G232] ;
No
```

```
?- is_list(X) .
No
```

Проверка принадлежности элемента списку.

Указанный элемент списка ищется на самом верхнем уровне: если элементы списка сами являются списками, то поиск внутри этих внутренних списков не осуществляется. В соответствии с этими соглашениями можно сформулировать следующий алгоритм.

Пусть список не пуст и имеет вид $[H|T]$. Если искомый элемент есть в списке, то он либо совпадает с головой H , либо принадлежит хвосту списка T , а там его можно искать снова рекурсивно. Получаем правила:

```
member1(H, [H|_]) .
member1(H, [_|T]) :-
    member1(H, T) .
```

В первом правиле анонимная переменная указывает, что хвост списка нам не нужен, поскольку искомый элемент является головой списка. Второе правило будет использоваться системой, если первое правило не применимо, тогда, конечно, значение головы списка безразлично.

Примеры:

```
?- member1(5, [1, 2, 3, 4, 5]) .
Yes
```

```
?- member1(5, [1, 2, 3, 4]) .
No
```

Мы используем обозначение `member1`, а не `member`, поскольку имеется встроенный предикат с данным именем и выполняющий требуемое действие (и его определение такое же, как у `member1`). Системе Пролог «не нравится», когда определяют предикат, имя которого совпадает с именем существующего предиката.

Мы программировали предикат `member1`, предполагая, что оба аргумента предиката должны быть входными. Но получили больше, чем ожидали.⁸

```
?- member1(X, [1, 2, 3]).
X = 1 ;
X = 2 ;
X = 3 ;
No
```

При таком запуске предикат выдает все элементы списка по очереди.

```
?- member(1, X).
X = [1|_G261] ;
X = [_G260, 1|_G264] ;
X = [_G260, _G263, 1|_G267]
Yes
```

А сейчас предикат создает списки (их бесконечно много), содержащие указанный элемент (остальные элементы обозначены внутренними переменными Пролога). Таким образом, типы аргументов предиката можно описать как `member1(?X, ?Y)`.

Рассмотрим теперь задачу поиска элемента в списке, но не только на первом уровне. Сейчас элементы могут быть списками, и при необходимости следует искать заданный элемент внутри этих списков. К двум правилам `member1` добавляется новое пра-

⁸ Древнегреческие математики называли такие вещи «поризмами» — плоды сбитые ветром. При программировании на Прологе предикатов со списками часто получается, что кроме запуска предиката в прямом направлении (от входных аргументов к выходным), мы видим, что предикат можно использовать и для решения обратной задачи.

вило: ищем элемент в голове, если он список (здесь нам понадобится предикат `listp`).

```
?- member2(1, [2, 3, 4, [[[[1]]], 5]]).
Yes
```

Добавить элемент в начало списка.

Следующий предикат, представленный в виде одного факта, решает данную задачу:

```
добавить(Н, Т, [Н|Т]).
```

На самом деле нет необходимости явно определять этот предикат в тех задачах, когда требуется добавить какой-то элемент в начало списка. Как можно обойтись без предиката `добавить` мы увидим при решении следующей задачи.

Соединение двух списков.

Напишем предикат `append(+X, +Y, -Z)`, который будет истинен тогда и только тогда, когда список `Z` получается в результате соединения (конкатенации) списков `X` и `Y`: список `Z` содержит сначала элементы списка `X`, а потом следуют элементы списка `Y`.

Рассмотрим рекурсивный алгоритм. Рекурсия будет идти по списку `X`:

- 1) если список `X` пуст, то и список `Z` равен `Y`;
- 2) иначе список `X` имеет вид `[Н|Т]`, для того чтобы получить результат соединения списков `X` и `Y`, соединим список `Т` и `Y` и в полученный список добавим элемент `Н` в начало.

Получаем правила:

```
append1([], Y, Z) :-
    Z=Y.
append1([Н|Т], Y, Z) :-
    append1(Т, Y, L),
    добавить(Н, L, Z).
```

Проверим:

```
?- append([1,2,3],[4,5],Z) .
Z = [1, 2, 3, 4, 5] ;
No
```

Но это определение предиката сложно. Упростим его. Во-первых, в первом правиле переменные Y и Z должны иметь одно значение во всех своих вхождениях. Поскольку любая переменная Пролога в правиле не меняет своего значения (вспомним, что никаких присваиваний в Прологе нет), то и нет необходимости вводить разные переменные для обозначения одной сущности. Во-вторых, заметим, что во втором правиле, если Z можно унифицировать со списком $[H|L]$, то можно сразу вместо Z писать $[H|L]$. Получаем вариант предиката (этот как раз то определение, которое имеет встроенный предикат `append`):

```
append([],Y,Y) .
append([H|T],Y,[H|L]):-
    append(T,Y,L) .
```

Проверим:

```
?- append([1,2,3],[4,5],Z) .
Z = [1, 2, 3, 4, 5] ;
No
```

Оказывается, мы снова получаем дополнительные возможности. Хотя мы программировали предикат для типов аргументов `append(+X,+Y,-Z)`, мы имеем типы `append(?X,?Y,?Z)`. Действительно,

```
?- append([1,2,3],[4,5],[1,2,3,4,5]) .
Yes
```

```
?- append(X,[4,5],[1,2,3,4,5]) .
X = [1, 2, 3] ;
No
```

```
?- append([1,2,3],Y,[1,2,3,4,5]).
Y = [4, 5] ;
No
```

```
?- append(X,Y,[1,2,3]).
X = []
Y = [1, 2, 3] ;
```

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

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

```
X = [1, 2, 3]
Y = [] ;
No
```

```
?- append(X,Y,Z).
X = []
Y = _G220
Z = _G220 ;
```

```
X = [_G304]
Y = _G220
Z = [_G304|_G220]
Yes
```

Таким образом, предикат `append` умеет не только соединять, но и расщеплять списки, а в последнем вызове он создает бесконечное множество списков, удовлетворяющих требуемому соотношению.

Определение количества элементов в списке.

Количество элементов списка легко определить рекурсивно: если к количеству элементов в хвосте списка добавить 1, то мы получаем количество элементов в исходном списке, учитывая и голову. Надо, конечно, не забыть про пустой список.

Определение встроенного предиката `length` (длина) следующее:


```
length([],0).
length(_|T,N)
    length(T,N1),
    N is N1+1.
```

Проверим:

```
?- length([a,b,c,d],X).
X=4;
No
```

Сортировка списка простым обменом.

Напомним алгоритм сортировки простым обменом:

1) просматриваем список, пока не обнаружим два соседних элемента, которые расположены не в нужном порядке, — меняем их местами;

2) заканчиваем алгоритм, если список оказался уже отсортированным, если же нет, то возвращаемся к шагу 1.

Этот алгоритм легко записывается в рекурсивном виде (сортировка по не убыванию):

```
sort1(L,R):-
    append(X,[A,B|Y],L),
    A>B,
    append(X,[B,A|Y],L1),
    sort1(L1,R).
sort1(L,L).
```

Проверим:

```
?- sort1([3,4,5,6],R).
R = [3, 4, 5, 6]
Yes
```

Первый `append` расщепляет список таким образом, чтобы выделить два рядом стоящих элемента *A* и *B*. Если условие *A>B* не выполняется, то происходит возврат и `append` расщепляет список снова. После нахождения пары элементов, которые нарушают требуемый порядок, работает второй `append`, который те-

перь соединяет списки (но элементы А и В стоят уже в нужном порядке). Результат второго вызова `append` теперь сортируется снова.

Второе правило, которое оставляет список неизменным, будет работать только тогда, когда первое правило постигнет неудача. А это произойдет лишь в том случае, если в списке уже не найдутся неправильно упорядоченные соседние элементы.

3.4 Структуры

Структуры данных, наряду с унификацией и перебором с возвратом, являются мощными инструментами программирования. Мы уже рассмотрели два класса структур: арифметические выражения и списки. Рассмотрим еще несколько примеров структур и манипулирования с ними.

3.4.1 Выборка структурированной информации из базы данных

Любую базу данных можно естественным образом представить на языке Пролог в виде множества фактов. Например, базу данных о семьях можно представить так, что для описания каждой семьи будет применяться только одно предложение. Каждая семья представляется предикатом `семья` с тремя аргументами: муж, жена и дети. Поскольку в разных семьях имеется различное количество детей, информацию о детях естественно представить в виде списка, который позволяет включать в него любое количество элементов. Информация о каждом человеке, в свою очередь, представляется в виде структуры, состоящей из трех компонентов: имя, пол, возраст. Следующий пример описывает три семьи:

```
семья(персона(том, мужчина, 45), персона(энн, женщина, 44),
      [персона(пэт, женщина, 22), персона(джим, мужчина, 18)]).
```

```
семья(персона(боб, мужчина, 35), персона(пэм, женщина, 30),
      [персона(кэт, женщина, 13)]).
```

```
семья(персона(генри, мужчина, 24), персона(лиз, женщина, 24), []).
```

Таким образом, рассматриваемая база данных может состоять из последовательности фактов, подобных этому, в котором описаны все семьи.

Пролог фактически является языком, который очень хорошо приспособлен для выборки требуемой информации из подобной базы данных (такие базы данных называются «реляционными»). Замечательным свойством этого языка является то, что он позволяет ссылаться на объекты, фактически не задавая все компоненты этих объектов. Пролог предоставляет возможность указывать структуру интересующих нас объектов и оставлять конкретные компоненты в этой структуре не заданными или заданными лишь частично.

Например, можно определить жену Тома с помощью вызова:

```
?- семья(персона(том,_,_), персона(X,_,_),_) .
X = энна
Yes
```

Например, можно найти все семьи с одним ребенком — девочкой:

```
?- семья(Y,Z,[D]), D=персона(_,женщина,_) .
Y = персона(bob, мужчина, 35)
Z = персона(пэм, женщина, 30)
D = персона(кэт, женщина, 13) ;
No
```

Чтобы найти всех замужних женщин, имеющих детей, можно сформулировать такой вопрос:

```
?- семья(_,X,[_|_]) .
X = персона(энна, женщина, 44) ;
X = персона(пэм, женщина, 30) ;
No
```

Из этих примеров следует вывод, что интересующие нас объекты можно определять не только по их содержимому, но и указывая их структуру.

Кроме того, существует возможность предусмотреть набор предикатов, которые будут служить в качестве утилит, позволяющих повысить удобство взаимодействия с базой данных. Некоторые полезные предикаты для рассматриваемой базы приведены ниже.

Предикат «быть мужем»:

```
муж(X) :- семья(персона(X, _, _), _, _).
```

```
?- муж(X).
```

```
X = том ;
```

```
X = bob ;
```

```
X = генри ;
```

```
No
```

Предикат «семейная пара»:

```
семейная_пара(X, Y) :-
```

```
    семья(персона(X, _, _), персона(Y, _, _), _).
```

```
?- семейная_пара(X, Y).
```

```
X = том
```

```
Y = энна ;
```

```
X = bob
```

```
Y = пэм ;
```

```
X = генри
```

```
Y = лиз ;
```

```
No
```

Предикат «родитель»

```
родитель(X, Y) :-
```

```
    семья(персона(X, _, _), _, L),
```

```
    member(персона(Y, _, _), L).
```

```
родитель(X, Y) :-
```

```
    семья(_, персона(X, _, _), L),
```

```
    member(персона(Y, _, _), L).
```

Кто родители Джима?

```
?- родитель (X, джим) .
X = том ;
X = энна ;
No
```

Теперь семейные пары, имеющие детей, можно определить с помощью запроса:

```
?- семейная_пара (X, Y) , родитель (X, _) .
X = том
Y = энна ;

X = том
Y = энна ;

X = bob
Y = пэм ;
No
```

Два одинаковых ответа, $X = \text{том}$, $Y = \text{энна}$, даны потому, что у Тома двое детей.

3.4.2 Рекурсивные структуры

Хороший пример встроенных рекурсивных данных — это список. Но программист может сам ввести новые рекурсивные структуры.

Рассмотрим использование рекурсивных структур при представлении простых электрических цепей. Пусть цепи содержат только резисторы, соединенные последовательно или параллельно.

Цепь, состоящую из одного резистора, будем представлять числом — номиналом резистора. Два резистора $R1$ и $R2$, соединенные последовательно, обозначим термом $\text{succ}(R1, R2)$, а параллельно — термом $\text{para}(R1, R2)$. Используя эти обозначения рекурсивно, мы можем определить произвольные цепи, состоящие только из последовательно и параллельно соединенных сопротивлений. Например, цепь из 4 резисторов:

```
para(1.0, succ(para(2.0, 3.0), 4.0)).
```

Цепь, состоящую только из последовательно или параллельно соединенных резисторов, можно упростить до одного резистора. Определим предикат `simple(+X, -R)` (X — исходная электрическая цепь, R — резистор — результат упрощения цепи):

```
simple(X,X):-                               % цепь состоит из одного резистора
    number(X).
simple(succ(X,Y),Z):-                       % последовательное соединение двух подцепей
    simple(X,R1),
    simple(Y,R2),
    Z is R1+R2.
simple(para(X,Y),Z):-                       % параллельное соединение двух подцепей
    simple(X,R1),
    simple(Y,R2),
    Z is (R1*R2)/(R1+R2).

?- simple(para(1.0, succ(para(2.0, 3.0), 4.0)), X).
X = 0.838710
Yes
```

Бинарные деревья можно задать с помощью тернарного функтора `tree(Left, Root, Right)`, где `Root` — элемент, находящийся в вершине, а `Left` и `Right` — соответственно левое и правое поддерево. Пустое дерево изображается атомом `nil`. Если дерево состоит из одной вершины, скажем число 5, и имеет пустые левые и правые поддерева, то в этом случае дерево будет представлено термом `tree(nil, 5, nil)`. Дерево на рис. 4 имеет левое поддерево

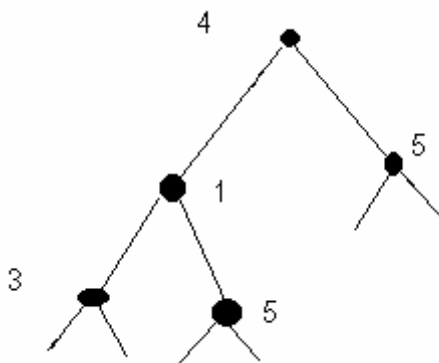


Рис. 4 — Бинарное дерево

`tree(tree(nil, 3, nil), 1, tree(nil, 5, nil))` и правое поддерево `tree(nil, 5, nil)`, а все само представлено термом

```
tree(tree(tree(nil, 3, nil), 1, tree(nil, 5, nil)),
4, tree(nil, 5, nil))
```

Пусть p — некоторый предикат для работы с деревьями (деревья представлены первым аргументом предиката), тогда нужно иметь два правила — для пустого дерева и для дерева в общем виде:

```
p(nil, ...) :-
```

```
p(tree(L, X, R), ...) :-
```

Чтобы найти сумму всех вершин бинарного дерева, используется простой рекурсивный алгоритм (два правила):

- Если дерево пустое (`nil`), то ответ 0.
- Если дерево не пустое, то имеет вид `tree(L, X, R)`, и для того, чтобы найти сумму S элементов во всем дереве, рекурсивно находим сумму в поддереве L (пусть это будет $S1$), рекурсивно находим сумму в поддереве R (пусть это будет $S2$) и вычисляем S как сумму трех величин X , $S1$ и $S2$.

```
сумма(nil, 0).
сумма(tree(L, X, R), S) :-
    сумма(L, S1),
    сумма(R, S2),
    S is X+S1+S2.
```

Вот предикат, который проверяет, является ли элемент X одной из вершин дерева:

```
tree_member(X, tree(_, X, _)).
tree_member(X, tree(Left, _, _)) :-
    tree_member(X, Left).
tree_member(X, tree(_, _, Right)) :-
    tree_member(X, Right).
```

Можно использовать упорядоченные бинарные деревья для сортировки списков.

```
% tree_sort(+X,-Y)
% X - исходный список, результат Y - упорядоченный список
tree_sort(X,Y):-
    make_tree(X,Z),
    flat(Z,Y).

% make_tree(+X, -Y) - создание упорядоченного дерева Y из списка X
make_tree([],nil).
make_tree([H|T],Z):-
    make_tree(T,Y),
    insert(H,Y,Z).

% insert(+N,+X,-Y) - вставка элемента N в упорядоченное дерево X
insert(N,nil,tree(nil,N,nil)).
insert(N,tree(L,Root,R),tree(L1,Root,R)):-
    N <= Root,
    insert(N,L,L1).
insert(N,tree(L,Root,R),tree(L,Root,R1)):-
    N > Root,
    insert(N,R,R1).

% flat(+X,-Y) - разглаживание дерева X в список Y
flat(nil,[]).
flat(tree(L,N,R),Z):-
    flat(L,L1),
    flat(R,R1),
    append(L1,[N|R1],Z).
```

Проверим:

```
?- make_tree([8,10,6,5],X).
X = tree(nil, 5, tree(nil, 6, tree(tree(nil, 8, nil), 10, nil)))
Yes

?- tree_sort([8,10,6,5],X).
X = [5, 6, 8, 10] ;
No
```

3.5 Модификация синтаксиса (операторная запись)

Прологовские **операторы** суть функторы и имена предикатов (унарных и/или бинарных), записанных до, после или между аргументами.

Некоторые встроенные бинарные операторы (имена арифметических операций) обладают ассоциативностью. Это, в частности, дает возможность писать такие арифметические выраже-

ния, как $a+b+c$ или $a*b*c$, не используя скобок. С помощи унификации можно выяснить структуру таких термов:

?- $a+b+c=X+Y$.

$X = a+b$

$Y = c$

Yes

?- $a*b*c=X*Y$.

$X = a*b$

$Y = c$

Yes

Таким образом, термы $a+b+c$ и $a*b*c$ рассматриваются системой по умолчанию, как $(a+b)+c$ и $(a*b)*c$ соответственно. Говорят, что операторы $+$ и $*$ являются лево-ассоциативными. С другой стороны, терм a^b^c рассматривается системой по умолчанию, как $a^(b^c)$. Говорят, что он является право-ассоциативным:

?- $a^b^c=X^Y$.

$X = a$

$Y = b^c$

Yes

Если в выражении присутствует несколько различных бинарных операторов, то структура термов определяется в соответствии с расстановкой скобок и приоритетами операций. Примеры:

?- $a+b*c = X+Y$.

$X = a$

$Y = b*c$

Yes

?- $a+b*c = X*Y$.

No

?- $(a+b)*c = X*Y$.

$X = a+b$

$Y = c$

Yes

Как видим, приоритет операции $*$ больше приоритета операции $+$.

В данных примерах встроенными операторами являются предикат `=` и функторы `+`, `*` и `^`, но программист может ввести собственные операторы. Поэтому, например, в программе можно определить атомы `имеет` и `support` как инфиксные имена предикатов, а затем записывать в программе факты наподобие следующих:

```
том имеет лошадь .
пол support стол .
```

Эти факты полностью эквивалентны следующим:

```
имеет(том, лошадь) .
support(пол, стол) .
```

Программист может определять новые операции, вставляя в программу (обычно, одна из первых строчек) предложения специального типа, называемые **директивами** (компилятору). В данном примере имя предиката `имеет` должно быть определено с помощью следующей директивы:

```
:- op(600, xfx, имеет) .
```

Такая конструкция сообщает системе Пролог, что предусмотрено использование атома `имеет` в качестве суффиксного оператора, который имеет приоритет 600 и относится к типу `xfx`; этот тип характеризует некоторые инфиксные операторы. Такая форма спецификатора `xfx` указывает, что данное имя оператора, обозначенное как `f`, должно находиться между двумя операндами, обозначенными как `x`.

Следует отметить, что ни с одним оператором при определении не связывается каких-либо действий над данными. Операторы, как и функторы, обычно используются только для объединения объектов в структуры, для повышения наглядности программы.

Операторы представляются атомами. Приоритет оператора задается целым числом из диапазона 1–1200.

Типы операторов подразделяются на три группы, которые обозначаются спецификаторами типа, такими, как xfx . Эти три группы перечислены ниже.

- Инфиксные операторы: xfx , yfx , xfy .
- Префиксные операторы: fx , fy .
- Постфиксные операторы: xf , yf .

Спецификаторы выбираются таким образом, чтобы они отражали структуру термина: f представляет оператор, а x и y — операнды. Положение f относительно x и y говорит о том, где должен находиться оператор — между операндами, перед или после операнда. Кроме того,

- запись yfx означает левую ассоциативность

$$a\ f\ b\ f\ c\ f\ d \equiv ((a\ f\ b)\ f\ c)\ f\ d;$$
- запись xfy означает правую ассоциативность

$$a\ f\ b\ f\ c \equiv a\ f(b\ f\ c);$$
- запись xfx говорит, что оператор не обладает ассоциативностью; например `mod`, поэтому `X is 120 mod 50 mod 2` — синтаксическая ошибка;
- запись fx означает, что двукратное применение оператора f к операнду требует скобок, например тип оператора « \rightarrow » задан как fx , поэтому `--x` — синтаксически неправильная запись (требуется писать `-(-x)`).
- запись fy означает, что двукратное применение оператора f к операнду не требует скобок.

Наиболее важный критерий для определения оператора — это удобство чтения программы.

Пример:

```
:- op(750, xfx, знает) .
:- op(500, yfx, and) .
:- op(700, xf, fact) .
```

Теперь факты «знает» можно представлять в программе в инфиксной форме:

джейн знает бетти.
сюзан знает мэри.

?- X знает мэри.
X = сюзан
Yes

?- X=and(a,b) .
X = a and b
Yes

?- Y = fact(a) .
Y = a fact
Yes

Функторы and и fact стали операторами: первый из них — бинарным, второй — унарным постфиксным. Напоминаем, что речь идет лишь о синтаксисе.

Некоторые из встроенных операторов приведены ниже. Все эти операторы с помощью директивы op можно переопределить.

1200	xfx	-->, :-
1200	fx	:-, ?-
1100	xfy	;,
1050	xfy	->
1000	xfy	,
954	xfy	\\
900	fy	\+, not
700	xfx	<, =, =.., =@=, =:=, =<, ==, =\=, >, >=, \=, \==, is
600	xfy	:
500	yfx	+, -,
500	fx	+, -, ?, \
400	yfx	*, /, //,
300	xfx	mod
200	xfy	^

4 УПРАВЛЕНИЕ ПОВТОРЕНИЕМ В ПРОЛОГЕ

Единственный способ организовать повторение в Прологе — это воспользоваться рекурсией. С помощью рекурсии можно организовать перебор с возвратом — классический способ поиска нужных объектов в структурированной информации. Мы уже немного говорили о рекурсии ранее. Более полно будем рассматривать ее снова в разделе 4.4. А сейчас рассмотрим вопросы, связанные с ограничением перебора.

4.1 Отсечение

Последовательность одинаково плоха и для ума и для тела. Последовательность чужда человеческой природе, чужда жизни. До конца последовательны только мертвецы.

Олдос Хаксли

4.1.1 Определение отсечения

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».

Рассмотрим двухступенчатую функцию (рис. 5).

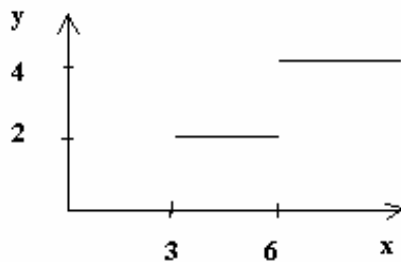


Рис. 5 — Двухступенчатая функция

Правило 1: если $X < 3$, то $Y = 0$.

Правило 2: если $3 \leq X$ и $X < 6$, 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:

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

$f(X, 2) :- 3 \leq X, X < 6.$

$f(X, 4) :- 6 \leq X.$

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

Эксперимент 1

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

Три правила, входящие в отношение f , являются взаимоисключающими, поэтому успех возможен самое большое 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 все равно они обречены на неудачу. О том, что в правиле 1 наступил успех, становится известно в точке, обозначенной словом «отсечение» (рис. 6). Для предотвращения бессмысленного перебора мы должны явно указать Пролог-системе, что не нужно осуществлять возврат из этой точки.

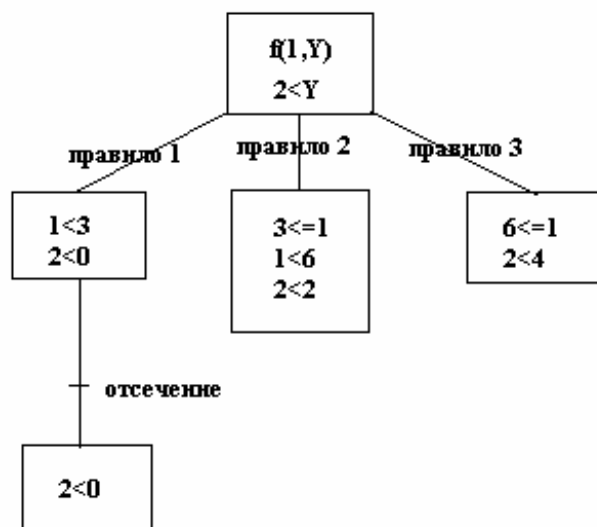


Рис. 6 — Применение отсечения

Мы можем сделать это при помощи конструкции **отсечения**. Отсечение записывается в виде символа «!», который вставляется между целями и играет роль некоторой псевдоцели (предикат без аргументов, который всегда истинен).

```
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. Если мы теперь спросим

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

то Пролог-система породит левую часть дерева, изображенного на рисунке 6. Эта ветвь потерпит неудачу на цели $2 < 0$. Система попытается сделать возврат, но вернуться она сможет не далее точки, помеченной символом «!». Альтернативные ветви, соответствующие правилу 2 и правилу 3, порождены не будут.

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 процедурный смысл программы, оставив при этом её декларативный смысл в неприкосновенности.

Эксперимент 2

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

```
?- f(7, Y).
Y=4
Yes
```

Перед тем, как был получен ответ, система пробовала применить все три правила. Вначале выясняется, что $X < 3$ не являет-

ся истиной ($7 < 3$ терпит неудачу). Следующая цель $3 = < X$ ($3 = < 7$ — 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 $6 = < X$ в правиле 3.

Теперь мы можем опустить в нашей программе те условия, которые обязательно выполняются при любом вычислении.

```
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) .
```

```
?- f(1, Y) .
Y=0;
Y=2;
Y=4;
No
```

Важно заметить, что в последней версии, в отличие от предыдущей, отсечения затрагивают не только процедурное поведение, но изменяют и декларативный смысл программы.

Назовем «целью-родителем» ту цель, которая унифицировалась 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.

Пример:

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

Будем считать, что это предложение активизировалось, когда некоторая цель G унифицировалась 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 больше не рассматриваются. Далее, цель G связывается теперь с этим предложением: любая попытка сопоставить G с головой какого-либо другого предложения пресекается.

Пример:

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

$C :- V.$

$A :- B, C, D.$

?- $A.$

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 . Отсечение повлияет только на цель C . Оно будет невидимо из цели A , и автоматический перебор все равно будет происходить в списке целей B, C, D вне зависимости от наличия отсечения в предложении, которое используется для достижения C .

4.1.2 Примеры программ с отсечением

Вычисление максимума

Введем предикат `максимум(+X, +Y, ?Z)`, который истинен, если `Z` равно наибольшему значению из чисел `X` и `Y`. Смысл каждого из правил данного предиката вполне очевиден.

```
максимум(X, X, X) .
максимум(X, Y, X) :- X > Y.
максимум(X, Y, Y) :- X < Y.
```

Посмотрим на реакцию интерпретатора Пролога на запросы, содержащие данный предикат.

```
?- максимум(20, 50, X) .
X = 50
Yes
```

```
?- максимум(100, 50, X) .
X = 100
Yes
```

```
?- максимум(X, 50, 100) .
X = 100
Yes
```

Последний ответ показывает, что наш предикат позволяет находить ответ на вопросы типа: «Каково должно быть число, чтобы максимум из искомого числа и числа 50 равнялся бы 100?». Как вы думаете, почему был получен ответ «No» на следующий запрос?

```
?- максимум(X, 50, 40) .
No
```

С помощью отсечения мы можем упростить программу:

```
максимум(X, Y, X) :-
    X > Y, !.
максимум(_, Y, Y) .
```

Теперь в отличие от первого определения, конечно, важен порядок правил. Но эта версия с декларативной точки зрения эквивалентна первоначальной версии.

Предикат `member`

Встроенный предикат `member` определен с помощью правил:

```
member(H, [H|_]) .
member(H, [_|T]) :-
    member(H, T) .
```

Этот предикат может вызываться с различными типами аргументов: `member(+H, +L)` и `member(-H, +L)`. Во втором случае предикат выдает все элементы списка.

При данном определении предикат `member` обладает некоторыми недостатками. Вернемся к программе из раздела 3.4.

```
семья(персона(том, мужчина, 45), персона(энн, женщина, 44),
      [персона(пэт, женщина, 22), персона(джим, мужчина, 18)]).

семья(персона(bob, мужчина, 35), персона(пэм, женщина, 30),
      [персона(кэт, женщина, 13)]).

семья(персона(генри, мужчина, 24), персона(лиз, женщина, 24), []).

семейная_пара(X, Y) :-
    семья(персона(X, _, _), персона(Y, _, _), _).

родитель(X, Y) :-
    семья(персона(X, _, _), _, L),
    member(персона(Y, _, _), L).

родитель(X, Y) :-
    семья(_, персона(X, _, _), L),
    member(персона(Y, _, _), L).

?- семейная_пара(том, Y), родитель(том, _).
Y = энн ;
Y = энн ;
No
```

Запрос выдал два одинаковых ответа. Постараемся понять, почему это произошло. После того как мы ввели «;», система пытается всюду, где это возможно, применить альтернативные правила.

Цель `семейная_пара(том, Y)` имеет только одно решение `Y=энн`. Цель `родитель(том, _)` приводит к выполнению правила

```
родитель(X, Y) :-
    семья(персона(X, _, _), _, L),
    member(персона(Y, _, _), L).
```

(Второе правило для предиката `родитель` неприменимо.) Таким образом, сначала вычисляется цель `семья(персона(том, _, _), _, L)`, в которой определяется список детей `L`. Эта цель не содержит никаких вариантов выполнения.

Но потом будет вычисляться цель `member(персона(Y, _, _), [персона(пэт, женщина, 22), персона(джим, мужчина, 18)])`, которая имеет два решения (два ребенка у Тома). Вот какова причина повторений ответа при исходном вызове!

Для того чтобы избежать повторений, необходимо так изменить предикат `member`, чтобы при вызове `member(-H, +L)` он давал только один ответ, независимо от того, сколько элементов в списке. Используем отсечение:

```
member1(H, [H|_]) :-!.
member1(H, [_|T]) :-
    member1(H, T).
```

Если первое правило будет применено при вычислении цели, то второе правило при повторении вызываться уже не будет. Проверим:

```
?- member1(2, [1, 2, 3]).
Yes
```

```
?- member1(X, [1, 2, 3]).
X = 1 ;
No
```

Теперь пусть в предикате `родитель` будет вызываться `member1`:

```
родитель (X, Y) :-
    семья (персона (X, _, _), _, L),
    member1 (персона (Y, _, _), L) .
```

```
родитель (X, Y) :-
    семья (_, персона (X, _, _), L),
    member1 (персона (Y, _, _), L) .
```

и вернемся к «критическому» вызову:

```
?- семейная_пара (том, Y), родитель (том, _) .
Y = энн ;
No
```

В данной программе использование отсечения в определении предиката `member` позволило избежать повторения ответа.

Замечание. В языке SWI-Prolog имеется предикат `memberchk (?X, +L)` – недетерминированная версия `member`:

```
?- memberchk (X, [1, 2, 3]) .
X = 1 ;
No
```

Добавление элемента списку, если он отсутствует (добавление без дублирования)

Задачу решает предикат с отсечением:

```
добавить (X, L, L) :-
    member (X, L), !.
добавить (X, L, [X|L]) .
```

Проверим:

```
?- добавить (5, [3, 4], X) .
X = [5, 3, 4] ;
No
```

```
?- добавить (5, [3, 4, 5], X) .
X = [3, 4, 5] ;
No
```

Если убрать отсечение:

```
добавить (X, L, L) :-
    member (X, L) .
добавить (X, L, [X|L]) .
```

то декларативный смысл предиката изменится:

```
?- добавить (5, [3, 4, 5], X) .
X = [3, 4, 5] ;
X = [5, 3, 4, 5] ;
No
```

4.2 Отрицание как неудача

Границы моего языка означают границы моего мира.

Людвиг Витгенштейн

Отрицание в Прологе реализовано прагматически и не имеет эквивалента в логике. Обсудим этот вопрос, следуя [6, с. 150–153].

Негативная информация

Информация о фактах, которые не являются истинными, или об отношениях, которые не соблюдаются, называется **негативной**. Обычно негативная информация не хранится в Пролог-программах в явной форме. Вместо этого считается, что вся информация, отсутствующая в текущем множестве фраз (предложений) ложна. Это эквивалентно предложению о том, что всегда имеет силу следующий принцип:

Если правило P не представлено в текущей программе, то считается, что представлено отрицание P .

Предположение о замкнутости мира

С практической точки зрения это означает, что интерпретатор не может отличить неизвестное предложение от доказуемо

неистинного предложения. Принцип, приведенный выше, известен как предположение **о замкнутости мира**. Множество предложений текущей программы называется *миром*. Это — замкнутый мир, поскольку интерпретатор ведет себя так, как будто бы в этом мире содержатся все возможные знания.

Тогда и только тогда, когда

Вследствие того, что предполагается замкнутость мира, множество предложений, определяющих отношение, имеет металингвистический смысл, несколько отличающийся от его смысла с позиций объектного языка. Предположим, например, что в программе представлено три предложения:

начальник (джордж) .
 начальник (гарри) .
 начальник (нэнси) .

На уровне объектного языка смысл этих фраз следующий:
 «*X* является начальником, если

X — это Джордж или
X — это Гарри или
X — это Нэнси».

Однако из-за предположения о замкнутости мира фактический смысл этих трех фраз на уровне метаязыка будет несколько иным:

«*X* является начальником тогда и только тогда, когда

X — это Джордж или
X — это Гарри или
X — это Нэнси».

Отрицание в явной форме

Встроенный предикат `not` («не») имеет один аргумент. Этим аргументом является цель, значение истинности которой (после обработки данного запроса) заменяется противоположным. Если запрос успешен, то отрицание этой цели (запроса) является неудачей, и наоборот, если запрос терпит неудачу, то его отрицание будет успехом. Запрос

?- not (отец (питер, X)).

будет истинным тогда и только тогда, когда

?- отец (питер, X) .

потерпит неудачу.

Переменные в цели с предикатом `not` квантифицированы универсальным образом (т.е. используется квантор всеобщности).

Напишем правило, определяющее сельского жителя как человека, который не является ни горожанином, ни жителем пригорода.

горожанин (джек) .

житель_пригорода (сюзан) .

сельский_житель (X) :-

not (горожанин (X)) ,

not (житель_пригорода (X)) .

?- сельский_житель (билл) .

Yes .

Билл является сельским жителем, хотя в программе нет абсолютно никакой информации о его месте жительства.

Еще один пример «неестественных» ответов:

женщина (Анна) .

женщина (Юлия) .

мужчина (X) :-

not (женщина (X)) .

?- мужчина (X) .

No

?- мужчина (Вера) .

Yes

?- мужчина (Анна) .

No

В более новых версиях языка SWI-Prolog рекомендуется вместо имени предиката `not` использовать префиксное имя предиката `\+` (словами передается как «не доказуемо»; цель `\+ Goal` истинна, если `Goal` не доказуема). Отказываться от использования `not` предлагается по следующей причине: отношение `not`, определенное как недостижение цели, не полностью соответствует понятию отрицания в математической логике. Имя `not` оставлено только для совместимости, предполагается в дальнейшем его убрать.

4.3 Трудности с отсечением и отрицанием

Всякая точная наука основывается на приближительности.

Бертран Рассел

Преимущества отсечения

- При помощи отсечения часто можно повысить эффективность программы. Идея состоит в том, чтобы прямо сказать Пролог-системе: не пробуй остальные альтернативы, так как они все равно обречены на неудачу.
- Применяя отсечение, можно описать взаимоисключающие правила. Выразительность языка при этом повышается. Например, использование отсечений так, как ниже для предиката `p`:

```
p:-a1,!.  
p:-a2,!.  
p:-a3.
```

говорит, что второе правило будет применяться, если нельзя применить первое, а третье правило будет применяться только тогда, когда нельзя применить первые два.

Недостатки отсечения

Нарушается соответствие между процедурным и декларативным смыслом программы. Рассмотрим пример [2, стр. 132–133]:

```
p :- a, b.
p :- c.
```

С точки зрения логики формула p истинна тогда и только тогда, когда истинна формула $(a \ \& \ b) \vee c$. Это же утверждение остается в силе, если переставить правила:

```
p :- c.
p :- a, b.
```

Теперь определим предикат p с помощью отсечения:

```
p :- a, !, b.
p :- c.
```

С точки зрения логики формула p истинна тогда и только тогда, когда истинна формула $(a \ \& \ b) \vee (\neg a \ \& \ c)$. Если теперь порядок следования правил изменить на противоположный:

```
p :- c.
p :- a, !, b.
```

то изменится и декларативный смысл предиката (теперь p равносильно $(a \ \& \ b) \vee c$).

Изменение порядка следования предложений, содержащих отсечения, может повлиять на декларативную семантику программы.

Недостатки отрицания

Оказывается отрицание в Прологе определено через отсечение. Прежде чем дать это определение упомянем два встроенных предиката: `true` («истина») и `fail` («неудача»). Эти предикаты

не имеют аргументов, первый всегда истинен, второй всегда ложен.

```
?- true.
Yes
```

```
?- fail.
No
```

Следующее определение «отрицания» дает искомое процедурное значение этого предиката.

```
\+(P) :-
    P,!,fail.

\+(P) :-
    true.
```

Поэтому к сознательно принятому недостатку отрицания («замкнутый мир») добавляются и недостатки отсеечения. Пример:

```
r(a).
g(b).
p(X) :- \+ r(X).
```

На первый взгляд это безобидная программа, но она таит подводные камни. Два вызова, равносильные с логической точки зрения, приводят к разным результатам:

```
?- g(X),p(X).
X = b ;
No
```

```
?- p(X),g(X).
No
```

Порядок целей повлиял на работу программы. В первом случае, когда вызывается подцель $p(X)$, переменная X уже конкретизирована значением b . Во втором случае $p(X)$ вызывается со свободной переменной.

4.4 Рекурсия

Никакая переменная в Прологе не может изменить значение, поэтому привычные циклы в Прологе реализовать нельзя — вам надо использовать рекурсию. Этот принцип состоит в том, что задача сводится к нескольким случаям, принадлежащим к двум группам.

Тривиальные, или **граничные**, случаи.

Общие случаи, в которых решение составляется из решений отдельных (более простых) вариантов первоначальной задачи.

Этот метод применяется в Прологе постоянно. Основной методический подход к решению задач со списками следующий. Мы должны применять рекурсию по списку (или по одному из списков, когда несколько аргументов-списков у предиката).

Граничный случай — список пустой. Пишем соответствующее правило для пустого списка — приблизительно так

```
p([], L) :-
    .....
```

Общий случай (**рекурсивный переход**) — список не пустой. Тогда он имеет голову и хвост. Пишем соответствующее правило — приблизительно так

```
p([H|T], L) :-
    p(T, X),      %X - результат обработки хвоста списка
    % обрабатываем голову H,
    % как именно, это зависит от конкретной задачи,
    % получаем Y,
    % имея X и Y, получаем L,
    % как именно, это зависит от конкретной задачи.
```

Предположим, что имеется список чисел и требуется получить список квадратов этих чисел. Запрограммируем предикат `квадраты(+List, -NewList)`, где `List` — первоначальный список и `NewList` — список всех преобразованных элементов. Задачу преобразования списка `List` можно свести к двум случаям, описанным ниже.

1. Граничный случай: `List = []`.

Если список пустой, то и результат `NewList` — пустой список.

2. Общий случай: `List = [Head|Tail]`.

Чтобы преобразовать список, имеющий голову `Head` и хвост `Tail`, необходимо выполнить следующие действия:

- преобразовать хвост списка `Tail` с помощью рекурсивного вызова `квадраты`, получив список `NewTail`;
- возвести элемент `Head` в квадрат, получив `NewHead`;
- из новой головы и хвоста составить результат.

Текст программы:

```
квадраты([], []).
квадраты([Head|Tail], [NewHead|NewTail]) :-
    квадраты(Tail, NewTail),
    NewHead is Head*Head.

?- квадраты([1,2,3,4,5], Squares).
Squares = [1, 4, 9, 16, 25];
No
```

Определим предикат для вычисления суммы всех чисел, входящих в список (некоторые элементы могут быть и не числами).

```
summa_list([], 0).
summa_list([H|T], S) :- number(H),
    summa_list(T, S1),
    S is S1+H.
summa_list([H|T], S) :- \+(number(H)),
    summa_list(T, S).
```

Напишем предикат для вычисления списка всех положительных чисел, входящих в данный список чисел (теперь все элементы исходного списка являются числами).

```
p([], []).
p([H|T], [H|X]) :-
    p(T, X),
    H>0.
p([H|T], X) :-
    p(T, X),
    H<0.
```

Типичный прием в процедурном программировании (такие языки, как Паскаль и С) — это хранение каких-то значений в глобальных переменных. В Прологе информацию из одного вызова предиката в другой вызов предиката (одноименного или другого) передают с помощью параметров. В этих параметрах хранят информацию, изменяют ее и накапливают постепенно результат. Этот прием называется **методом накапливающего параметра**.

Напишем предикат для вычисления суммы всех чисел, входящих в список (некоторые элементы могут быть и не числами). Наш главный предикат `summa_list(+V, -N)` теперь обращается к вспомогательному предикату с дополнительным параметром (теперь получились два предиката с одним и тем же именем, но они различаются количеством аргументов, и в Прологе это допускается):

```
summa_list(V,N):-
    summa_list(V,0,N).    % второй параметр - накапливающий,
                          % исходное значение = 0
```

Теперь мы перенесли рекурсию в новый предикат:

```
summa_list([],N,N).
% закончилась рекурсия по списку, третий параметр стал таким же,
% как накопленный второй

summa_list([H|T],X,N):-
    number(H),
    Y is X + H,           % произошло изменение
    summa_list(T,Y,N).    % второго параметра

summa_list([H|T],X,N):-
    \+(number(H)),
    summa_list(T,X,N).    % второй параметр не изменился
```

Определим предикат для вычисления списка всех положительных чисел, входящих в данный список чисел (теперь все элементы исходного списка являются числами). Наш главный предикат `p(+V, -L)` теперь обращается к вспомогательному предикату с дополнительным параметром

```

p(V, L) :-
    p(V, [], L).
% второй параметр - накапливающий,
% исходное значение - пустой список

```

Теперь мы перенесли рекурсию в новый предикат:

```

p([], L, L).
% закончилась рекурсия по списку, третий параметр стал таким же,
% как накопленный второй

```

```

p([H|T], X, L) :-
    H > 0,
    p(T, [H|X], L). % произошло изменение второго параметра

```

```

p([H|T], X, L) :-
    H <= 0,
    p(T, X, L). % второй параметр не изменился

```

В двух последних примерах использование вспомогательного параметра просто демонстрация — можно обойтись и без него. В следующей задаче вспомогательный параметр более полезен.

Задача: определите предикат $p(+V, -L)$ — истинный тогда и только тогда, когда L — список всех элементов списка V , встречающихся в нем более одного раза.

Наш главный предикат $p(+V, -L)$ теперь обращается к вспомогательному предикату с дополнительным параметром

```

p(V, L) :-
    p(V, [], L).
% второй параметр - накапливающий,
% исходное значение пустой список

```

Теперь мы перенесли рекурсию в новый предикат:

```

p([], L, L).
% закончилась рекурсия по списку, третий параметр стал таким же,
% как накопленный второй

```

```

p([H|T], X, L) :-
    \+(member(H, T)),
    p(T, X, L).
% голова списка не входит еще раз в список, поэтому этот элемент не
% учитывается (не накапливается во втором параметре)

```

```

p([H|T],X,L):-
    member(H,T),
    \+(member(H,X)),
    p(T,[H|X],L).
% голова списка входит еще раз в список, и этот элемент не учитывался
% (не накапливался во втором параметре), поэтому мы его добавляем
% (накапливаем во втором параметре)

p([H|T],X,L):-
    member(H,T),
    member(H,X),
    p(T,X,L).
% голова списка входит еще раз в список, и этот элемент уже учитывался
% (накапливался во втором параметре), поэтому мы его не добавляем
% (не накапливаем во втором параметре)

```

Проверим:

```

?- p([a,b,a,c,c,d,d,e,d],X).
X = [d, c, a] ;
X = [d, c, a] ;
No

```

Чтобы избежать повторения ответов, надо вместо предиката `member` использовать предикат `memberchk` (см. раздел 4.1).

Накапливающие параметры могут использоваться не только в предикатах, задающих отношение на списках. Пример вычисления факториала с накапливающим параметром:

```

факториал(N,R):-
    f(N,0,1,R).

f(N,N,R,R).
f(N,X,Y,R):-
    X \= N,
    X1 is X+1,
    Y1 is Y*X1,
    f(N,X1,Y1,R).

```


5 ВНЕЛОГИЧЕСКИЕ ПРЕДИКАТЫ ПРОЛОГА

Пролог часто подвергается критике за использование средств, выходящих за рамки логики. Однако нам кажется, что программирование и не может оставаться чисто логическим. Для того чтобы некоторая сугубо логическая концепция могла найти применение в реальном программировании, ее надо основательно «подпортить».

*В.М. Антимиров, А.А. Воронков,
А.И. Дегтярев, М.В. Захарьяцев,
В.С. Проценко [3, стр. 363]*

В языке Пролог широко используются встроенные предикаты, которые нельзя описать в рамках логики первого порядка.

5.1 Анализ и синтез термов

5.1.1 Проверка типа термов

Термы могут принадлежать к разным типам: переменная, целое число, атом и т.д. Если терм представляет собой переменную, то он может быть в некоторый момент во время выполнения программы конкретизирован или не конкретизирован. Кроме того, если он конкретизирован, его значением может быть атом, структура и т.д. Иногда необходимо знать, к какому типу относится это значение. Например, если в программе используется оператор `is`, то правый операнд не должен содержать атомов или неконкретизированных переменных (или переменных конкретизированных не числами). К встроенным предикатам для проверки типа термов относятся `var`, `nonvar`, `atom`, `integer`, `float`, `number`, `atomic`, `compound`. Назначение этих предикатов описано ниже.

- `var(X)`. Выполняется успешно, если `X` во время проверки — не конкретизированная переменная.

- `nonvar(X)`. Выполняется успешно, если `X` — не переменная или `X` — уже конкретизированная переменная.
- `atom(X)`. Принимает истинное значение, если `X` во время проверки обозначает атом.
- `integer(X)`. Принимает истинное значение, если `X` во время проверки обозначает целое число.
- `float(X)`. Принимает истинное значение, если `X` во время проверки обозначает число с плавающей точкой.
- `number(X)`. Принимает истинное значение, если `X` во время проверки обозначает число.
- `atomic(X)`. Принимает истинное значение, если `X` во время проверки обозначает число или атом.
- `compound(X)`. Принимает истинное значение, если `X` во время проверки обозначает составной терм (структуру).

Следующие примеры вызовов иллюстрируют работы этих предикатов:

```
?- var(Z), Z=2.
```

```
Z = 2
```

```
Yes
```

```
?- Z=2, var(Z).
```

```
No
```

```
?- var(X+Y).
```

```
No
```

```
?- Z=2, nonvar(Z).
```

```
Z = 2
```

```
Yes
```

```
?- nonvar(Z), Z=2.
```

```
No
```

```
?- Z=2, integer(Z), integer(5).
```

```
Z = 2
```

```
Yes
```

```
?- atom(6.7) .
```

```
No
```

```
?- X=abc, atom(X), atom(==) .
```

```
X = abc
```

```
Yes
```

```
?- atomic(5.6), atomic(<>) .
```

```
Yes
```

```
?- compound(_+_), compound(f(a)) .
```

```
Yes
```

5.1.2 Создание и декомпозиция термов

Для декомпозиции термов и создания новых термов предусмотрены предикаты: `functor`, `arg`, `name` и «`=..`». Вначале рассмотрим предикат `=..`, который записывается как инфиксный оператор и читается как «юнив» (`univ`). Цель

```
Term =.. List
```

является истинной, если `List` — список, содержащий главный (самый внешний) функтор терма `Term`, за которым следуют его параметры. Примеры:

```
?- f(a,b,c) =.. X.
```

```
X = [f, a, b, c]
```

```
Yes
```

```
?- X =.. [=, 6, a] .
```

```
X = 6=a
```

```
Yes
```

```
?- X =.. [1, 2, 3] .
```

```
ERROR: Type error: 'atom' expected, found '1'
```

```
?- X =.. a .
```

```
ERROR: Type error: 'list' expected, found 'a'
```

```
?- [1, 2, 3] =.. X.
X = ['.', 1, [2, 3]]
Yes
```

Вернемся к программе дифференцирования, описанной в разделе 3.2. Во-первых, удалим из нее последнее правило

```
dv (A, X, dv (A, X) ) .
```

которое было временным.

Во-вторых, ликвидируем замеченные недостатки программы. Программа не умеет дифференцировать сложные функции. Правило дифференцирования сложной функции представляет собой более тонкий случай. В правиле утверждается, что производная от $f(g(x))$ по x есть производная $f(g(x))$ по $g(x)$, умноженная на производную $g(x)$ по x . В данной форме правило использует квантор по функциям и находится вне области логического программирования, рассматриваемого нами. Нам понадобится встроенный предикат `=.. / 2`.

Использование предиката `=.. / 2` позволяет изящно задать правило дифференцирования сложных функций (которое в программе должно находиться последним).

```
dv (F _G _X, X, DF*DG) :-
    F _G _X =.. [_, G _X],
    dv (F _G _X, G _X, DF) ,
    dv (G _X, X, DG) .
```

Проверим:

```
?- dv (ln (x*x) , x, R) .
R = 1 / (x*x) * (1*x+1*x)
Yes
```

Другой недостаток программы — невозможность продифференцировать y^3 по x . Необходимо заменить следующие два правила

```

dv(Y,_,0):-
    number(Y).
dv(Y,X,0):-
    atom(Y),
    X \= Y.

```

на правило, которое выдает 0, если дифференцируемое выражение не содержит переменной дифференцирования. Нам нужен предикат `in(+Term,+SubTerm)`, который выясняет, содержит ли терм `Term` данный подтерм `SubTerm`. Мы будем использовать предикат `=..`, который позволит нам получить компоненты сложного терма и рекурсивно проверять вхождения подтерма. Удобно использовать предикат `inList(+List,+SubTerm)`, который пытается обнаружить подтерм в терме из списка. Предикаты `in` и `inList` взаимно рекурсивны: `in` вызывает `inList`, а тот в свою очередь вызывает `in`.

```

in(Term,Term):-!.
in(Term,SubTerm):-
    compound(Term),
    Term =.. [_|Args],
    inList(Args,SubTerm).

inList([H|_],SubTerm):-
    in(H,SubTerm),!.
inList([_|T],SubTerm):-
    inList(T,SubTerm).

```

Проверим:

```

?- in([1,2,3],20).
No

?- in([1,2,f(3,20)],20).
Yes

?- in([1,2,f(3,20+a)],20+a).
Yes

?- in([1,2,f(3,20+a)],a).
Yes

```

Теперь необходимое правило дифференцирования выглядит так:

```
dv(Y, X, 0) :-
    \+ in(Y, X) .
```

и программа может теперь продифференцировать y^3 по x :

```
?- dv(y^3, x, R) .
R = 0
Yes
```

Замечание. Полученная программа все еще обладает недостатками. Правила дифференцирования X^A и A^X дополнительно требуют, чтобы A не содержало X . Кроме того, наша программа дает множество ответов, потому что, кроме стандартного правила, скажем, производная $\sin(x)$ по x , можно применить и правило для дифференцирования сложной функции. Следовательно, с помощью отсечений необходимо ликвидировать применение различных правил для одного выражения.

Рассмотрим теперь другие предикаты, предназначенные для декомпозиции термов.

- `functor(?Term, ?Functor, ?Arity)`

Предикат выдает истину, если терм `Term` имеет функтор `Functor` и его арность равна `Arity`.

- `arg(?N, +Term, ?Value)`

Предикат выдает истину, если терм `Term` является составным и его N -ый по-порядку аргумент (нумерация идет с 1) унифицируется с `Value`.

- `name(?AtomOrInt, ?List)`

Предикат выдает истину, если `List` есть список кодов ASCII символов, составляющих первый аргумент (атом или число).

Проиллюстрируем примерами использование этих предикатов:

```
?- functor(f(a,b,c), X, Y) .
X = f
Y = 3
Yes
```

```
?- functor(X,s,2) .
X = s(_G274, _G275)
Yes
```

```
?- arg(3,f(a,b,c),X) .
X = c
Yes
```

```
?- arg(X,f(a,b,c),Y) .
X = 1
Y = a ;
X = 2
Y = b ;
X = 3
Y = c ;
No
```

```
?- name(abcd,X),name(Y,X) .
X = [97, 98, 99, 100]
Y = abcd
Yes
```

```
?- name(123.45,X) .
X = [49, 50, 51, 46, 52, 53]
Yes
```

5.2 Ввод и вывод

Файлы в Прологе рассматриваются как последовательные потоки информации. Существуют **текущие входной** и **выходной потоки**. По умолчанию текущим входным потоком считается клавиатура, а выходным потоком — экран. Переключение между потоками осуществляется с помощью процедур:

- `see (+File)` — файл становится текущим входным потоком;
- `tell (+File)` — файл становится текущим выходным потоком;
- `seen` — закрывается текущий входной поток;
- `told` — закрывается текущий выходной поток.

Файлы читаются и записываются двумя способами: как последовательности символов и как последовательности термов.

Встроенные процедуры для чтения и записи символов и термов таковы:

- `read(-Term)` — вводит следующий терм;
- `write(+Term)` — выводит `Term`;
- `put(+Char)` — выводит символ, `Char` должно быть целочисленное выражение, значение которого есть код ASCII или атом в виде одной литеры;
- `get0(-КодСимвола)` — вводит следующий символ;
- `get(-КодСимвола)` — вводит ближайший следующий «печатаемый» символ.

Приведем два примера ввода и вывода в программах на Прологе, в которых также дополнительно показывается, как можно организовать повторение при вводе или выводе.

Пример 1. Пусть во время работы создана база данных, состоящая из множества фактов вида `отец(X, Y)`. Требуется записать все эти факты в файл `base.dat`.

```
'записать базу данных':-
    tell('base.dat'),
    base,
    told. % текущим выходным потоком стал экран
base:-
    отец(X, Y),
    write(отец(X, Y)),
    write(' '), nl, % предикат nl переводит курсор на новую строку
    fail.
base.
```

Предикат `fail` в первом правиле для `base` выдает неудачу и поэтому происходит возврат до первой возможной альтернативы — в данном случае это предикат `отец(X, Y)`. Повторный вызов его приводит к новой конкретизации переменных `X` и `Y`, после чего факт записывается в файл. Это повторяется до тех пор, пока не переписутся все факты «отец». Работа предиката `base`, тем не менее, успешно заканчивается, поскольку второе правило для `base` всегда выполняется.

Пример 2. Программа «Эхо» считывает терм, введенный с клавиатуры, и дублирует его на экран; если пользователь введет `stop`, то программа завершается.


```

эхо:-
    repeat,
    read(Term) ,
    write_ln(Term) ,
    'конец ввода?' (Term) , !.

'конец ввода?' (stop) :-
    write_ln(' - OK,bye') .
'конец ввода?' (_) :-
    fail.

```

Предикат `write_ln` осуществляет печать аргумента и затем переводит курсор на новую строку. Предикат `repeat` является встроенным; его определение очень просто:

```

repeat.
repeat:-
    repeat.

```

Такое определение предиката гарантирует, что сколько бы раз его не вызывали (в данном примере — в теле предиката `эхо`), он всегда выполняется. Неудача при вызове предиката `'конец ввода?'` вызывает возврат, а так как для предикатов `read` и `write` нет альтернативных правил, то повтор происходит в `repeat`. Отсечение в `эхо` после успешного выполнения предиката `'конец ввода?'` прекращает возврат в `repeat`.

5.3 Метапрограммирование

5.3.1 Эквивалентность программ и данных

Характерной особенностью Пролога является эквивалентность программ и данных — и то и другое представлено логическими термами. Для того чтобы можно было использовать эту эквивалентность, необходимо, рассматривать программы в качестве данных, а данные превращать в программы.

Смысл встроенного предиката `call(X)` в Прологе состоит в передаче терма `X` в качестве цели. Вместо вызова предиката

`call(X)` можно писать просто `X`. Соответствующие примеры будут рассмотрены ниже и в следующих разделах.

Доступность метапеременных означает, что в качестве целей в конъюнктивных вопросах и в теле предложений разрешается использовать переменные. В процессе вычисления, в момент обращения к такой переменной, ей должно быть сопоставлено значение — терм. Если переменной не сопоставлено значение в момент обращения, то возникает ошибочная ситуация.

Доступность метапеременных является важным инструментом метапрограммирования, используемым, в частности, при построении метапрограмм, которые обращаются с другими программами как с данными; они выполняют их анализ, преобразование и моделирование. Это же свойство существенно используется при определении отрицания (см. раздел 4.3) и при определении предикатов высших порядков.

5.3.2 Предположение об открытости мира

*Обладаю ли я знаниями? Нет.
Но когда низкий человек спросит меня о чем-либо, то, даже если я не буду ничего знать, смогу рассмотреть этот вопрос с двух сторон и обо всем рассказать ему.*

Конфуций

При работе с неизвестной информацией альтернативой предположению о замкнутости мира служит **предположение об открытости мира**:

если правило `P` отсутствует в текущей программе, то считается, что `P` ни истинно, ни ложно.

В соответствии с предположением об открытости мира запрос может обладать одним из трех допустимых истинностных значений: *истина*, *ложь* или *неизвестно*. Если запрос признан неизвестным, то программа может предпринять какие-то особые действия, скажем, она может обратиться к альтернативному ис-

точнику знаний. По умолчанию интерпретатор языка Пролог руководствуется предположением о замкнутости мира. Поэтому если требуется, чтобы поведение программы соответствовало предположению об открытости мира, то это нужно выражать в явном виде при составлении программы. Составление такой программы равнозначно изменению неявного предиката метаязыка, описывающего смысл запроса.

Пример программы, поведение которой соответствует предположению об открытости мира [6, с. 153].

Программа содержит позитивные факты

```
отец(том, боб) .
отец(боб, пэт) .
```

Явно заданы негативные факты в виде

```
false(отец(пэт, _)) .
```

Если нельзя доказать, что утверждение истинно или ложно, то оно считается неизвестным.

```
prove(P) :- P,
            write_ln('=> истинно'),!.
prove(P) :- false(P),
            write_ln('=> ложно'),!.
prove(P) :- \+(P),
            \+(false(P)),
            write_ln('=> неизвестно').
```

```
?- prove(отец(пэт, джим)) .
=> ложно
Yes
```

```
?- prove(отец(джим, X)) .
=> неизвестно
X = _G253
Yes
?- prove(отец(X, боб)) .
=> истинно
X = том
Yes
```

5.3.3 Программирование второго порядка

Программирование на Прологе расширяется введением методов, отсутствующих в модели логического программирования. Эти методы основаны на свойствах языка, выходящих за рамки логики первого порядка. Они названы методами второго порядка, поскольку речь здесь идет о множествах и их свойствах, а не об отдельных элементах. Кроме того, использование предикатных переменных позволяет рассматривать отношения как «первоклассные» объекты данных.

Предикат `findall(+Var, +Goal, -Bag)` создает список всех конкретизаций переменной `Var`, полученных при бэктрекинге (перебор с возвратом) при выполнении цели `Goal`, и унифицирует результат с `Bag`. `Bag` — пустой список, когда `Goal` не имеет решения.

```
?- findall(X, (member(X, [1,2,3,4,5]), 0 is X mod 2), L).
X = _G450
L = [2, 4]
Yes
```

Логика первого порядка позволяет проводить квантификацию по отдельным элементам. Логика второго порядка, кроме того, позволяет проводить квантификацию по предикатам. Включение такого расширения в логическое программирование влечет за собой использование правил с целями, предикатные имена которых являются переменными. Имена предикатов допускают обработку и модификацию.

Предикат `apply(+Term, +List)` присоединяет элементы списка `List` к аргументам термина `Term` и вызывает полученный терм в качестве цели. Например, `apply(plus(1), [2, X])` вызывает `plus(1, 2, X)`.

```
?- apply(plus, [1,2,X]).
X = 3
Yes
```

```
?- apply(is, [X, 4*6*(3+2)]).
X = 120
Yes
```

```
?- apply(=, [X, a*6*(x+2)]).
X = a * 6 * (x + 2)
Yes
```

```
?- apply(append, [[1,2,3], [6,7,8], X]).
X = [1,2,3,6,7,8]
Yes
```

Предикат `checklist(+Pred, +List)` проверяет все ли элементы списка `List` удовлетворяют предикату `Pred`.

```
?- checklist(number, [1,4,8.9,5/7]).
No
```

```
?- checklist(number, [1,4,8.9]).
Yes
```

```
?- [user].
|   p(2).
|   p(3).
|   p(5).
|   p(7).
|   user compiled, 52.34 sec, 388 bytes.
```

```
Yes
```

В данном запросе имя файла `user` используется для ввода небольших программ с клавиатуры. Конец файла указывается нажатием клавиш `CTRL + D`.

```
?- checklist(p, [2,5,2,7]).
Yes
```

```
?- checklist(p, [2,5,2,1]).
No
```

Пример. Предикат `map(+List, +F, -NewList)`, где `List` — первоначальный список, `F` — правило преобразования (бинарное отношение) и `NewList` — список всех преобразованных элементов. Задачу преобразования списка `List` можно свести к двум случаям: граничный случай, `List = []`, и общий случай, `List = [Head|Tail]`.

На языке Пролог получаем рекурсивную программу:

```
map([],_,[]).
map([Head|Tail], F, [NewHead|NewTail]):-
    map(Tail, F, NewTail),
    apply(F, [Head,NewHead]).
```

5.4 Операции с базой данных

5.4.1 Добавить в базу данных и удалить

База данных, в соответствии с реляционной моделью баз данных, представляет собой спецификацию набора отношений. Любая Пролог-программа может рассматриваться как подобная база данных; в ней спецификация отношений частично является явной (факты), а частично неявной (правила). Некоторые встроенные предикаты позволяют модифицировать эту базу данных во время выполнения программы. Такое обновление осуществляется путем добавления новых предложений в программу или удаления существующих предложений (причем эти операции осуществляются во время выполнения программы). Для этого предназначены предикаты `assert`, `asserta`, `assertz`, `retract` и `retractall`.

Предикат `assert(+Term)` добавляет факт или правило в программу (в базу данных в оперативной памяти). `Term` добавляется как последний факт или правило соответствующего предиката.

Например, цель

```
?- assert(родитель(том, боб)).
```

всегда истинна и в качестве побочного эффекта вызывает *подтверждение* (assertion) факта `родитель (том, боб)`. При добавлении правил следует вводить дополнительные скобки, чтобы учитывать старшинство термов. Например, синтаксически правильным является выражение

```
?- assert ( (мать (X, Y) :-родитель (X, Y) , женщина (X) ) ) .
```

Внесенные таким образом в базу данных предложения действуют точно так же, как часть первоначальной программы.

При выполнении предиката `retract (+Term)`, когда терм `Term` унифицируется с первым подходящим фактом или правилом в базе данных, факт или правило удаляется из базы данных.

Предикат `retractall (+Term)` удаляет все факты или правила в базе данных, которые унифицируются с `Term`.

Те предикаты, которые будут добавляться или изменяться с помощью предикатов `assert` и `retract`, необходимо объявить в программе как **динамические**. Для этого используется директива

```
:- dynamic <имя предиката>/<арность>.
```

Если динамическими являются несколько предикатов, то они в директиве перечисляются через запятую.

Предположим, что имеется следующая программа, которая отражает родственные отношения:

```
родитель ( пэм, боб) .
родитель ( том, боб) .
родитель ( том, лиз) .
% родитель ( боб, энн) .
родитель ( боб, пэт) .
родитель ( пэт, джим) .
```

```
женщина (пэм) .
женщина (пэт) .
женщина (лиз) .
женщина (энн) .
```

Следующий запрос дает единственный ответ

```
?- родитель (боб, X) .
X = энна ;
No
```

Добавим в базу данных новый факт и повторим запрос:

```
?- assert (родитель (боб, пэт) ) .
Yes
```

```
?- родитель (боб, X) .
X = энна ;
X = пэт ;
No
```

Удалим один факт:

```
?- retract (родитель (боб, _)) .
Yes
```

```
?- родитель (боб, X) .
X = пэт ;
No
```

Вернем факт родитель (боб, энна) в программу и проверим, как работает retractall:

```
?- assert (родитель (боб, энна) ) .
Yes
```

```
?- retractall (родитель (боб, _)) .
Yes
```

```
?- родитель (боб, X) .
No
```

Если даже удаляемых правил нет, то retractall все равно выдает Yes, в отличие от retract.

```
?- retractall (родитель (боб, _)) .
Yes
```



```
?- retract(родитель(боб, _)) .
No
```

Во время работы программы можно добавить любой предикат, который в программе отсутствовал; в этом случае его не надо предварительно объявлять динамическим — он становится таковым по умолчанию. Пример:

```
?- assert((мать(X, Y) :- родитель(X, Y), женщина(X)) .
X = _G408
Y = _G409
Yes

?- мать(X, Y) .
X = пэм
Y = боб ;
X = пэт
Y = джим ;
No
```

При внесении предложения в базу данных может потребоваться указать позицию, в которой это новое правило должно быть вставлено в базу данных. Предикаты `asserta` и `assertz` предоставляют возможность управлять позицией вставки: `asserta` добавляет в начало базы данных, а `assertz` (так же как и `assert`) — в конец.

Замечание. Обратите внимание, что если дважды вызвать `assert` с одним и тем же аргументом, то в базе данных появится два одинаковых предложения. Поэтому запросы, касающиеся этого предложения, будут давать несколько одинаковых ответов.

5.4.2 Пример: база данных «Достопримечательности»

Приведем пример программы, которая учится у пользователя [6, с. 162]. Предикат `place(Место, Авеню, Стрит)` связывает название места в городе с номерами улиц (авеню и стрит), на пересечении которых это место расположено. По заданному названию места данный предикат пытается определить его адрес, просматривая базу данных — множество фактов вида `adress(whitehous, 7, 1)`. Предикат `place` действует с

предположением об открытости мира в том смысле, что он не просто завершается неудачей, если не может найти название места в базе данных. Вместо этого предикат переключается на другую стратегию и получает сведения от пользователя, выступающего в роли альтернативного источника знаний. Предикат `place` учится на своем опыте, добавляя новые ответы в текущую программу.

Эта программа работает следующим образом.

```
?- goal.
Спрашивайте:
% adress.dat compiled 0.00 sec, 380 bytes
Введите название достопримечательности
|: whitehouse.
Это место вблизи 7  авеню
и 1 стрит.
Продолжать? yes
Введите название достопримечательности
|: grand_central.
-- это место - grand_central
вблизи какой авеню? (номер) 42.
вблизи какой стрит? (номер) 8.
Это место вблизи 42  авеню
и 8 стрит.
Продолжать? yes
Введите название достопримечательности
|: grand_central.
Это место вблизи 42  авеню
и 8 стрит.
Продолжать? no
Yes
```

При работе программы загружается файл `adress.dat`, который содержит единственный факт `address(whitehouse, 7, 1)`. Обратите внимание, что текстовый файл с адресами `adress.dat` должен существовать до работы нашей программы (в крайнем случае, пустой).

Заметьте, что при выполнении второго запроса система спросила у пользователя адрес «`grand_central`». Пользователь ввел 42 и 8, а затем интерпретатор вывел эти же самые значения как ответы на запрос. Потом тот же запрос идет повторно, и пре-

дикат place уже знает адрес «grand_central». После отрицательного ответа на вопрос «Продолжать?» в файл address.dat записываются все факты address из оперативной памяти Пролога (в данном случае добавился address(grand_central,42,8)).

```
:- dynamic address/3.

place(X,Avenu,Street) :-
    address(X,Avenu,Street),!.
place(X,Avenu,Street) :-
    write('-- это место - '),write(X),nl,
    write('вблизи какой авеню? (номер) '),
    read(Avenu),
    write('вблизи какой стрит? (номер) '),
    read(Street),
    assert(address(X,Avenu,Street)).

run(X) :-
    place(X,Avenu,Street),
    write('Это место вблизи '),write(Avenu),write(' авеню'),nl,
    write('и '),write(Street),write(' стрит.').

'ввести базу знаний' :- consult('address.dat').

goal:-
    write('Спрашивайте:'),nl,
    'ввести базу знаний',
    repeat,
    write('Введите название достопримечательности'),nl,
    read(X),
    run(X),nl,
    продолжать.

продолжать:- write('Продолжать? '), yes.

yes:-
    get_single_char(C),
    (name(n,[C]),write(no),'записать базу данных',!;
    write(yes),nl,fail).

'записать базу данных':-
    tell('address.dat'),
    base,
    told.

base:-
    address(X,Avenu,Street),
    write(address(X,Avenu,Street)),
    write('.'),nl, fail.

base.
```

Замечание 1. В правиле для предиката `yes` впервые в данном пособии используется синтаксическая конструкция с оператором «;». Правило вида

```
A :-  
    B1;  
    B2.
```

рассматривается системой как два правила

```
A :-  
    B1.  
A :-  
    B2.
```

Приоритет оператора «;» ниже, чем приоритет «,», поэтому если `B1` или `B2` есть конъюнкция предикатов, то `B1` или `B2` надо брать в скобки (что и сделано в правиле для `yes`).

Замечание 2. Встроенный предикат `get_single_char(C)` из входного потока, вводимого из клавиатуры, читает единственный символ и возвращает его код ASCII. Введенный символ не отображается на экране.

5.4.3 Пример: запоминающая функция

Этой ужасной минуты я не забуду никогда в жизни! — сказал Король.

Забудешь — заметила Королева, — если не запишешь в записную книжку.

*Льюис Кэрролл.
Алиса в Зазеркалье*

Запоминающие функции сохраняют промежуточные результаты с целью их использования в дальнейших вычислениях. Запоминание промежуточных результатов в чистом Прологе невозможно, поэтому реализация запоминающих функций основана на побочных эффектах.

Исходной запоминающей функцией является предикат `lemma (Goal)`. Операционное понимание состоит в следующем: предпринимается попытка доказать цель `Goal`, и если попытка удалась, результат доказательства сохраняется в виде леммы. Отношение задается следующим образом:

```
lemma(P) :- P, asserta((P :- !)).
```

Следующая попытка доказать цель `P` приведет к применению нового правила, что позволяет избежать ненужного повторения вычислений. Отсечение введено для того, чтобы воспрепятствовать повторному рассмотрению всей программы. Применение отсечения обосновано лишь в тех случаях, когда цель `P` имеет не более одного решения.

Использование лемм демонстрируем на примере программы, вычисляющей функцию Аккермана:

$$\begin{aligned} f(0, n) &= n+1; \\ f(m, 0) &= f(m-1, 1), \text{ если } m > 0; \\ f(m, n) &= f(m-1, f(m, n-1)), \text{ если } m, n > 0. \end{aligned}$$

Факт `do` будем использовать в качестве глобального счетчика для подсчета вызовов функции `ackerman`, предикат без аргументов `inc` увеличивает значение счетчика на 1.

```
:- dynamic do/1.
```

```
ackerman(0, N, N1) :-
    inc,
    N1 is N+1.
```

```
ackerman(M, 0, Val) :-
    M>0,
    inc,
    M1 is M-1,
    ackerman(M1, 1, Val).
```

```
ackerman(M, N, Val) :-
    M>0, N>0,
```

```

    inc,
    N1 is N-1,
    ackerman(M,N1,Val1),
    M1 is M-1,
    ackerman(M1,Val1,Val) .

```

```
inc:-
```

```

    do(X),
    X1 is X+1,
    retract(do(_)),
    assert(do(X1)) .

```

```
do(0) .
```

```
?- ackerman(3,2,X),do(Y) .
```

```
X = 29
```

```
Y = 541
```

```
Yes
```

В данном случае предикат `ackerman` вызвал себя рекурсивно 541 раз. Теперь приведем вариант с запоминающей функцией.

```
:- dynamic do/1,ackerman/3.
```

```
ackerman(0,N,N1):-
```

```

    inc,
    N1 is N+1.

```

```
ackerman(M,0,Val):-
```

```

    M>0,
    inc,
    M1 is M-1,
    lemma(ackerman(M1,1,Val)) .

```

```
ackerman(M,N,Val):-
```

```

    M>0,N>0,
    inc,
    N1 is N-1,
    lemma(ackerman(M,N1,Val1)),
    M1 is M-1,
    lemma(ackerman(M1,Val1,Val)) .

```

```

lemma(P):-
    P,
    asserta((P:-!)).

do(0).

inc:-
    do(X),
    X1 is X+1,
    retract(do(_)),
    assert(do(X1)).

?- ackerman(3,2,X),do(Y).
X = 29
Y = 73
Yes

```

Теперь предикат `ackerman` вызывает себя только 73 раза.

Приведем замечание из [2, стр. 164]. Приведенные примеры показывали некоторые, безусловно удобные способы применения предикатов `assert` и `retract`. Но при использовании этих предикатов необходимо соблюдать особую осторожность. Излишнее и непродуманное применение этих средств нельзя рекомендовать в качестве хорошего стиля программирования. Использование операций подтверждения и извлечения фактов и правил по сути приводит к модификации программы. Поэтому отношения, которые в какой-то момент были действительными, в другое время могут оказаться недействительными. В разное время на одни и те же вопросы система будет давать разные ответы. Поэтому применение значительного количества операций подтверждения и извлечения может затемнить смысл программы. В конечном итоге может оказаться, что поведение программы трудно понять, нелегко объяснить, и поэтому ей нельзя доверять.

ЛИТЕРАТУРА

1. Robinson J.A. A machine-oriented logic based on resolution principle. *Journal of the ACM*, 12: 23–41, 1965.
2. Братко И. Алгоритмы искусственного интеллекта на языке PROLOG: Пер. с англ. — 3-е изд. — М.: Издательский дом «Вильямс», 2004. — 640 с.
3. В.А. Антимиров, А.А. Воронков, А.И. Дегтярёв, М.В. Захарьящев, В.С. Проценко. Математическая логика в программировании. Обзор // Математическая логика в программировании: Сб. статей 1980–1988 гг.: Пер. с англ. — М.: Мир, 1991. — 408 с.
4. Грэй П. Логика, алгебра и базы данных: Пер. с англ. — М.: Машиностроение, 1989. — 359 с.
5. Дейкстра Э. Заметки по структурному программированию // У. Дал, Э. Дейкстра, К. Хоор. Структурное программирование. — М.: Мир, 1975. — С. 7–97.
6. Малпас Дж. Реляционный язык Пролог и его применение. — М.: Наука, 1990. — 464 с.
7. Одинцов И.О. Профессиональное программирование. Системный подход. — СПб.: БХВ-Петербург, 2002. — 512 с.
8. Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог. — М.: Мир, 1990. — 235 с.
9. Язык Пролог в пятом поколении ЭВМ: Сб. статей, 1983–1986 гг. / Сост. Н.И. Ильинский. — М.: Мир, 1988.

ПРИЛОЖЕНИЕ

Помощь в SWI-Prolog'e

После инсталляции интерпретатора SWI-Prolog есть возможность обратиться к помощи через ярлык «manual.html». Кроме того, документация для пользователя находится в файле manual.pdf (читается с помощью программы Adobe Acrobat).

Кроме того, SWI-Prolog имеет помощь online. Вызов «?- help(<имя предиката>) .» выдает на экран информацию об этом предикате. Вызов «?- help.» выдает информацию о том, как пользоваться этой помощью.

Трассировка выполнения программы

Предикат trace позволяет пользователю отслеживать состояние интерпретатора Пролога.

- trace/0

Стартует трассировщик. В процессе мониторинга в выходной файл выводятся все целевые утверждения, обрабатываемые интерпретатором языка. Зачастую этой информации для пользователя слишком много.

- trace(+Pred)

Эквивалентно вызову trace(+Pred, +all) .

- trace(+Pred, +Ports)

Устанавливаются точки трассировки на всех предикатах, удовлетворяющих спецификации Pred. Ports есть список портов (call, redo, exit, fail). Атом all ссылается на все порты. Если перед именем порта стоит символ «-», то данный порт не трассируется. Знак «+» говорит о том, что установлена трассировка данного порта.

При трассировке предиката получаем следующую информацию:

- 1) уровень глубины рекурсивных вызовов;
- 2) когда предпринимается первая попытка обработки целевого утверждения (порт call);
- 3) когда цель успешно достигнута (порт exit);
- 4) возможность других соответствий целевому утверждению (порт redo);

5) невозможность достижения цели, поскольку все попытки завершились неудачно (порт `fail`).

Примеры:

```
?- trace(hello) . % трассировка всех портов предиката hello с любой аргументом
?- trace(foo/2, +fail) . % трассировка неудач при вызове foo/2
?- trace(bar/1, -all) . % прекращение трассировки bar/1.
```

Проведем трассировку для программы:

```
родитель ( пэм, боб) .
родитель ( том, боб) .
родитель ( том, лиз) .
родитель ( боб, энн) .
родитель ( боб, пэт) .
родитель ( пэт, джим) .

предок (X,Y) :-
    родитель (X,Y) .

предок (X,Y) :-
    родитель (X,Z) ,
    предок (Z,Y) .

?- trace(родитель) .
%      родитель/2: [call, redo, exit, fail]
Yes

[debug] ?- trace(предок) .
%      предок/2: [call, redo, exit, fail]
Yes

[debug] ?- предок(том,пэт) .
T Call: (6) предок(том, пэт)
T Call: (7) родитель(том, пэт)
T Fail: (7) родитель(том, пэт)
T Redo: (6) предок(том, пэт)
T Call: (7) родитель(том, _G414)
T Exit: (7) родитель(том, боб)
T Call: (7) предок(боб, пэт)
T Call: (8) родитель(боб, пэт)
T Exit: (8) родитель(боб, пэт)
T Exit: (7) предок(боб, пэт)
T Exit: (6) предок(том, пэт)
Yes
```