ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
Государственное образовательное учреждение высшего профессионального образования
«ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

В.Б. Новосельцев, Г.Д. Копаница

ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ И МОДЕЛИРОВАНИЕ СИСТЕМ

Учебное пособие

Издательство
Томского политехнического университета
2008

УДК 004.89 (075.8)
ББК 32.813я73
Н 74

Новосельцев В.Б.

Н 74 Логическое программирование и моделирование систем: учебное пособие / В.Б. Новосельцев, Г.Д. Копаница. – Томск: Изд-во Томского политехнического университета, 2008. – 112 с.

В пособии изложены современные подходы к программированию систем управления информационными комплексами. Представлен теоретический аппарат, необходимый для успешной работы в различных подобластях Искусственного Интеллекта. Рассмотрены современные трактовки парадигмы наиболее популярного в настоящее время логического подхода. Описаны методы работы с неопределенностью и модальностью при реализации программных проектов, ориентированных на работу с распределенной когнитивной информацией.

Предназначено для студентов направления 552800 – «Информатика и вычислительная техника».

УДК 004.89 (075.8)
ББК 32.813я73

Рекомендовано к печати Редакционно-издательским советом
Томского политехнического университета

Рецензенты

Доктор технических наук, профессор ТГУ
А.Ю. Матросова

Доктор технических наук, профессор ТУСУРа
В.Т. Калайда

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 4
ГЛАВА 1. ВВЕДЕНИЕ В МАТЕМАТИЧЕСКУЮ ЛОГИКУ 8
1.1. Классические логические системы 8
1.1.1. Логика высказываний 8
1.1.2. Логика предикатов первого порядка 11
1.2. Модальная логика 15
ГЛАВА 2. СТРУКТУРНЫЕ ИМПЛИЦИТНЫЕ МОДЕЛИ 37
2.1. Формальная теория 37
2.2. Нерекурсивные детерминированные С-модели 42
ГЛАВА 3. ВВЕДЕНИЕ В VISUAL PROLOG 44
3.1. Установка и начало работы в Visual Prolog 45
3.2. Программирование в Логике 48
3.3. Программы Visual Prolog 64
3.4. Унификация и поиск с возвратом 80
СПИСОК ЛИТЕРАТУРЫ 111

ВВЕДЕНИЕ

В книге обсуждается набор технологий и методов, которые принято относить к области Искусственного Интеллекта – ***ИИ*** (*Arttificial Intelligense* – ***AI***), ориентированных на работу с семантической составляющей информации (смыслом). Мы попытались максимально приблизить содержание пособия к практическим потребностям, возникающим при моделировании информационных структур, разработке систем управления знаниями и другими распределенными структурами, не пренебрегая при этом строгостью изложения необходимых теоретических положений.

Представленный материал появился в результате многолетнего чтения курсов по дисциплинам, связанным с ***ИИ***, в классическом и политехническом университетах Томска. Авторы надеются, что представленное учебное пособие сможет послужить основой преподавания дисциплин по приложениям ***ИИ*** или иным направлениям систем управления знаниями – ***СУЗ*** (*Knowledge Based Systems* – ***KBS***).

Сразу стоит отметить, что сам термин *искусственный интеллект* является весьма расплывчатым и недостаточно адекватным, а границы соответствующей предметной области – нечеткими и размытыми. В различных источниках к тематике ***ИИ*** относят *игру* в шахматы (и не только в них), машинное *зрение*, *поведение* человекоподобных роботов и многое другое. Первоначально целью создания *Искусственного Интеллекта* считалось компьютерное моделирование *всех* функций высшей нервной деятельности человека, но, очень быстро было показано, что это – задача нереализуемая, по крайней мере, на нынешнем уровне знаний. Невозможно, в частности, понять сущности *интуиции* (никто до сих пор так и не смог формализовать акта открытия Ньютоном закона всемирного тяготения), механизма *веры*, природы других *иррациональных отношений* (любви, антипатии) и так далее.

В сообществе специалистов по информационным технологиям бытует шутка: «искусственным интеллектом занимаются те, кому недостает естественного». Поскольку в каждой шутке есть доля «шутки», имеет смысл дистанцироваться от несколько одиозного термина и ограничить оптимистические настроения, касающиеся возможности создания «полноценных Големов». Более того, не предполагая наличия волшебства, мы можем, опираясь на фундаментальную теорему Гёделя, утверждать, что любая достаточно богатая модель интеллекта, является неполной,

т. е. содержит парадоксы[1] – утверждения, которые в рамках модели не являются ни истинными, ни ложными. К сожалению, большая часть разработчиков т. н. «интеллектуальных комплексов» не имеет представления о принципиальных ограничениях, либо не придает им должного значения.

Возвращаясь к обсуждаемой проблеме, мы должны учитывать следующие ***важные соображения***:

1. «Интеллект» в общем смысле – понятие софистическое, т. е. неформализуемое – поэтому мы, как правило, будем использовать термин системы управления знаниями – СУЗ, что примерно соответствует английскому *knowledge based systems – KBS*. При этом под знаниями подразумевается совокупность четко определенных и интерпретируемых информационных структур и вполне дефинированных правил этих структур обработки. Важно отметить (обычно это остается «за кадром»), что упомянутые вполне определенные правила являются неотъемлемой (и, возможно, наиболее существенной) составляющей того, что здесь мы называем знаниями.
2. Даже, введя ту или иную формализацию, мы будем стремиться отдавать себе отчет о вычислительной (алгоритмической) сложности работы с введенными структурами. – Так, использование семантических сетей или сетей фреймов, в общем случае, подразумевает *экспоненциальные* оценки соответствующих алгоритмов [1], в то время как обсуждаемые здесь формализмы обладают полиномиальными оценками.
3. Имея соображения о теоретической сложности решения поставленной проблемы, мы не должны упускать из внимания реализационных аспектов: подбора структур данных, учета особенностей используемой платформ и т. д.
4. Наконец, мы не должны упускать из внимания так называемой проблемы *конечного пользователя* (*end-user*). Можно создать мощнейшую систему, но, если общение с ней предполагается на языке модальной логики высших порядков или в терминах теории ординалов, то вряд ли она станет популярной.

Итак, задачей ***ИИ*** – пока мы будем использовать этот термин – является построение программной системы, которая обладала бы уровнем эффективности решения некоторых неформализуемых (плохоформализуемых) задач, сравнимым с человеческим (или превосходящим его). Общей теории ***ИИ*** не существует, поскольку, как было отме-

[1] *Проще говоря, одним из следствий этой теоремы является невозможность создания полной формальной модели* ***интеллекта вообще*** *средствами формальной составляющей* ***интеллекта*** *человеческого.*

чено выше, эклектика и софистика являются явными (и, возможно, необходимыми на данном этапе) явлениями. К настоящему моменту не определено, что именно считать необходимыми и достаточными условиями достижения интеллектуальности. На этот счёт существует ряд критериев [2] и гипотез, как правило, общефилософского характера (не представляющих интереса (см. критику Сёрля [3]) в рамках данного текста, поскольку здесь хотелось бы рассуждать о практически реализуемых и полезных проектах, а не о пресловутой «голой курице» Платона-Диогена). – Попытаемся ограничиться программно реализуемыми подходами к моделированию так называемых интеллектуальных процессов.

В рамках исследований по ***ИИ*** фиксируются два базовых подхода: *имитация* и *моделирование*. Первый из них – имитация – является весьма привлекательным и вызывает порой неоправданный энтузиазм. На базе этого подхода строятся многие игровые программы, программы распознавания образов, оптимизирующие стратегии и т. д. Именно в рамках этого подхода создавались первые, да и нынешние, разговаривающие куклы и механические слуги. Наиболее интересным примером имитации, пожалуй, является программа *Eliza* (в варианте врача-психоаналитика), разработанная *Джозефом Вейзенбаумом* в 1966 году, которая способна поддерживать диалог с пациентом. Определенный практический эффект достигнут в области анализа изображений. Современные методы позволяют даже работать с многообъектными сценами, такими как групповые снимки, хотя и остаются, по-видимому, весьма далекими от тех, которые свойственны человеку. Общей характеристикой имитирующих систем является использование конечного числа статичных шаблонов. Вообще говоря, сказанного вполне достаточно для того, чтобы согласиться с бесперспективностью данного направления в плане сколь-нибудь близкого приближения к генеральной цели – реализации искусственного разума.

Направление *моделирования* сравнительно молодо. Несмотря на перспективность и известную успешность этого подхода, здесь имеются свои «подводные камни – мы можем моделировать *поведение* или *структуру* системы, если достаточно четко воспринимаем законы ее (системы) *функционирования*. К сожалению, до сих пор понимание того, что каждый из нас носит в черепной коробке, по большому счету, вряд ли можно считать достаточно полным. Если говорить о структуре мозга, то известно, что образующими элементами выступают клетки особого вида – нейроны. Между нейронами существуют физические связи, организованные аксонами, синапсами и чем-то там еще. Доказано, что, по крайней мере, одна из форм взаимодействия нейронов реализуется электрохимическим путем, но вопрос, является ли эта форма един-

ственной, остается открытым. Современная ситуация напоминает гипотетическую попытку кроманьонца понять, как устроен персональный компьютер, и что он может делать (что-то жужжит и пощелкивает, можно что-то нажимать и двигать, на экране меняются картинки и нагреваются отдельные части). Впрочем, что там говорить о мозге, если мы до сих пор не в состоянии полностью осмыслить механики процесса хождения на двух конечностях. С другой стороны, не все уж так беспросветно в направлении *моделирования* когнитивных процессов.

Исходя из сказанного, можно смело утверждать, что наиболее перспективным является подход, когда разработчик четко фиксирует *область применения* своего «интеллектуального» продукта, формализует механизм, *реализующий* этот программный продукт, и оценивает *эффективность* работы построенного механизма.

На данный момент не существует систем искусственного интеллекта, однозначно отвечающих основным задачам, обозначенным выше. Успехи в исследовании функциональных и немонотонных рассуждений позволяют совершить серьёзный шаг вперёд в построении систем обработки знаний.

Остановимся подробнее на используемых в области интеллектуальных систем подходах, по возможности иллюстрируя их практически полезными и интересными примерами.

Наиболее часто используемым при построении ***СУЗ*** является логический подход, причем это необязательно подразумевает работу с экзотическим программным инструментарием. В то же время хотелось бы отметить, что современные системы логического и функционального программирования, несомненно, являются наиболее естественными для реализации ***СУЗ*** самого широкого класса, именно поэтому в учебнике довольно большое внимание уделяется приемам использования конкретной версии языка Prolog.

ГЛАВА 1. ВВЕДЕНИЕ В МАТЕМАТИЧЕСКУЮ ЛОГИКУ

В практической деятельности, связанной с реализацией программных комплексов с элементами интеллекта, выяснилась недостаточность классической логики (логики высказываний и логики предикатов) для моделирования и решения целого ряда проблем. Изучение этих ограничений привело к разработке иных логических систем, которые принято называть *неклассическими* [4]. Существует великое многообразие неклассических логик, но здесь будут рассмотрены те из них, которые уже сейчас находят применение в области ***ИИ***.

Существуют две группы неклассических логик: одни принципиально конкурируют с классическими, а другие, в некотором смысле, являются их расширениями[2]. Классическими представителями первой группы являются многозначная и интуиционистская логики. Вторая группа характеризуется, прежде всего, немонотонностью процесса вывода – в нее входят разнообразные модальные исчисления. Название *модальная логика* отражает тот факт, что в соответствующих формальных теориях используются операторы модальности (возможности/необходимости), действующие в высказываниях и формулах.

Напомним, прежде всего, базовые понятия и свойства классических исчислений.

1.1. Классические логические системы

1.1.1. Логика высказываний

Логика высказываний изучает предложения (декларативные фразы), которые могут быть истинными, либо ложными, но ни то и другое вместе (см. [5]). *Высказывания* составляют элементарные фразы логического языка, называемые также *атомарными формулами* или *атомами*. Фразы логического языка строятся рекурсивно на основе высказываний и множества синтактических правил. Эти фразы называют *формулами*.

Синтаксис

- **Базис**: всякое высказывание является формулой.
- **Индукционный шаг**: если X и Y являются формулами, то

$$\neg X, (X \wedge Y), (X \vee Y), (X \supset Y) \text{ и } (X \equiv Y) \text{ суть формулы.}$$

[2] *Являются расширением в том смысле, что их языки и теоремы обобщают языки и теоремы классических исчислений.*

- **Ограничение**: формула однозначно получается с помощью правил, описанных в базисе и индукционном шаге.

Данный синтаксис позволяет распознавать формулы среди произвольных соединений символов, а семантика определяется множеством законов, которые дают возможность приписывать формулам определенные значения. Поскольку высказывание может быть либо истинным, либо ложным, это естественно подводит к введению *семантической области* $\mathbb{S}=_{def}$ {**И**, **Л**} (или {1, 0}). *Интерпретация* формулы состоит в приписывании ей одного и только одного из двух *значений истинности*: **И** (истинно) или **Л** (ложно). Семантика должна быть *композиционной*: значение формулы есть функция значений ее компонент, точнее, приписываемое формуле значение истинности будет зависеть только от структуры этой формулы и от значений истинности, приписанных составляющим ее высказываниям. Семантика отрицания ($\neg$) и бинарных связок ($\wedge$, $\vee$, $\supset$, $\equiv$) имеет обычный логический смысл и приводится в виде предложений в [5,6].

Семантика

Интерпретация есть функция, которая сопоставляет каждому высказыванию *p* определенное значение истинности. Эта функция, область определения которой является множеством высказываний, продолжается на множество формул с помощью правил семантики. Соответствующее продолжение тоже называется *интерпретацией*. Интерпретация, при которой формула истинна, называется моделью для этой формулы. Формула называется *общезначимой*, если она всегда истинна, независимо от значений истинности, приписанных составляющим ее высказываниям. Формула называется *выполнимой*, если существует хотя бы одна интерпретация, при которой она истинна. Формула, которая не является выполнимой, называется *невыполнимой* или *противоречивой*. Формула, не являющаяся ни общезначимой, ни противоречивой, называется *нейтральной*. Итак, множество формул разбивается на три подмножества, содержащие соответственно общезначимые, нейтральные и невыполнимые формулы. Общезначимые формулы иначе называют *тавтологиями*. Если A – формула, то запись $\models A$ выражает тот факт, что A является тавтологией. Вообще, если $\mathbb{E}$ – множество формул, то запись $\mathbb{E}\models A$ означает, что все интерпретации, обращающие в истинные все формулы из $\mathbb{E}$, обращают в истинную и формулу A. При этом A называется *логическим следствием из* $\mathbb{E}$.

Существует и более традиционный способ определения понятий *общезначимой формулы* и *логического следствия* для формальных языков,

который осуществляется посредством некоторой *аксиоматической системы* или *системы натурального вывода*. Аксиоматическая система состоит из множества *аксиом*, т. е. из механизмов, позволяющих выводить новые общезначимые предложения из аксиом и уже полученных общезначимых предложений. В таком контексте общезначимые предложения, построенные указанным способом, называются *теоремами*, доказательство которых – это упорядоченный список аксиом, правил вывода и уже известных теорем, позволяющих получить (вывести) данную теорему.

Если задана некоторая аксиоматическая система, то факт доказуемости в ней формулы A записывается как $\vdash A$. Более общо, если $\mathbb{E}$ является множеством формул, то выражение $\mathbb{E} \vdash A$ означает, что A доказуема с использованием формул из $\mathbb{E}$; при этом элементы множества $\mathbb{E}$ рассматриваются как дополнительные аксиомы и являются *гипотезами*.

От аксиоматической системы требуется, чтобы она была *адекватной* в том смысле, что все ее теоремы должны быть общезначимыми формулами. Требуется также и *полнота*, т. е. взаимность адекватности: всякая общезначимая формула должна быть теоремой. В этих предположениях символы $\vDash$ и $\vdash$ являются совершенно эквивалентными.

Соответствие между определениями, базирующимися на понятии семантики, и определениями, базирующимися на понятии аксиоматической системы, проиллюстрировано таблицей на рис. 1.1.

Семантические определения	**Аксиоматические определения**
Формула общезначима, если она истинна для всех ее интерпретаций	Формула является теоремой, если она представляет собой результат доказательства на основе некоторых схем аксиом
Формула невыполнима, если она ложна для всех ее интерпретаций	Формула невыполнима, если ее отрицание является теоремой
Формула A является логическим следствием некоторого множества формул $\mathbb{E}$, если все интерпретации, при которых истинны все формулы из $\mathbb{E}$, обращают в истинную и формулу A	Формула A доказуема на основе некоторого множества формул $\mathbb{E}$, если она является результатом некоторого доказательства на основе каких-то схем аксиом и формул из множества $\mathbb{E}$
Две формулы логически эквивалентны, если они истинны при одних и тех же интерпретациях	Две формулы логически эквивалентны, если каждая из них доказуема на основе другой формулы

Рис. 1.1. Семантические и аксиоматические определения

В качестве примера (см. [5,6]) можно привести адекватную и полную аксиоматическую систему, содержащую три схемы аксиом (A1–A3) и одно правило вывода (***MP***).

(A1) $(X \supset (Y \supset X))$,

(A2) $((X \supset (Y \supset Z)) \supset ((X \supset Y) \supset (X \supset Z)))$

(A3) $((\neg X \supset \neg Y) \supset ((\neg X \supset Y) \supset X)$.

Каждый раз, когда в одной из этих схем аксиом X, Y, Z заменяются произвольными формулами, получается тавтология.

(***MP***) Если X и $(X \supset Y)$ – теоремы, то Y – теорема.

Правило вывода (***MP***) называют ***modus ponens***.

1.1.2. Логика предикатов первого порядка

Укажем основные понятия логики предикатов (более подробно см. [3]). Вначале определяются атомы логики предикатов (аналогично логике высказываний). Атомы представляются символами следующих четырех классов:

- *переменные* (обозначаемые *x, y, z* и т. д.);
- *индивидные константы* (обозначаемые *a, b, c* и т. д.);
- *функциональные константы* или *имена функций* (*f, g, h, …*);
- *предикатные константы* или *имена предикатов* (*P, Q, R, …*).

Индивидные, функциональные и предикатные константы являются *внелогическими константами* языка.

Эти основные символы служат для формализации четырех понятий:

- *Терм* есть переменная или функциональная форма.
- *Функциональная форма* – это функциональная константа, соединенная с подходящим числом термов. Если f является функциональной n-местной константой и $t_1,\ldots,t_n$ – термы, то соответствующая форма обычно обозначается через $f(t_1,\ldots,t_n)$. Если $n = 0$, то функциональная форма $f(\,)$ – индивидная константа.
- *Предикатной формой* называется предикатная константа, соединенная с подходящим числом термов. Если P является предикатной m–местной константой и $t_1,\ldots,t_m$ – термы, то соответствующая форма обычно обозначается через $P(t_1,\ldots,t_m)$. При $m = 0$ предикатная форма $P(\,)$ обозначается через P и называется *высказыванием.*
- *Атомом* является предикатная форма или некоторое равенство, т. е. выражение вида $(s=t)$, где s и t – термы.

Слова «*функция*» и «*предикат*» часто используются соответственно вместо выражений «*функциональная форма*» и «*предикатная форма*».

Формулы логического языка строятся по индукции из атомов в соответствие с некоторым множеством синтаксических правил. В этих правилах используются связки $\neg$, $\square$, $\square$, $\supset$ и $\equiv$ из логики высказываний и

еще два символа $\forall$ и $\exists$, называющиеся соответственно *квантором общности* и *квантором существования*. Названные связки и константы являются логическими константами языка.

Синтаксис

- **Базис**: любой атом является формулой.
- **Индукционный шаг**: если X и Y – фомулы и x – переменная, то $\neg X$, $(X \wedge Y)$, $(X \vee Y)$, $(X \supset Y)$, $(X \equiv Y)$, $\forall x X$ и $\exists x X$ – формулы.
- **Ограничение**: формула получается только с помощью правил, описанных в базисе и индукционном шаге.

Формулы исчисления предикатов могут быть проинтерпретированны, т. е. могут быть получены некоторые *значения истинности*. Составные части формулы исчисления предикатов – не только подформулы, но и термы, поэтому необходимо интерпретировать еще и термы. Интуитивно терм означает *объект*. Итак, интерпретация должна указывать множество объектов, называемое *областью интерпретации*. Точнее, *интерпретация* **I** есть тройка $(\mathbb{S}, \mathbf{I}_c, \mathbf{I}_v)$, где

- $\mathbb{S}$ – непустое множество (*область интерпретации*);
- $\mathbf{I}_c$ – функция, сопоставляющая каждой n-местной константе f функцию $\mathbf{I}_c(f)$ из $\mathbb{S}^n$ в $\mathbb{S}$; она также каждой m-местной предикатной константе ставит в соответствие функцию $\mathbf{I}_c(P)$ из $\mathbb{S}^m$ в {**И, Л**};
- $\mathbf{I}_v$ – функция, сопоставляющая каждой переменной некий элемент из $\mathbb{S}$.

Затем вводится обозначение, служащее для интерпретации двух видов квантифицированных формул. Если **I** – некоторая интерпретация с областью $\mathbb{S}_\mathbf{I}$, x – переменная и d – элемент из $\mathbb{S}_\mathbf{I}$, то $\mathbf{I}_{x/d}$ означает такую интерпретацию **J**, что $\mathbb{S}_\mathbf{J}=\mathbb{S}_\mathbf{I}$, $\mathbf{J}_c=\mathbf{I}_c$, $\mathbf{J}_v(x)=d$ и $\mathbf{J}_v(y)=\mathbf{I}_v(y)$ для всех переменных y, отличных от x. Теперь для каждой интерпретации $\mathbf{I} = (\mathbb{S}, \mathbf{I}_c, \mathbf{I}_v)$ можно задать такие правила интерпретации, которые каждой формуле A сопоставят значение истинности $\mathbf{I}(A)$ и каждому терму t сопоставят элемент $\mathbf{I}(t)$ из $\mathbb{S}$. Эти правила интерпретации образуют семантику языка логики предикатов.

Семантика

- Если x – переменная, то $\mathbf{I}(x) =_{def} \mathbf{I}_v(x)$.
- Если f – функциональная n–местная константа и $t_1,\ldots,t_n$ – термы, то $\mathbf{I}(f(t_1,\ldots,t_n)) =_{def} \mathbf{I}_c(f)(\mathbf{I}(t_1),\ldots,\mathbf{I}(t_n))$.
- Если P – предикатная m-местная константа и $t_1,\ldots,t_m$ – термы, то $\mathbf{I}(P(t_1,\ldots,t_m)) =_{def} \mathbf{I}_c(P)(\mathbf{I}(t_1),\ldots,\mathbf{I}(t_m))$.
- Если s и t – термы, то $\mathbf{I}(s=t)$ есть **И**, если $\mathbf{I}(s)=\mathbf{I}(t)$, в противном случае это будет **Л**.
- Если A и B – формулы, то $\neg A$, $(A \wedge B)$, $(A \vee B)$, $(A \supset B)$ и $(A \equiv B)$ интерпретируются так же, как в исчислении высказываний.

- Если A – формула и x – переменная, то $\mathbf{I}(\forall xA)$ есть **И**, если $\mathbf{I}_{x/d}(A)$ есть **И** для всех элементов d из $\mathbb{S}$.
- Если A – формула и x – переменная, то $\mathbf{I}(\exists xA)$ есть **И**, если $\mathbf{I}_{x/d}(A)$ есть **И** хотя бы для одного элемента d из $\mathbb{S}$.

Формула A исчисления предикатов называется *истинной при интерпретации* **I**, если $\mathbf{I}(A) = \mathbf{И}$. Понятия общезначимой, выполнимой, невыполнимой формул, тавтологии, логического следствия и модели для формулы, теоремы, доказательства, правила вывода, аксиоматической системы, которые были введены в контексте логики высказываний, таким же образом определяются и в логике предикатов.

Аксиоматическую систему логики высказываний можно приспособить к исчислению предикатов без равенства. Схема аксиом (A1), (A2), (A3) и правило (***MP*** – *modus ponens*) остаются без изменений. Дополнительные две схемы и правило вывода (G – правило обощения) позволяют манипулировать с кванторами.

Введем сокращение, используемое при записи схем. Будем говорить, что *терм t свободен для переменной x в формуле Q, если ни x, ни произвольная переменная из t не квантифицированы в Q. При этом через* $Q_{x/t}$ *обозначается формула, полученная путем одновременной замены всех вхождений x на t.*

Аксиоматическая система

(A1) $(P\supset (Q\supset P))$,

(A2) $((P\supset (Q\supset R))\supset ((P\supset Q)\supset (P\supset R)))$,

(A3) $((\neg P\supset\neg Q)\supset ((\neg\neg P\supset Q)\supset P))$,

(A4) $\forall x(P\supset Q)\supset (P\supset\forall xQ)$,

(x не содержится в P и не является связанной в Q).

(A5) $\forall xQ\supset Q_{x/t}$) ($x$ свободна для t в Q).

(MP) Если X и $(X\supset Y)$ – теоремы, то Y – теорема.

(G) Если P – теорема и x не связана в P, то $\forall xP$ – теорема.

В этом контексте квантор существования не вводится, т. к. формула $\exists xA$ считается сокращением для $\neg\forall x\neg A$.

Модель логики первого порядка

Семантика, определенная в предыдущем параграфе, базируется на понятии *интерпретации*. Семантические правила заимствованы на теории логики предикатов, изложенной в [7]. Можно определить семантику, опи-

раясь на понятие модели; оба этих определения эквивалентны; они отличаются лишь видами используемых словарей и обозначениями. Поскольку семантика обычной логики определяется на понятии модели, полезно развить такой же подход в теории предикатов первого порядка без равенств. *Логикой первого порядка* (или *теорией первого порядка*) называется логика (т. е. множество формул), основанная на исчислении предикатов.

Правила интерпретации из предыдущего параграфа определяют семантику языка предикатов. Эти правила базируются на интерпретации **I**, заданной как тройка ($\mathbb{S}$, $\mathbf{I}_c$, $\mathbf{I}_v$), где $\mathbb{S}$ – непустое множество элементов (т. е. область интерпретации), $\mathbf{I}_c$ – функция (т. е. оценка), которая приписывает определенные значения предикатным и функциональным константам, $\mathbf{I}_v$ – функция (назначения), которая каждой индивидной переменной ставит в соответствие некий элемент из S.

Моделью **M** логики первого порядка *L* (без равенства) называется пара ($\mathbb{S}$, V), где $\mathbb{S}$ – область интерпретации и V – функция, совпадающая с функцией $\mathbf{I}_c$ интерпретации **I**. Роль функции V состоит в том, чтобы интерпретировать функциональные и предикатные константы языка в терминах элементов области $\mathbb{S}$.

Определение семантики в рамках этой модели требует еще и определения некой функции, которая присваивает значения переменным языка. Эта функция обозначается через *g*; она соответствует функции $\mathbf{I}_v$ интерпертации **I**. (Следует заметить, что функция присваивания *g* не является частью модели **M**. Эта функция никоим образом не входит в способ интерпретации основных констант языка).

Семантика формул из *L* описывается формой $\mathbf{M} \models_g A$, что означает «*A истинна в модели* **M** *для присваивания g*».

Если α – выражение из *L*, то через $\ulcorner \alpha \lrcorner^{\mathbf{M},g}$ будет обозначаться семантическое значение выражения α в модели **M** для присваивания *g*. Итак, имеем следующие эквивалентные обозначения для формулы A из логики *L*:

$$\mathbf{M} \models_g \mathrm{A} \Leftrightarrow \ulcorner \mathrm{A} \lrcorner^{\mathbf{M},g} = 1,$$

$$\mathbf{I}(\mathrm{A}) = 1 \Leftrightarrow \ulcorner \mathrm{A} \lrcorner^{\mathbf{M},g} = 1.$$

Семантика

- Если *x* – переменная, то $\ulcorner x \lrcorner^{\mathbf{M},g} =_{def} g(x)$.
- Если *f* является *n*-местной функциональной константой и $t_1,\ldots,t_n$ – термы, то $\ulcorner f(t_1, \ldots, t_n) \lrcorner^{\mathbf{M},g} =_{def} \mathrm{V}(f)(\ulcorner t_1 \lrcorner^{\mathbf{M},g}, \ldots, \ulcorner t_n \lrcorner^{\mathbf{M},g})$.

- Если P является n–местной предикатной константой и $t_1,\ldots,t_n$ – термы, то $\ulcorner P(t_1,\ldots,t_n)\lrcorner^{\mathbf{M},g} =_{def} \mathrm{V}(P)(\ulcorner t_1\lrcorner^{\mathbf{M},g}, \ldots,\ulcorner t_n\lrcorner^{\mathbf{M},g})$.
- $\mathbf{M} \models_g P(t_1,\ldots,t_n)$ тогда и только тогда, когда $\ulcorner P(t_1,\ldots,t_n)\lrcorner^{\mathbf{M},g}$ *истинна*.
- $\mathbf{M} \models_g \neg A$ тогда и только тогда, когда $\mathbf{M} \not\models_g A$.
- $\mathbf{M} \models_g A \wedge B$ тогда и только тогда, когда $\mathbf{M} \models_g A$ и $\mathbf{M} \models_g B$.
- $\mathbf{M} \models_g A \vee B$ тогда и только тогда, когда $\mathbf{M} \models_g A$ или $\mathbf{M} \models_g B$.
- $\mathbf{M} \models_g A \supset B$ тогда и только тогда, когда из того, что $\mathbf{M} \models_g A$ импликационно следует $\mathbf{M} \models_g B$.
- $\mathbf{M} \models_g A \equiv B$ тогда и только тогда, когда из того, что $\mathbf{M} \models_g A$ импликационно следует $\mathbf{M} \models_g B$, и из того, что $\mathbf{M} \models_g B$ импликационно следует $\mathbf{M} \models_g A$.
- $\mathbf{M} \models_g \forall xA$ тогда и только тогда, когда $\mathbf{M} \models_{g`} A$ для всех присваиваний $g`$, которые отличаются от присваивания g только сопоставлением значения переменной x.

Формула A общезанчима тогда и только тогда, когда $\mathbf{M} \models_g A$ для всех моделей $\mathbf{M}$ и всех присваиваний g.

1.2. Модальная логика

В отличие от языка классической логики выскзываний в языке модальной логики высказываний испоьзуются два дополнительных символа: $\Box$ и $\Diamond$. Они называются соответственно *модальным оператором общности* и *модальным оператором существования*. Эти операторы воздействуют на формулы логики высказываний, изменяя их смысл.

Синтаксис

- Все синтаксические правила логики высказываний являются также синтаксическими правилами модальной логики.
- Если A – формула, то $\Box A$ и $\Diamond A$ – формулы.

  На операторы $\Box$ и $\Diamond$ накладывается соотношение двойственности:

$$\Diamond A =_{def} \neg\Box\neg A. \qquad (1.1)$$

Эту связь можно рассматривать как определение оператора $\Diamond$ через оператор $\Box$, так что модальные формулы всегда можно записывать, используя лишь оператор $\Box$. Точно так же, как связки логики высказываний могут быть «прочитаны» в естественном языке, модальным опера-

торам $\Box$ и $\Diamond$ обычно присваивают определенные значения. В действительности, каждому из этих операторов можно придать бесконечное число значений, которые должны попарно соответствовать друг другу согласно соотношению (1.1). Соответствие, существующее между некоторыми прочтениями формул $\Box A$ и $\Diamond A$, проиллюстрировано таблицей, представленной на рис. 1.2.

Трактовки оператора $\Box A$	**Трактовки оператора** $\Diamond A$
Необходимо, чтобы *A*	Возможно, что *A*
Всегда будет истинно, что *A*	Иногда будет истинно, что *A*
Требуется, чтобы *A*	Разрешается, чтобы *A*
Предполагается, что *A*	Противоположное к *A* не предполагается
Известно, что *A*	Противоположное к *A* неизвестно
Любое выполнение программы дает результат *A*	Существует такое выполнение программы, которое дает результат *A*

Рис. 1.2. Словесное выражение модальных операторов

Для осуществления семантического анализа модальных формул потребуются следующие понятия.

Структурой называется пара $\mathbb{F}=(\mathbb{W},\mathcal{R})$, где $\mathbb{W}$ – непустое множество, а $\mathcal{R}$ – бинарное отношение на множестве $\mathbb{W}$, т. е. некоторое подмножество из $\mathbb{W}\times\mathbb{W}$. Элементы из $\mathbb{W}$ называются «*точками*».

Пусть $\mathcal{P}$ – множество атомов (или высказываний) модального языка $\mathcal{L}$. Тогда *$\mathcal{P}$-моделью* на структуре $(\mathbb{W},\mathcal{R})$ называется тройка $\mathbf{M}=(\mathbb{W},\mathcal{R},\mathcal{V})$, где $\mathcal{V}$ – отображение из $\mathcal{P}$ в $2^{\mathbb{W}}$ (множество всех подмножеств множества $\mathbb{W}$), сопоставляющее каждому высказыванию p из $\mathcal{P}$ подмножество $\mathcal{V}$(p) из $\mathbb{W}$. Иначе $\mathcal{V}(p)$ интерпретируется как множество точек из $\mathbb{W}$, в которых высказывание p оказывается истинным. Если контекст не приводит ни к какой двусмысленности, то префикс $\mathcal{P}$ в определении модели опускают и говорят просто о *модели*.

Пусть w – элемент из $\mathbb{W}$ и A – модальная формула в языке $\mathcal{L}$. Запись $\mathbf{M} \models_w A$ означает семантику формулы A и означает, что «A истинно в точке w в модели **M**». Эта семантика определяется следующим образом.

Семантика

$\mathbf{M} \not\models_w \mathbf{Л}$;

$\mathbf{M} \models_w p$, если $w \in \mathcal{V}(p)$;

$\mathbf{M} \models_w A_1 \supset A_2$, если из $\mathbf{M} \models_w A_1$ следует $\mathbf{M} \models_w A_2$;

$\mathbf{M} \models_w \Box A$, если из $w\mathcal{R}t$ вытекает, что $\mathbf{M} \models_t A$ для всех $t \in \mathbb{W}$.

Последнее правило выражает тот факт, что формула $\Box A$ истинна в точке w модели **М**, если формула A истинна во всех точках t, находящихся в отношении $\mathcal{R}$ с точкой w. Эти определения задают базовые семантические отношения.

Семантическая оценка (семантическое означивание) формул **И**, $\neg A$, $\Diamond A$, $(A_1 \vee A_2)$, $(A_1 \wedge A_2)$ и $(A_1 \equiv A_2)$ получается из базовых семантических отношений с помощью эквивалентных преобразований, позволяющих записать константу **И**, оператор $\Diamond$ и связки $\vee$, $\wedge$, $\equiv$ в виде формул, построенных из константы **Л**, оператора $\Box$ и связок $\neg$ и $\supset$. Например, из соотношений $\neg A =_{def} (A \supset \text{Л})$ и (2.1) выводятся следующие семантические правила:

$\mathbf{M} \models_w \neg A$, если $\mathbf{M} \mid\neq_w A$;

$\mathbf{M} \models_w \Diamond A$, если $M \models_t A$ хотя бы для одного $t \in \mathbb{W}$, такого, что $w \mathcal{R} t$.

С помощью определения истинности модальных формул посредством семантических правил, можно ввести понятия формулы, *истинной в модели*, формулы, *истинной в структуре*, и *общезначимой* формулы.

- Формула *А истинна в модели* **М**, если она истинна во всех точках этой модели, т. е. если $\mathbf{M} \models_w$ А для всех $w \in \mathbb{W}$. Обозначение: $\mathbf{M} \models A$.
- Формула А *истинна в структуре* $\mathbf{F} = (\mathbb{W}, \mathcal{R})$, если А истинна в любой модели $(\mathbb{W}, \mathcal{R}, \mathcal{V})$, т. е. если $\mathbf{M} \models A$ для всех моделей $\mathbf{M} = (\mathbb{W}, \mathcal{R}, \mathcal{V})$. Это обозначается следующим образом: $\mathbf{F} \models A$.
- Формула *А общезначима*, если она истинна во всех структурах $(\mathbb{W}, \mathcal{R})$. Это обозначается так: $\models A$.

Примеры и приложения

Следующие формулы истинны во всех моделях и, следовательно, истинны во всех структурах:

$\Box(A \supset B) \supset (\Box A \supset \Box B)$,
$\Diamond(\mathrm{A} \supset \mathrm{B}) \supset (\Diamond \mathrm{A} \supset \Diamond \mathrm{B})$,
$\Box(A \wedge B) \equiv (\Box A \wedge \Box B)$,
$\Diamond(A \vee B) \equiv (\Diamond A \vee \Diamond B)$,
$\Box(A \supset \Diamond(B \supset C)) \supset \Diamond(B \supset (\Box A \supset \Diamond C))$.

Докажем первую формулу методом от противного, т. е. найдем модель, в которой эта формула ложна [4, стр. 37–38]. Импликативная фор-

мула ложна только в том случае, когда ее антецедент истинный, а консеквент ложный. Значит, надо найти модель, в которой формула $\Box(A \supset B)$ *истинна* и формула $(\Box A \supset \Box B)$ *ложна*. Применяя аналогичные рассуждения к консеквенту исходной формулы, получаем дополнительные ограничения: $\Box A$ *истинна*, $\Box B$ *ложна*.

Условие «$\Box A$ *истинна*» влечет условие «*A истинна* во всех точках», которое, будучи соединенным с условием «$\Box(A \supset B)$ *истинна*», дает: формула «$\Box B$ *истинна* во всех точках». Но это не может иметь место в силу второго ограничения, т. к. $\Box B$ *ложно*. Значит, утверждение доказано.

Следующие формулы истинны, но не во всех структурах:

$\Box A \supset A$,
$\Box A \supset \Box\Box A$,
$\Box(A \vee B) \supset (\Box A \supset \Box B)$.

В таблице, изображенной на рис. 2.1, указаны различные «*возможные прочтения*» модальных формул $\Box A$ и $\Diamond A$. Эти разнообразные прочтения лежат в основе разных применений модальной логики; здесь вводится терминология *семантики различных миров*. Непустое множество $\mathbb{W}$, фигурирующее в определении модели, называется *универсумом*. Элементы $w \in \mathbb{W}$, ранее называемые «*точками*», будут теперь называться *возможными мирами универсума*. Бинарное отношение $\mathcal{R}$ будем называть *отношением достижимости*. Итак, структура состоит из множества $\mathbb{W}$ возможных миров, связанных отношением достижимости $\mathcal{R}$. Если два мира *s* и *t* принадлежат универсуму $\mathbb{W}$, то достижимость мира *t* из мира *s* обозначается так: $s\mathcal{R}t$.

Логики возможного и необходимого

Предполагается, что отношение достижимости $\mathcal{R}$ *рефлексивно*, т. е. *sRs* справедливо $\forall s \in \mathbb{W}$. *Необходимой истиной* в мире *s* называется формула, которая подтверждается во всех мирах *t*, достижимых из *s*. Формула $\Box A$ читается так: «*A необходимо истинна*».

Возможной истиной в мире *s* называется формула, которая подтверждается хотя бы в одном из миров, достижимых из *s*. Формула $\Diamond A$ читается так: «*возможно, что A истинна*». Часто выделяется один из миров универсума в качестве «*реального мира*». Формула *A*, которая необходимо истинна в реальном мире, будет истинна и во всех мирах, достижимых из реального мира, а значит, в частности, и самом реальном мире, т. е. верно:

$\Box A \supset A$.

Если формула A истинна в реальном мире, то она будет истинна хотя бы в одном из миров, достижимых из реального мира. Значит, формула

$$A \supset \Diamond A$$

истинна в рассматриваемой структуре.

Можно рассматривать различные виды необходимости [6, стр. 39]. Например, *логическая необходимость* отличается от *физической необходимости*, в рамках которой формула $\Box A$ означает: «*A является следствием законов физики*». Если t есть абсолютная температура, то формула $\Box(t > -273)$ истинна в реальном мире при условии, что «$\Box$» означает физическую необходимость. На практике полагают, что законы физики верны во всех мирах, достижимых из реального мира, однако логически возможен вариант ложности формулы $(t > -273)$.

Временные логики

Во временной логике возможные миры представляют состояния некоторого мира в различные моменты его эволюции. Реальным миром является состояние рассматриваемого мира в некоторый исходный момент, принятый за точку отсчета. Отношение достижимости $s\mathcal{R}t$ указывает на то, что «*t следует за s*». Формула $\Box A$ означает так: «*во все будущие моменты произойдет A*», тогда как формула $\Diamond A$ читается: «*по крайней мере в один из будущих моментов совершится A*». В случае, когда отношение достижимости $s\mathcal{R}t$ указывает на то, что «*t предшествует s*», то формулы $\Box A$ и $\Diamond A$ означают соответственно «*во все прошлые моменты было A*» и «*по крайней мере в один прошлый момент было A*». Для временной логики являются естественными те структуры $(\mathbb{W}, \mathcal{R})$, у которых $\mathbb{W}$ – одно из множеств **N** (натуральные числа), **Z** (целые числа), **Q** (рациональные числа) или **R** (действительные числа), а отношение $\mathcal{R}$ совпадает с одним из линейных порядков $\leq, <, \geq, >$.

Классический образ времени – образ, используемый в ньютоновской физике [4, стр. 40]; в ней время рассматривается как «*линейный одномерный континуум*».

Динамические логики

Динамическая логика основана на сопоставлении некоторого модального оператора каждой команде некоего языка программирования. В этом контексте множество $\mathbb{W}$ интерпретируется как совокупность возможных состояний в ходе вычислений. Отношение $s\mathcal{R}t$ указывает на то, что вычисления начинаются в некотором состоянии s и заканчиваются в состоянии t. Недетерминированная программа может приводить к многочисленным результатам. Формула $\Box A$ означает тогда, что «*все выполнения программы*

оканчиваются тем, что A истинна». Формула $\Diamond A$ означает, что «*хотя бы одно выполнение программы оканчивается тем, что A истинна*».

Логика веры и знания

Если главная функция модальной логики состоит в формализации модальностей «*необходимости*» и «*возможности*», то одним из прочих направлений ее преимущественного применения являются моделирование и анализ парадигм «знания» и «*веры*». С этой целью различные логические системы используют формальные языки, содержащие операторы «*веры*» и «*знания*». В рамках логик веры и знания оператор $\Box$ принимает соответственно значения «*предполагается*» и «*известно*». Двойственный ему оператор $\Diamond$ принимает соответственно значения «*противоположное не предполагается*» и «*противоположное неизвестно*». Такие логики рассматриваются в [7, гл. 4].

Бинарные отношения и схемы формул

Рассмотрим список, содержащий перечень интересных свойств, которыми может обладать бинарное отношение $\mathcal{R}$ [4, стр. 41].

1.	Рефлексивность	$\forall s\,(s\mathcal{R}s)$
2.	Симметричность	$\forall s \forall t\,(s\mathcal{R}t \supset t\mathcal{R}s)$
3.	Репродуктивность	$\forall s \exists t\,(s\mathcal{R}t)$
4.	Транзитивность	$\forall s \forall t \forall u\,(s\mathcal{R}t \wedge t\mathcal{R}u \supset s\mathcal{R}u)$
5.	Евклидовость	$\forall s \forall t \forall u\,(s\mathcal{R}t \wedge s\mathcal{R}u \supset t\mathcal{R}u)$
6.	Частичная функциональность	$\forall s \forall t \forall u\,(s\mathcal{R}t \wedge s\mathcal{R}u \supset t = u)$
7.	Функциональность	$\forall s \exists !t\,(s\mathcal{R}t)$
8.	Слабая плотность	$\forall s \forall t\,(s\mathcal{R}t \supset \exists u\,(s\mathcal{R}u \wedge u\mathcal{R}t))$
9.	Слабая связность	$\forall s \forall t \forall u\,(s\mathcal{R}t \wedge s\mathcal{R}u \supset t\mathcal{R}u \vee t = u \vee u\mathcal{R}t)$
10.	Слабая направленность	$\forall s \forall t \forall u\,(s\mathcal{R}t \wedge s\mathcal{R}u \supset \exists v\,(t\mathcal{R}v \wedge u\mathcal{R}v))$

Этому списку свойств соответствует список схем формул.

1. $\Box A \supset A$	(T)
2. $A \supset \Box\Diamond A$	(B)
3. $\Box A \supset \Diamond A$	(D)
4. $\Box A \supset \Box\Box A$	(4)
5. $\Diamond A \supset \Box\Diamond A$	(5)
6. $\Diamond A \supset \Box A$	
7. $\Diamond A \equiv \Box A$	
8. $\Box\Box A \supset \Box A$	
9. $\Box(A \wedge \Box B \supset B) \vee \Box(B \wedge \Box B \supset A)$	
10. $\Diamond\Box A \supset \Box\Diamond A$	

Некоторые из приведенных схем формул принимаются в качестве схем аксиом в аксиоматических системах – этот факт выражает теорема.

Теорема 1.1. Если задана структура **F**=*(*$\mathbb{W}$*, *$\mathcal{R}$*), то отношение *$\mathcal{R}$ *тогда и только тогда обладает конкретным из свойств* 1–10*, когда соответствующая схема формул истинна в* **F**.

Теорема 1.1 имеет фундаментальное значение относительно реляционной семантики, введенной Крипке; такие структуры хорошо приспособлены для применений. Также важные свойства отношения R, входящего в определение модели, отражаются многочисленными модальными схемами. Однако есть и привычные свойства бинарного отношения R, не соответствующие общезначимости никакой модальной схемы. Например:

Иррефлективность	$\forall s \neg (s \mathcal{R} s)$
Антисимметричность	$\forall s \forall t \, (s \mathcal{R} t \wedge t \mathcal{R} s \supset s = t)$
Асимметричность	$\forall s \forall t \, (s \mathcal{R} t \supset \neg (t \mathcal{R} s))$

Оценки и тавтологии

Введем набор понятий, служащих для строго определения классических аксиоматических систем модальной логики.

Множество *Sf*(*A*) всех *подформул* формулы *A* определяется посредством обычной рекурсии:

- Если *p* – атом, то *Sf*(*p*) = {*p*}.
- *Sf*(**Л**) = {**Л**}.
- $Sf(A_1 \supset A_2) = \{A_1 \supset A_2\} \cup Sf(A_1) \cup Sf(A_2)$.
- $Sf(\Box A) = \{\Box A\} \cup Sf(A)$.

Пусть заданы модель **M**, элемент $s \in \mathbb{W}$ и множество высказываний **P**, тогда оценка (функция оценки, функция означивания) $\mathcal{V}_s : \mathbf{P} \to \{$*истинно*, *ложно*$\}$ определяется как:

$\mathcal{V}_s$(p) = *истинно*, если $s \in \mathcal{V}$(p),
$\mathcal{V}_s$(p) = *ложно*, если $s \notin \mathcal{V}$(p).

Функция $\mathcal{V}_s$ дает оценку атомарным формулам (атомам) множества **P**; эта функция продолжается на множество всех формул, построенных из высказываний, принадлежащих **P**, и логических связок (за исключением оператора $\Box$) посредством семантических правил логики высказываний.

Итак, модель **M** = ($\mathbb{W}$, $\mathcal{R}$, $\mathcal{V}$) на данной структуре служит основой множества $\{V_s : s \in \mathbb{W}\}$ оценок для **P**; наоборот, такое множество определяет модель, для которой функция V задается отношением $\{\mathcal{V}_s : s \in \mathbb{W} : \mathcal{V}_s$ (p) = *истинно*$\}$.

Формула A называется *квазиатомарной*, если она либо атомарна (т. е. $A \in \mathbf{P}$), либо начинается с символа $\Box$ (т. е. если $A = \Box B$, где B – формула).

Пусть $\mathbf{P}^q$ множество всех квазиатомарных формул. Любую формулу A можно построить из элементов множества $\mathbf{P}^q \cup \{\mathbf{Л}\}$ и связки $\supset$. Используя семантическое определение этой связки ($X \supset Y$) *истинна* $\Leftrightarrow$ (X *ложна* или Y *истинна*), и оценки квазиатомарных формул

$$V_s : \mathbf{P} \to \{\textit{истинно, ложно}\}, \text{ где } s \in \mathbb{W},$$

легко распространить оценки $\mathcal{V}_s$ на все формулы A модального языка $\mathcal{L}$, порожденного множеством **P**. Таким способом получаются оценки

$$\mathcal{V}_s : \mathcal{L} \to \{\textit{истинно, ложно}\}.$$

Формула $A \in \mathcal{L}$ является тавтологией, если $\mathcal{V}_s(A)$ = истинно для всех оценок $\mathcal{V}_s$ ее квазиатомарных подформул.

Можно доказать ([6, стр. 44]), что всякая модальная тавтология получается из подходящей тавтологии логики высказываний (логики, не содержащей оператора $\Box$) путем одновременной подстановки в нее соответствующих формул. Например, модальная тавтология

$$(\Box(p \supset \Box q) \supset \Box \neg r) \supset (\neg \Box \neg r \supset \neg \Box(p \supset \Box q))$$

может быть получена из следующей тавтологии логики высказываний:

$$(p \supset q) \supset (\neg q \supset \neg p).$$

Подходящая подстановка такова:

p заменяется на $\Box(p \supset \Box q)$, а q заменяется на $\Box \neg r$.

Логики

При наличии языка $\mathcal{L}$, основанного на множестве атомов **P**, *логика* (*теория*) $\mathbf{\Lambda}$ определяется как подмножество формул, порожденных на основе **P** и удовлетворяющим следующим условиям:

- содержит все тавтологии;
- если A и $A \supset B$ – элементы из Λ, то B – элемент из $\mathbf{\Lambda}$;
- если A – элемент из $\mathbf{\Lambda}$, то всякая формула A`, получаемая из A одновременной подстановкой в нее формул вместо атомов, тоже является элементом из $\mathbf{\Lambda}$.

Двумя примерами логик служат сам язык $\mathcal{L}$ и все тавтологии из $\mathcal{L}$.

Элементы логики называются *теоремами*. Пишется $\vdash_{\mathbf{\Lambda}} A$ для обозначения

$$\vdash_{\mathbf{\Lambda}} A \Leftrightarrow A \in \Lambda.$$

Пусть ζ – модель или структура. Тогда логика $\mathbf{\Lambda}$ называется *адекватной* по отношению к ζ, если любая теорема A из $\mathbf{\Lambda}$ является формулой, истинной в ζ, т. е. если верно

$$\vdash_{\mathbf{\Lambda}} A \Rightarrow \zeta \models A.$$

Логика $\mathbf{\Lambda}$ называется полной по отношению к ζ, если каждая формула A, истинная в ζ, является теоремой логики $\mathbf{\Lambda}$, т. е. если справедливо отношение

$$\zeta \models A \Rightarrow \vdash_{\mathbf{\Lambda}} A.$$

Логика $\mathbf{\Lambda}$ называется детерминированной посредством ζ, если она одновременно адекватна и полна по отношению к ζ, т. е. если

$$\vdash_{\mathbf{\Lambda}} A \Leftrightarrow \zeta \models A.$$

Логика Λ нормальна, если она содержит схему формулы:

$$\mathrm{K}: \Box(A \supset B) \supset (\Box A \supset \Box B),$$

и снабжена *модальным правилом вывода необходимости*:

$$\vdash_{\mathbf{\Lambda}} A \Rightarrow \vdash_{\mathbf{\Lambda}} \Box A.$$

Приведенное правило утверждает, что если A – теорема логики $\mathbf{\Lambda}$, то и $\Box A$ – теорема из $\mathbf{\Lambda}$; это записывается так:

$$A \vdash_{\mathbf{\Lambda}} \Box \mathrm{A}.$$

Если $\{\mathbf{\Lambda}_i \mid i \in \mathrm{I}\}$ – какое-либо множество нормальных логик, то их пересечение

$$\bigcap \{\mathbf{\Lambda}_i \mid i \in \mathrm{I}\}$$

является нормальной логикой. Например, логика $\mathbf{K}$, определенная соотношеним:

$$\mathrm{K} = \{\Lambda_{\mathrm{I}} \mid \Lambda_{\mathrm{i}} \text{ есть нормальная логика}\},$$

т. е. пересечение всех нормальных логик есть наименьшая нормальная логика. Следующая теорема показывает важность этой «*минимальной*» логики.

Теорема 1.2. Формула A *является теоремой логики* **K** *тогда и только тогда, когда* A *общезначима (т. е. истинна во всех структурах)*.

Следующие отношения выполняются в любой нормальной логике:

$\vdash_{\mathbf{\Lambda}} A \supset B \Rightarrow \vdash_{\mathbf{\Lambda}} \Box A \supset \Box B$ и $\vdash_{\mathbf{\Lambda}} \Diamond A \supset \Diamond B$,

$\vdash_{\mathbf{\Lambda}} A \equiv B \Rightarrow \vdash_{\mathbf{\Lambda}} \Box A \equiv \Box B$ и $\vdash_{\mathbf{\Lambda}} \Diamond A \equiv \Diamond B$,

$\vdash_{\mathbf{\Lambda}} \Box A \wedge \Box B \equiv \Box(A \wedge B)$,

$\vdash_{\mathbf{\Lambda}} \Diamond(A \vee B) \equiv \Diamond A \vee \Diamond B$,

$\vdash_{\mathbf{\Lambda}} \Box A \vee \Box B \supset \Box(A \vee B)$,

$\vdash_{\mathbf{\Lambda}} \Diamond(A \wedge B) \equiv \Diamond A \wedge \Diamond B$.

Аксиоматические системы

Аксиоматическая система (для) нормальной логики состоит из трех следующих элементов:

- Некоторая аксиоматическая система логики высказываний.
- Схема аксиом $\Box(A \supset B) \supset (\Box A \supset \Box B)$, обозначаемая K.
- Модальное правило вывода необходимости.

Обычно принято представлять наименьшую нормальную логику, содержащую схемы $\Sigma_1, \ldots, \Sigma_n$ с помощью обозначения $\mathbf{\Lambda} = K\,\Sigma_1 \ldots \Sigma_n$.

Эта логика определяется так:

$\mathbf{\Lambda} = \bigcap \{\mathbf{\Lambda}_i | \Lambda_i$ нормальна и $\Sigma_1 \cup \ldots \cup \Sigma_n \subset \mathbf{\Lambda}_i\}$.

Классическими обозначениями для некоторых схем являются следующие:

D: $\Box A \supset \Diamond A$,
T: $\Box A \supset A$,
4: $\Box A \supset \Box\Box A$,
B: $A \supset \Box\Diamond A$,
5: $\Diamond A \supset \Box\Diamond A$,
L: $\Box((A \wedge \Box A) \supset B) \vee \Box((B \wedge \Box B) \supset A)$,
W: $\Box(\Box A \supset A) \supset \Box A$.

Для некоторых широко используемых логик широко применяются обозначения:

$S$4: $KT$4,
$S$5: $KT$45.

Теорема 1.3. Формула A является теоремой логики KT тогда и только тогда, когда A истинна во всех структурах, в которых R рефлексивно.
Теорема 1.4. Формула A является теоремой логики S4 тогда и только тогда, когда A истинна во всех структурах, в которых R рефлексивно и транзитивно.

Теорема 1.5. Формула A является теоремой логики S5 тогда и только тогда, когда A истинна во всех структурах, в которых R рефлексивно, транзитивно и евклидово (т. е. R – отношение эквивалентности).

Теоремы 1.3–1.5 обосновываются с использованием следующей леммы, сформулированной на содержательном уровне:

Логики *KT*, *S*4 и *S*5 важны в рамках формализации систем знания и веры, при которой модальный оператор принимает соответственно значения «*известно*» и «*предполагается*» [6, гл. 4]. По определению знание есть истинная информация. Итак, схема *T* требует, чтобы «то, что известно, является истинным»; эта схема добавляется к нормальной логике, если оператор означает «*известно*». Если же модальный оператор означает «предполагается», то схема 4 гласит: «*если я верю, что A подтверждается, то я верю, что я верю, что A подтверждается*»; эта способность к позитивной интроспекции необходима для формализации совершенного интроспективного интеллекта. Схема 5 формализует совершенную негативную интроспекцию; она означает: «*если я не верю, что A подтверждается, то я верю, что я не верю, что A подтверждается*»; этим свойством выражено совершенное знание наших пределов веры.

Аксиоматические системы для логик *KT*, *S*4, *S*5 получаются добавлением соответственно схем *T*, 4 и 5 в качестве схем аксиом к аксиоматической системе нормальной логики.

Выбор модальной системы зависит от моделируемого понятия. Если нужно охарактеризовать знания некоего разумного субъекта, обладающего совершенной способностью к логической интроспекции по поводу того, что «*известно*» и что «*неизвестно*», то выбирают модальную систему *S*5. Если же надо моделировать предположения некоторого идеального разумного субъекта (т. е. такого субъекта, кое-какие предложения которого могут показаться ошибочными, но который тем не менее обладает совершенной способностью к логической интроспекции относительно того, во что он верит и во что не верит), то будет выбрана система *K*45; ее называют также *слабой S*5 системой.

Каждая из этих различных модальных систем индуцирует синтаксическое отношение выводимости, которое ей свойственно; оно обозначается символом $\vdash_S$, где *S* – название рассматриваемой модальной системы.

Мультимодальные языки

Существуют языки, содержащие достаточно большой набор модальных операторов и использующие совокупность символов {[i] |

i∈Ind}, сопоставляя каждый ее элемент некоторому оператору общности (модальному квантору). Оператор существования, двойственный оператору [i], обозначается символом <i> и определяется формой вида ¬[i]¬. Если в логике более одного модального оператора (общности), то она называется мультимодальной.

Синтаксис

Все синтаксические правила логики высказывания есть синтаксические правила мультимодальной логики.

Если A – формула, то [i]A и <i>A – формулы.

Структура **F** для модального языка определяется следующим образом: **F** = ($\mathbb{W}$, $\{\mathcal{R}_i \mid i \in \mathrm{Ind}\}$), где $\mathbb{W}$ – непустое множество и $\{\mathcal{R}_i\}$ – набор бинарных отношений $\mathcal{R}_i \subset \mathbb{W} \times \mathbb{W}$ для соответствующих модальных кванторов. *Модель* **M**=(**F**, $\mathcal{V}$) на структуре **F** определяется, как и прежде, через отображение $\mathcal{V}$ из **P** в $2^{\mathbb{W}}$, где **P** – множество атомов данного языка.

Семантика

Семантика мультимодального языка определяется почти так же, как в разд. 2, отличие заключается в том, что последнее из приводимых правил заменяется нижеследующим множеством правил (где $i \in \mathrm{Ind}$).

M $\models_w$ s[i]A, если для всякого $t \in \mathbb{W}$ из $w\mathcal{R}_i t$ следует **M** $\models_t A$.

Определения *формулы* (A), *истинной в модели*, *истинной в структуре* (**F**$\models A$) и *общезначимой* ($\models A$), остаются неизменными, к тому же логика Λ опять определяется как подмножество мультимодального языка. Оно содержит все тавтологии и замкнуто относительно правил *modus ponens* и одновременной подстановки.

Логика Λ называется *нормальной*, если она содержит схемы теорем:

$$K_i\colon [i](A \supset B) \supset ([i]A \supset [i]B)$$

и если ей приданы правила необходимости

$$\vdash_\Lambda A \Rightarrow \vdash_\Lambda [i]A$$

(для всех $i \in \mathrm{Ind}$). Наименшая нормальная логика обозначается через K_{Ind}.

Временная логика

Язык временной логики, рассматриваемой в этом параграфе, получается из языка классической логики высказываний путем добавления двух

модальных операторов [*Б*] и [*П*], означающих соответственно «*всегда в будущем*» и «*всегда в прошлом*». Им отвечают модальные операторы существования <*Б*> и <*П*>, соответственно означающих «*иногда в будущем*» и «*иногда в прошлом*». Структура для этого языка имеет вид $(\mathbb{W},\mathcal{R}_{\text{Б}},\mathcal{R}_{\text{П}})$. Если **M** – модель, то $\mathbf{M}\models_s [Б]A$, если из $s\mathcal{R}_{Б}t$ следует $\mathbf{M}\models_t A$ для всех $t\in\mathbb{W}$, $\mathbf{M}\models_s [П]A$, если из $s\mathcal{R}_{П}t$ следует $\mathbf{M}\models_t A$ для всех $t\in\mathbb{W}$.

Свойства $s\mathcal{R}_{Б}t$ и $s\mathcal{R}_{П}t$ интерпретируются, соответственно, как «*t находится в будущем относительно s*» и «*t находится в прошлом относительно s*». Очевидно, требуется, чтобы *t* было в прошлом относительно *s* в том и только том случае, когда *s* находится в будущем относительно *t*, т. е. $s\mathcal{R}_{П}t \Leftrightarrow t\mathcal{R}_{Б}s$, таким образом, отношения $\mathcal{R}_{П}$ и $\mathcal{R}_{Б}$ *взаимно обратны*.

При заданной структуре $F=(\mathbb{W},\mathcal{R}_{Б},\mathcal{R}_{П})$ можно доказать следующие утверждения:

- $\mathbf{F}\models A\supset [Б]<П>A \Leftrightarrow \forall s\forall t(s\mathcal{R}_{П}t\Rightarrow t\mathcal{R}_{П}s)$.
- $\mathbf{F}\models A\supset [П]<Б>A \Leftrightarrow \forall s\forall t(s\mathcal{R}_{П}t\Rightarrow t\mathcal{R}_{Б}s)$.
- Если нормальная логика $\mathbf{\Lambda}$ содержит схему теорем $A\supset [П]<Б>A$, то канонические отношения $\mathcal{R}_{П}^{\mathbf{\Lambda}}$ и $\mathcal{R}_{Б}^{\mathbf{\Lambda}}$ обладают свойством $s\mathcal{R}_{П}^{\mathbf{\Lambda}}t \Rightarrow s\mathcal{R}_{Б}^{\mathbf{\Lambda}}t$.
- Если нормальная логика $\mathbf{\Lambda}$ содержит схему теорем $A\supset [Б]<П>A$, то канонические отношения $\mathcal{R}_{П}^{\mathbf{\Lambda}}$ и $\mathcal{R}_{Б}^{\mathbf{\Lambda}}$ обладают свойством $t\mathcal{R}_{Б}^{\mathbf{\Lambda}}s \Rightarrow s\mathcal{R}_{П}^{\mathbf{\Lambda}}t$.

Эти предложения формализуют следующие факты.

- Если формула *A* истинна в настоящем, то существует такое прошлое, относительно которого *A* истинна в будущем.
- Если формула *A* истинна в настоящем, то существует такое будущее, относительно которого *A* истинна в прошлом.

Так как отношения $\mathcal{R}_{Б}$ и $\mathcal{R}_{П}$ определяются одно через другое, то можно выбрать одно из них в качестве первичного (обозначим его $\mathcal{R}$ без индекса) и использовать структуры $\mathbf{F}=(\mathbb{W},\mathcal{R})$, где $\mathcal{R}\subset\mathbb{W}\times\mathbb{W}$ и

$$\mathbf{M}\models_s [Б]A \Leftrightarrow (s\mathcal{R}t \Rightarrow \mathbf{M}\models_t A),$$

$$\mathbf{M}\models_s [Б]A \Leftrightarrow (s\mathcal{R}t \Rightarrow \mathbf{M}\models_t A).$$

Если на отношение $\mathcal{R}$ не наложено никаких ограничений, то получается *нормальная временная логика*. Аксиоматическая система состоит из трех частей:

- Аксиоматическая система классической логики высказываний.

- Схемы аксиом

  $K_Б$: $[Б](A \supset B) \supset ([Б]A \supset [Б]B)$,
  $K_П$: $[П](A \supset B) \supset ([П]A \supset [П]B)$,
  $C_Б$: $A[Б]<П>A$,
  $C_Б$: $A[П]<Б>A$.

- Правила необходимости $A \vdash [Б]A$, $A \vdash [П]A$.

Следовательно, нормальная временная логика является нормальной логикой с языком операторов $[Б]$ $[П]$. Наименьшая нормальная временная логика обозначается через K_t. Все остальные временные логики получаются из некоторой нормальной логики наложением ограничений на отношение $\mathcal{R}$. Наложение тех или иных ограничений на отношение $\mathcal{R}$ определяет набор формул, присоединяемых к нормальной аксиоматической системе.

Отношение $\mathcal{R}$ называется *линейным слева*, если

$$\forall t \forall s \forall u((t\mathcal{R}u \wedge s\mathcal{R}u) \supset t\mathcal{R}s \vee t=s \vee s\mathcal{R}t).$$

Это свойство означает, что если t предшествует u и s предшествует u, то либо t предшествует s, либо t совпадает с s, либо s предшествует t. Налагая на отношение $\mathcal{R}$ условия линейности слева и транзитивности, приходим к понятию «*древовидного времени*» («*branching time*»).

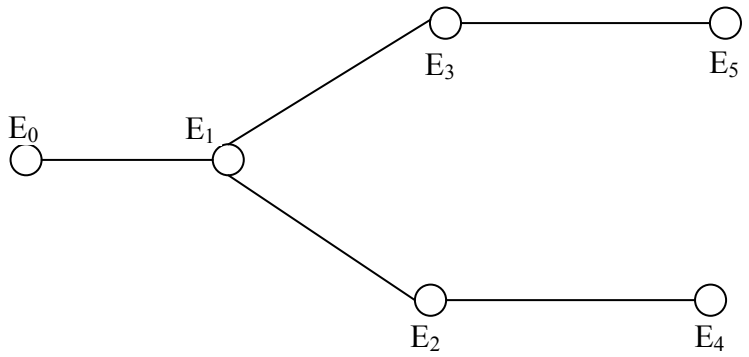

Рис. 1.3 Последовательности во временной древовидной структуре

Рассмотрим последовательность событий E_1, E_2, E_4; допустим, что она является некоторой чередой событий среди множества других физически возможных последовательностей; например, в это множество может входить последовательность E_1, E_3, E_5 (рис. 1.3). Основная идея «древовидности времени» состоит в том, что каждому событию E_i непосредственно предшествует единственное событие (прошлое однозначно), но непосредственно за ним может следовать много событий (будущее многозначно).

Свойство транзитивности отношения $\mathcal{R}$ обеспечивают две схемы аксиом:

$[Б]A \supset [Б][Б]A$,
$[П]A \supset [П][П]A$.

Свойство линейности слева описывается следующей схемой:

$([П](A \vee B) \wedge [П](A \vee [П]B) \wedge [П]([П]A \vee B)) \supset ([П]A \vee [П]B)$.

При помощи оператора существования $<П>$ эта схема аксиом записывается в эквивалентном виде:

$(<П>A \wedge <П>B) \supset (<П>(A \wedge B) \vee <П>(A \wedge <П>B) \vee <П>(B \wedge <П>A))$.

Полученная схема аксиом формализует следующий факт: если события A и B осуществились в прошлом, то либо они реализовывались одновременно, либо A предшествовало B, либо B предшествовало A.

Итак, аксиоматическую систему временной логики, формализующей понятие «*древовидного времени*», можно построить посредством добавления трех предыдущих схем аксиом к аксиоматической системе минимальной временной логики K_t. Полученная в результате логика обозначается K_b (от английского *branching*). Эта логика полна.

Динамическая логика

Язык динамической логики является мультимодальным языком, который каждой команде α некоторого языка программирования сопоставляет модальный оператор $[\alpha]$. При этом формула $[\alpha]$A означает «*после каждого завершения выполнения команды* α *формула A истинна*». Двойственная формула $<\alpha>$A означает «имеется хотя бы одно выполнение команды, которое заканчивается превращением формулы А в истинную». Этот тип модального оператора позволяет построить теорию, формализующую свойства составных программ с помощью модальных операторов и элементарных программ (называемых также «атомарными программами»). По сравнению с изученными прежде логическими языками язык динамической логики использует множество дополнительных основных символов α, которые представляют *элементарные программы*.

Пусть через П обозначено множество элементарных программ, а через P – множество высказываний данного языка.

Синтаксис

Все синтаксические правила мультимодальной логики являются также синтаксическими правилами мультимодальной логики. Модальные операторы суть $[\alpha]$ и $<\alpha>$, где α – некоторая программа.

Синтаксис программ.

- **Базис**: любая атомарная программа есть программа.

- **Индукционный шаг**: если α, α_1 и α_2 – программы и A – логическая формула, то $(\alpha_1 ; \alpha_2)$, $(\alpha_1 \cup \alpha_2)$, α^* и A? – тоже программы.
- **Ограничение**: программа получается только с помощью правил, описанных в базисе и индукционном шаге.

Операции над программами интерпретируются следующим образом:

Программа	**Значение**
$[\alpha]$A	A истинна после α,
$\alpha_1 ; \alpha_2$	выполнить α_1 и затем α_2,
$\alpha_1 \cup \alpha_2$	выполнить либо α_1, либо α_2 (недетерминированность),
α^*	выполнить α конечное число раз,
A?	тестировать A: если A истинна, то продолжать, иначе прервать.

Эти элементарные операции позволяют определить классические инструкции языков программирования:

if A then α else β $= _{def} (\text{A}?; \alpha) \cup (\neg \text{A}?; \beta)$,

while A do α $= _{def} (\text{A}?; \alpha)^*; \neg \text{A}?$,

repeat α until A $= _{def} \alpha ; (\neg \text{A}?; \alpha)^*$.

Моделью M для языка динамической логики называется тройка

$$M = (\mathbb{W}, \{R\alpha \mid \alpha \in \Pi\}, V),$$

где R_α – бинарное отношение на $\mathbb{W}$ и V – отображение из P в $2^{\mathbb{W}}$.

Семантика

$$M \vDash_s [\alpha]\text{A}, \text{ если } sR\alpha t \text{ влечет } M \vDash_t \text{A}.$$

Бинарное отношение $\text{R}\alpha$ должно отражать значения программ α. Модель M называется стандартной, если соответствующее бинарное отношение удовлетворяет следующим условиям:

$R\alpha_{;\beta} = \{(\text{s}, \text{t}) \mid \exists \text{u}(s\, R\alpha\, u \wedge u\, R\beta\, t)\}$,

$R\alpha_{\cup\beta} = R\alpha \cup R\beta$,

$R\alpha_* = (R\alpha)^*$ (рефлекисвное и транзитивное замыкание отношения $R\alpha$),

$R_{A?} = \{(\text{s}, \text{s}) \mid M \vDash_s \text{A}\}$.

Рассмотрим наименьшую нормальную логику, содержащую схемы аксиом:

Аксиомы

$[\alpha ; \beta]\text{A} \equiv [\alpha][\beta]\text{A}$,

$[\alpha \cup \beta] \equiv [\alpha]\text{A} \wedge [\beta]\text{A}$,

$[A?]B \equiv (A \supset B)$,
$[\alpha *]A \supset (A \wedge [\alpha][\alpha *]A)$,
$[\alpha *](A \supset [\alpha]A) \supset (A \supset [\alpha *]A)$.

Можно доказать, что эта логика определена классом стандартных моделей [4, стр. 57].

Логика веры знания

Очевидно, что логики, предметом исследования которых являются предположения или знания многих субъектов, суть мультимодальные логики. Формула [i]A означает «i-й субъект верит, что A» или «i-й субъект знает, что A» (подробнее см. [6, стр. 57, 71–75].

Модальная логика предикатов

Область модальных операторов □ и ◊ –это высказывания, значение истинности которых зависят не только от самого сформулированного высказывания, но также, например, от момента формулирования этого высказывания. Займемся теперь изучением действия модальных операторов на предикатные формы, значения истинности которых суть функции от значений, принимаемых их составляющими. Если составляющие – индивидные переменные, то они обычно будут находиться в области действия квантора общности или существования. Переменная x принимает значения из вполне определенной области, и квантификации $\forall x$ и $\exists x$ означают соответственно: «для любого значения из данной области» и «хотя бы для одного значения из данной области». Первоочередная задача модальной логики предикатов состоит в том, чтобы четко очертить упомянутую область. При этом универсум $\mathbb{W}$ возможных миров вводит дополнительную степень свободы, которую надо будет учесть при определении области значений переменных, входящих в составляющие части предикатов. Из наиболее очевидных определений такой области можно назвать следующие.

1. Множество Γ_r всех элементов, существующих в реальном мире.
2. Множество Γ_s всех элементов, существующих в некотором мире *s* универсума $\mathbb{W}$.
3. Множество Γ всех элементов, существующих в произвольном мире (т. е. $\Gamma = \bigcup_{s \in W} \Gamma_s$).

Кванторы общности и существования, отвечающие этим областям, определяются следующим образом:

1. $\forall_r x$ означает «для всех *x* в реальном мире»;
   $\exists_r x$ означает «хотя бы для одного *x* в реальном мире»;
2. $\forall_s x$ означает «для всех *x* в мире *s*»;
   $\exists_s x$ означает «хотя бы для одного *x* в реальном мире *s*»;

3. $\forall^* x$ означает «для всех x»;

$\exists^* x$ означает «хотя бы для одного x».

Например, если в различных возможных мирах задать временную интерпретацию, то формулу

$$\forall x(\textit{Человек}(x) \supset \textit{Смертен}(x))$$

можно интерпретировать тремя следующими различными способами – в соответствии с тем, какое определение дано квантору $\forall$:

1. Все ныне живущие люди смертны.
2. Все люди, жившие в определенную эпоху, смертны.
3. Все люди смертны.

Третья интерпретация может принять более ограничительное решение при переходе к временной логике прошлого или будущего:

3’. Все люди, жившие до настоящего момента, смертны.

3’’. Все люди, которые будут жить после настоящего момента, смертны.

Существует особая форма модальных операторов, которая позволяет легко включить их в формализм логики предикатов. Введем выражение $\mathbf{И}_s(p)$, означающее: «высказывание p истинно в мире s»; оно позволяет определить модальные операторы $\Box$ и $\Diamond$ следующим образом:

$$\Box p = \forall s\, \mathbf{И}_s(p),$$

$$\Diamond p = \exists s\, \mathbf{И}_s(p).$$

Эти определения применимы непосредственно в контексте логики предикатов. Символ p, который ранее обозначал высказывание, здесь может обозначать предикатную форму. (Здесь мы уже вышли за пределы логики первого порядка). В *модальной логике предикатов* имеется возможность построения восьми формул (приводится квантифицированная версия модальных операторов):

1. $\Box\forall x\, \mathrm{A}(x) = \forall s\, \mathbf{И}_s\, (\forall x\, \mathrm{A}(x))$,
2. $\forall x\Box\, \mathrm{A}(x) = \forall x \forall s\, \mathbf{И}_s\, (\mathrm{A}(x))$,
3. $\exists x\Box\, \mathrm{A}(x) = \exists x \forall s\, \mathbf{И}_s\, (\mathrm{A}(x))$,
4. $\Box\exists x\, \mathrm{A}(x) = \forall s\, \mathbf{И}_s\, (\exists x\, \mathrm{A}(x))$,
5. $\Diamond\forall x\, \mathrm{A}(x) = \exists s\, \mathbf{И}_s\, (\forall x\, \mathrm{A}(x))$,
6. $\forall x\Diamond\, \mathrm{A}(x) = \forall x \exists s\, \mathbf{И}_s\, (\mathrm{A}(x))$,
7. $\Diamond\exists x\, \mathrm{A}(x) = \exists s\, \mathbf{И}_s\, (\exists x\, \mathrm{A}(x))$,
8. $\exists x\, \Diamond\mathrm{A}(x) = \exists x\, \exists s\, \mathbf{И}_s\, (\exists x\, \mathrm{A}(x))$.

Пусть выполняются следующие эквивалентности:

$$\mathbf{И}_s\, (\forall x\, \mathrm{A}(x)) = \forall x\, \mathbf{И}_s\, (\mathrm{A}(x)), \qquad (1.2)$$

$$\mathbf{И}_s\, (\exists x\, \mathrm{A}(x)) = \exists x\, \mathbf{И}_s\, (\mathrm{A}(x)). \qquad (1.3)$$

С помощью этих эквивалентностей из формул 1–8 выводятся соотношения:

$$\Box\forall x\ A(x) = \forall x\ \Box A(x),$$
$$\Diamond\forall x\ A(x) \supset\ \forall x\ \Diamond A(x),$$
$$\Diamond\exists x\ A(x) = \exists x\ \Diamond A(x),$$
$$\exists x\Box\ A(x) \supset\ \Box\exists x\ A(x).$$

Однако эквивалентности (1.2) и (1.3) выполняются не всегда; все зависит от той или иной интерпретации, которая выбрана для кванторов $\forall$ и $\exists$. Например, можно принять интерпретацию $\forall x$, т. е. «для всех x в реальном мире». Беря некоторый конкретный предикат $A(x)$, получаем такую последовательность интерпретаций:

- $A(x)$: если x – человек, то x смертен.
- $\forall_{г} x\ A(x)$: все ныне живущие люди смертны.
- $\mathbf{И}_s(\forall_{г}\ x\ A(x))$: «все ныне живущие люди смертны» истинно в момент времени s.
- $\mathbf{И}_s(A(x))$: «если x – человек, то x смертнен» истинно в момент времени s.
- $\forall_{г} x(\mathbf{И}_s(A(x)))$: для всех ныне живущих людей истинно, что человек смертен в момент времени s.

Формула $\mathbf{И}_s(\forall_{г}\ x\ A(x))$ утверждает, что «в момент времени s все люди смертны»; формула $\forall_{г} x(\mathbf{И}_s(A(x)))$ гласит, что «все живущие люди смертны в момент времени s». Значит, мы имеем два различных по смыслу утверждения; следовательно, эквивалентности (1.2) и (1.3) для интерпретаций $\forall_{г}\ \exists_{г}$ не верны [6, стр. 66]. Приведенное замечание об общезначимости соотношений (1.2) и (1.3) порождает вопрос об условии общезначимости *формул Баркан*

$$\Diamond\exists x\ A(x) \supset\ \exists x\ \Diamond A(x),\ \forall x\Box\ A(x) \supset\ \Box\forall x\ A(x) \qquad (1.4)$$

входящих в ряд аксиоматических систем модальной логики предикатов. Пусть $A(x)$ значит, что «кто-то пришел». В первом случае формула $\Diamond\exists x\ A(x)$ означает: «возможно, что кто-то пришел», причем существование этого «кого-то» и его приход под вопросом; в то же время формула $\exists x\ \Diamond A(x)$ означает: «существует кто-то, кто, может быть, придет», следовательно, здесь существование этого «кого-то» гарантируется и проблематичен лишь его приход. Во втором случае формула $\forall x\Box\ A(x)$ значит «все обязательно придут», а $\Box\forall x\ A(x)$ «обязательно все придут».

Синтаксис

- Все синтаксические правила обычной логики предикатов являются также синтаксическими правилами модальной логики предикатов.
- Если A–формула, то $\Box A$ и $\Diamond A$ – формулы.

Аксиоматическая система

- Схемы аксиом обычной логики предикатов.
- Схемы модальных аксиом, иногда включающие схемы Баркан.
- Правила вывода логики предикатов.
- Модальное правило вывода необходимости, к которому иногда добавляются специальные правила для формализации некоторых видов рассуждений. Например, немонотонное правило вывода «*нельзя вывести* $\neg p$» $|\!-\Diamond p$ присоединяют для моделирования «веры» [7, §4.5.3].

Модель и семантика модальной логики предикатов

Моделью *M* модальной логики предикатов называется четверка ($\mathbb{W}$, R, S, V), составленная следующим образом.

- $\mathbb{W}$ – непустое множество возможных миров.
- R – бинарное отношение достижимости на множестве $\mathbb{W}$, т. е. подмножество декартова произведения $\mathbb{W}\times\mathbb{W}$.
- S – непустое множество элементов.
- V – функция оценки, которая
  - каждой паре, состоящей из элемента *w* из $\mathbb{W}$ и *n*-местной функциональной константы *f*, сопоставляет некую функцию из S^n в S.
  - каждой паре, состоящей из элемента *w* из $\mathbb{W}$ и *n*-местной предикатной константы *P*, сопоставляет некоторую функцию из S^n в {1, 0}.

Для любого мира *w* функция V каждой функциональной константе *f* сопоставляет то, что называется *расширением в w*. Точно таким же образом она действует на каждую предикатную константу *P*. Некоторые предикаты могут принимать различные значения истинности в разных возможных мирах универсума $\mathbb{W}$. В модели самого общего вида можно допустить, что значения индивидных констант могут различаться в разных мирах. Такие модели рассмотрены в [6, стр. 95–107]. Интерпретация формул модального языка отличается от формул языка первого порядка той ролью, которую играет универсум возможных миров $\mathbb{W}$. Интерпретация всегда осуществляется относительно некой функции *g*, которая присваивает индивидным переменным элементы из S.

Семантика формул *L* описывается с помощью символики $M \vDash_{\mathrm{W},\,g} \mathrm{A}$, которая означает «A истинна в модели *M* в мире *w* при функции присваивания *g*». Если α – некоторое выражение из *L*, то семантическое значение для α в модели *M* в мире *w* при функции присваивания *g* будет также обозначаться через $\ulcorner \alpha \lrcorner^{M,w,g}$. Итак, имеем следующую эквивалентность:

$$M \vDash_{\mathrm{W},\,g} \mathrm{A} \Leftrightarrow (\ulcorner \mathrm{A} \lrcorner^{M,w,g} = 1)$$

для всех формул A из L. Если c – какая–то константа (предикатная или функциональная), то справедливо соотношение

$$\ulcorner c \urcorner^{M,w,g} = \mathrm{V}(w,c).$$

Интерпретация формул модального языка осуществляется в соответствии с семантическими правилами языка логики предикатов, в котором все вхождения g заменяются на w, g. Добавим правила относительно модальных операторов $\Box$ и $\Diamond$.

Семантика

- $M \models_{w,g} \Box A$ тогда и только тогда, когда $M \models_{w',g} A$ для всех таких $w' \in \mathbb{W}$, что $(w, w') \in \mathrm{R}$.
- $M \models_{w,g} \Diamond A$ тогда и только тогда, когда $M \models_{w',g} A$ хотя бы для одного такого $w' \in \mathbb{W}$, что $(w, w') \in \mathrm{R}$.

Формула A *общезначима* тогда и только тогда, когда $M \models_{w,g} A$ для любой модели M, любого значения g и любого мира w.

Динамическая логика первого порядка

Здесь вводится некоторый язык, получающийся от слияния динамической логики и некоторого языка первого порядка. В языке динамической логики первого порядка атомарные программы Π заменяются на *команды присваивания*, записываемые в виде $[x := \sigma]$; здесь x – индивидная переменная, а σ – терм; эта команда читается: «переменной x присвоить текущее значение σ». Существует тесная связь между вычислительным процессом присваивания значения переменной и синтаксическим процессом подстановки переменной. Если через $A^{x:=\sigma}$ обозначить результат замещения всех свободных вхождений переменной x ее значением σ в формуле первого порядка A, то формула $[x := \sigma]A \equiv A^{x:=\sigma}$ общезначима. Значит, модальные формулы вида $[x := \sigma]A$ можно использовать во всех тех случаях, когда в логике первого порядка применяется выражение $A^{x:=\sigma}$. В этом формализме формулы неклаузных резолюций [7, §1.1.15] записываются так:

$$\mathrm{T}_x(A_1, A_2) = _{def} [x := 0]\, A_1 \wedge [x := 1]\, A_2,$$
$$\bot_x(A_1, A_2) = _{def} [x := 0]\, A_1 \vee [x := 1]\, A_2.$$

Например, этот формализм позволяет избежать обязательной разработки некоторой теории синтаксических подстановок в модальной логике.

В контексте вычисления (или программы) можно определить понятие *состояния* как множество значений совокупности всех переменных.

Следовательно, можно отождествить состояние с *означиванием* всех индивидных переменных. В таком контексте программу можно интерпретировать как бинарное отношение между означиваниями. Для этого нужно определить отношение эквивалентности $s \sim_x t$ следующим образом: оно означает, что состояния s и t отличаются только тем значением, которое присвоено переменной x. Тогда семантические определения, касающиеся кванторов $\exists$ и $\forall$, можно записать в виде:

$\models_s \exists x A$ Тогда и только тогда, когда существует такое состояние t, что $\models_t A$ и $s \sim_x t$.

$\models_s \forall x A$ Тогда и только тогда, когда имеет место $\models_t A$ для всех состояний t, таких, что и $s \sim_x t$.

Отсюда видно, что кванторы логики первого порядка ведут себя как модальные операторы.

С формальной точки зрения выражения $\exists x$ и $\forall x$ ведут себя как операторы $\Diamond$ и $\Box$ в модальной логике $S5$. Приведем список аксиом, применяемых в аксиоматизации некой динамической логике первого порядка.

1. $\forall x(A \supset B) \supset (\forall x\, A \supset \forall x\, B)$,
2. $A \supset \forall x A$ (x не входит в A),
3. $\forall x A \supset [x := \sigma]A$,
4. $\forall y\, [x := y]A \supset \forall x A$ (x не входит в A и $x \neq y$),
5. $\forall x A \supset [x := \sigma] \forall x A$,
6. $\forall y [x := \sigma]A \supset [x := \sigma] \forall y A$,
7. $<x := \sigma>A \equiv [x := \sigma]A$,
8. $[x := \sigma]A \equiv A^{x:=\sigma}$,
9. $[x := \sigma][y := \tau]A \supset [y := \tau^{x := \sigma}]A$,
10. $[x := \sigma][y := \tau]A \supset [y := \tau^{x := \sigma}][x := \sigma]A$,
11. $\sigma = \tau \supset ([x := \sigma]A \equiv [x := \tau]A)$.

Польза модальных языков, основанных на динамической логике, главным образом обусловлена их большой выразительной силой. По существу эти языки дают подходящие рамки, в которых можно выразить большинство формальных свойств, относящихся к языкам программирования. Таковыми являются свойства частичной корректности или завершаемости программ.

ГЛАВА 2. СТРУКТУРНЫЕ ИМПЛИЦИТНЫЕ МОДЕЛИ

В данной главе мы попытаемся построить формализм, призванный обеспечить возможность создания строгих и, в то же время адекватных моделей для широкого круга предметных областей, обладающих свойством конструктивной (программной) реализуемости. Для достижения этой цели мы будем отталкиваться от хорошо разработанной теории вычислительных моделей [8] и исполь зовать ряд современных подходов и достижений.

2.1. Формальная теория

Дадим основные определения. Прежде всего, зафиксируем сигнатуру **Σ,** в рамках которой будем производить дальнейшие рассуждения.

Определение 2.1. Четверку $\Sigma = (A,F,P,D)$, где ***A, F, P*** и ***D*** – не более чем счетные множества (элементарных) имен объектов, функциональных символов, селекторных символов и имен схем, соответственно, назовем сигнатурой. Считается, что выделено непустое конечное подмножество $E \subset D$ имен *примитивных* или *первичных* схем. Элементы множества ***A*** называются *именами объектов* или *объектами,* если это не приводит к двусмысленности. Связь объекта ***a*** со схемой ***S*** отражается записью ***S(a).*** Если объект ***a*** связан со схемой ***S*** и $S \in E$, стандартная запись ***S(a)*** заменяется записью a^S либо просто ***a***, когда ссылка на схему не важна или очевидна из контекста. – Истинностное значение синонимов a^S и ***S(a)*** будем определять следующим образом: $a^S = true$, тогда и только тогда, когда $a|_I \in S|_I$, где $a|_I$ и $S|_I$ обозначают реализации (на интерпретации **I**) величины и схемы соответственно**.** Подчеркнем еще раз, что экземпляры реализаций первичных схем несут с собой первичные свойства (операции и отношения соответствующих экспортирующих типов).

Определение 2.2. Выражение вида

$$f:a_1,\ldots,a_n \to a_0, \tag{2.1}$$

где a_i, $i=0,\ldots,n$ – имена объектов, называется функциональной связью (***ФС***). В записи (2.1) ***f*** – это имя ***ФС***, a_i – аргументы (i = 1,…,n), a_0 – результат ***ФС***[3]. Неформально ***ФС*** трактуется как возможность вычисления

[3] *Реализация **f** – вычислимая функция и, как правило, однородная, т. е. экспортированная некоторым одним первичным типом – реализацией первичной схемы. Допускаются, впрочем, и смешанные функции, тем не менее остающиеся вычислимыми. – Последнее важно для прикладных теорий со строгой типизацией (когда, например, различаются реализации понятий «время», «скорость» и т. д.).*

значения атрибута a_0 по набору значений атрибутов $a_1,\ldots,a_n$ применением реализации отображения f.

Имена объектов при моделировании конкретной ***ПО*** формируются из элементов множества A и удовлетворяют следующему рекурсивному определению:

Определение 2.3. Пусть $a \in A$, α – имя объекта, тогда a, $\alpha.a$ и $a.\alpha$ также являются именами объектов. В записи вида $\alpha.a$ α называется префиксом. *Длина* произвольного имени α определяется числом вхождений элементов множества A (с учетом возможных повторений), из которых сформировано имя.

Одним из базовых является понятие (непервичной) схемы:
Определение 2.4. Схема S объекта r определяется записью вида

$$S(r) = r.(S_{01}(a_{01}),\ldots, S_{0N0}(a_{0N0}),$$
$$\textbf{if} \quad p_1 \supset S_{11}(a_{11}),\ldots,S_{1N1}(a_{1N1}) \square\ldots\square \qquad (2.2)$$
$$p_k \supset S_{k1}(a_{k1}),\ldots,S_{kNk}(a_{kNk}) \; \textbf{fi} \; / \textbf{filter}),$$

где $S \in D \backslash E$, $r \in A$. Для всех возможных значений индексов i, j, $S_{ij} \in D$ – *собственные подсхемы* схемы S, $a_{ij} \in A$ – ее *собственные величины*, $p_i \in P$ – (возможно, параметризованные) выбирающие *селекторы*. В правой части (2.2) r называется *префиксом* схемы, участок до вертикальной черты – *заголовком*, фрагмент ***if…fi*** – *вариантной* его *частью*, а остальная часть заголовка – *постоянной частью*. Символ «□» заимствован из нотации «охраняемых команд» Дейкстры [9] и соответствует традиционному «***else_if***». Иероглиф ***filter*** скрывает список *собственных* ***ФС*** схемы S. Считается, что префикс r может «проноситься» в скобочный фрагмент, так что $r.(S(a),\ldots)=_{def}(r.S(a),\ldots)=_{def}(S(r.a),\ldots)$. Аналогичным образом префиксируются имена селекторов и отображений, а также имена величин, вовлеченных в формирование ***ФС*** из ***filter***[4].

Селекторы на интерпретации **I** получают истинностные (шкальные) значения и образуют полную систему, т. е. $\forall i,j,\ i \neq j,\ p_i|_{\mathbf{I}} \& p_j|_{\mathbf{I}} = \textit{false}$ и $p_1|_{\mathbf{I}} \vee \ldots \vee p_k|_{\mathbf{I}} = \textit{true}$. Синтаксическая конструкция (2.2) на интерпретации **I** определяет размеченное отношение в смысле [9] [5].

---

[4] ***(NB)*** *Фиксация множества $E \subset D$ для конкретной* ***ПО*** *не является статической. При необходимости можно сделать «доопределение вниз», представив ранее первичный элемент в форме (2.2), и детализировать рассматриваемую* ***ПО*** *соответствующей модификацией множества E. Требование конечности множества E при этом не нарушается.*

[5] *Определение 2.4, на первый взгляд, предоставляется не более чем механизмом введения сокращения при описании* ***ПО****, но смысл использования данного понятия явля-*

Можно отметить некоторую аналогию между введенным здесь понятием схемы и отношением в теории реляционных баз данных [10]. Используя это, мы иногда будем употреблять термины реляционного подхода, такие как «атрибут», «подсхема», «кортеж» и т. д. Укажем теперь требования, которым удовлетворяет набор ***filter*** схемы. Для этого, прежде всего, определим, какие имена величин порождаются заголовком схемы.

Определение 2.5. Пусть $S(r)=(\ldots S_{ij}(r.a_{ij})\ldots)$ – схема. Если $S_{ij}\in E$, то $r.a_{ij}$ – имя величины схемы S. Если $S_{ij}\in D\backslash E$ и α – имя величины схемы S_{ij}, то $r.a_{ij}.\alpha$ – имя величины схемы S. В дальнейшем имена величин схемы S будем называть её атрибутами. В качестве имен атрибутов будем использовать строчные буквы начала латинского алфавита либо буквы греческого алфавита – если надо подчеркнуть сложную природу атрибута.

Определение 2.6. Рассмотрим схему S. $ФС\ f:\alpha_1,\ldots,\alpha_n\rightarrow\alpha_0$ из ***filter*** называется допустимой для схемы S, если и только если $\alpha_0,\alpha_1,\ldots,\alpha_n$ – атрибуты схемы S. Схема называется синтаксически правильной, если ***filter*** содержит только допустимые $ФС$.

В дальнейшем будем использовать только синтаксически правильные схемы. Помимо этого будем считать, что $ФС$, содержащиеся в наборе ***filter*** схемы S, отвечают следующим требованиям:

- $ФС$ содержит по крайней мере один собственный атрибут схемы;
- в наборе ***filter*** нет $ФС$, в которые вовлечены атрибуты из разных альтернативных ветвей схемы S или атрибуты из альтернативных частей её подсхемы;
- длина любого атрибута, связанного $ФС$, не превышает двух (то есть мы не опускаемся более чем на один уровень).

Иногда величины предметной области, для которой строится модель, имеют смысл только при выполнении некоторого критерия. Так, о гипотенузе и катетах можно говорить лишь в контексте прямоугольного треугольника. – Для отражения подобных фактов служит альтернативная часть заголовка схемы – она указывает, когда определены те или иные атрибуты. В ряде случаев информацию об осмысленности атрибутов схемы бывает важно иметь в наборе ***filter***. Для этого вводится понятие атрибута с условием или *y*-атрибута:

Определение 2.7. Факт наличия условия p у атрибута α будем отмечать записью α/p, а соответствующее выражение называть *y-атрибутом*.

Будем считать, что с атрибутами, определёнными в общей части заголовка, связывается условие ***true***. Если в записи *y*-атрибута α/p $p=_{def}$***true***, то p можно опустить.

ется существенно более глубоким. Ниже, следуя [4]*, мы попытаемся обосновать введение данного расширения понятия атрибута.*

В качестве примера схемы приведём описание понятия ***член-ряда***, определяющего *n*-й член натурального ряда либо ряда Фибоначчи в зависимости от значения скаляра *s*. Символом «***N***» будем обозначать тождественное отображение:

член_ряда(r) = r.(scalar(s), integer(x), integer(n),

if (= s "натуральный") ⊃ integer(xn) □

(= s "фибоначчи") ⊃ число_фиб(xf) fi |

N : n → xn ; N : xn → x ; N : n → xf.n ; N : xf.f → x).

В этом примере неэлементарной является схема ***число_фиб***, которая определяет член ряда Фибоначчи. Описание этой схемы будет приведено позднее, когда мы коснёмся рекурсивных конструкций. Собственными атрибутами схемы ***член_ряда*** являются ***s, x, n, xn*** и ***xf***. В схеме ***число_фиб***, связанной с атрибутом ***xf***, определены атрибуты ***n*** и ***f***, таким образом в схеме ***член_ряда*** появляются префиксированные атрибуты ***xf.n*** и ***xf.f***. Для натурального ряда значение *n*-го члена совпадает с его номером *n* – этот факт отражается двумя первыми ***ФС*** схемы. Число Фибоначчи вычисляется по более сложным законам, определяемым схемой ***число_фиб***. Вход в подсхему ***число_фиб*** схемы ***член_ряда*** и выход из неё осуществляется посредством двух последних ***ФС*** набора ***filter*** схемы ***член_ряда.***

Здесь мы явно используем язык типизированных логик высших порядков. Следует отметить, что применяемые в работе элементы конструктивизма и некоторые другие ограничения оставляют теорию разрешимой с хорошими алгоритмическими оценками.

Определим понятие структурной вычислительной модели.

Определение 2.8. Структурной (***C-***) моделью называется конечная совокупность (описаний) схем $M=(T_1,\ldots,T_m)$, где каждая T_i, $i=1,\ldots,n$ является элементарной или задана в соответствие с определением 2.4[6].

Определение 2.9. С-модель ***M*** является (синтаксически) замкнутой, если и только если для каждого её элемента $T_i \in M$ схемы, встречающиеся в определении T_i, являются элементарными либо определены в описании С-модели ***M***.

---

[6] *Можно заметить сходство определяемого здесь понятия модели и библиотеки классов в так называемом «объектно-ориентированном» подходе к программированию. Ниже будет показано, как с использованием стандартных логических формализмов могут быть проинтерпретированы такие «столпы» ООП, как инкапсуляция, полиморфизм и наследование.*

Описывая С-модель для предметной области, мы строим специальную (формальную) теорию, в которой можно решать различные алгоритмические проблемы. Здесь мы, главным образом, рассматриваем проблему синтеза (вывода) реализующих алгоритмов (программ) и сложностные характеристики синтеза.

Постановка задачи в С-модели носит непроцедурный характер – в ней указываются лишь исходные и искомые атрибуты некоторой схемы. Клиентом (субъектом) модели может выступать не только пользователь-человек, но и любой функциональный агент, авторизованный в модели (последним может быть объявленная в модели схема или внешний потребитель). Предполагается, что субъекты модели функционально согласованы с требованиями, заданными вышеприведенными определениями.

Содержательно постановка задачи на модели есть задание на построение схемы алгоритма, реализующего требуемые вычисления. Схема алгоритма извлекается из (конструктивного) доказательства соответствующей теоремы существования. Схема становится алгоритмом (программой) в результате задания интерпретации для С-модели.

Определение 2.10. Задачей в С-модели $\boldsymbol{M}$ называется тройка $\boldsymbol{S=(A_0,X_0,T)}$, где $\boldsymbol{A_0}$ и $\boldsymbol{X_0}$ – наборы имён, соответственно, исходных и искомых величин, а $\boldsymbol{T}$ – схема С-модели $\boldsymbol{M}$, в которой определены эти имена[7].

При исследовании свойств С-моделей и алгоритмов синтеза программ в них нам потребуется понятие развёртки схемы $\boldsymbol{T}$. Оно определяется через развёртку схемы на атрибуте.

Определение 2.11. Пусть $\boldsymbol{T(r)=r.(\ldots T_{ij}(a_{ij}),\ldots|\ filter)}$ – схема, и $\boldsymbol{T_{ij}\notin E}$. Развёрткой схемы $\boldsymbol{T}$ на атрибуте $\boldsymbol{a_{ij}}$ будем называть выражение, получающееся в результате

а) подстановки в заголовок исходной схемы на место $\boldsymbol{T_{ij}(a_{ij})}$ заголовка схемы $\boldsymbol{T_{ij}}$ и

б) присоединения к набору ***filter*** схемы $\boldsymbol{T}$ всех ***ФС*** схемы $\boldsymbol{T_{ij}}$.

Все добавленные в схему $\boldsymbol{T}$ в результате такого действия атрибуты, имена селекторов и отображений модифицируются префиксом $\boldsymbol{r.a_{ij}}$.

Процесс построения развёртки на атрибуте может повторяться многократно – пока имеются атрибуты, связанные с неэлементарными схемами.

Определение 2.12. Под полной развёрткой $\boldsymbol{T}$ С-модели понимается объект, полученный из схемы $\boldsymbol{T}$, у которой в результате последовательно-

[7] *В определении задачи не все имена атрибутов из наборов $\boldsymbol{A_0}$ и $\boldsymbol{X_0}$ обязаны явно присутствовать в заголовке схемы $\boldsymbol{T}$ – это могут быть и произвольные атрибуты схемы.*

сти развёрток на атрибутах в заголовке остаются только атрибуты, связанные с элементарными схемами.

Полная развёртка, вообще говоря, не является схемой в смысле определения 2.4, однако понятие схемы нетрудно переопределить с тем, чтобы оно оставалось корректным для развёртки на атрибуте и полной развёртки. Для этого необходимо в выражении (2.2) разрешить появление нескольких ***if…fi*** – участков и потребовать, чтобы единственным видом их пересечения было строгое вложение (как оно трактуется в структурных языках программирования).

При построении С-модели допускается возможность рекурсивных определений схем. В общем случае рекурсивная конструкция должна содержать, по меньшей мере, одну цепочку определений вида T_1=(…[T_k]…), …, T_k=(…[T_1]…), обеспечивающую участие в дефиниции некоторого понятия ссылки на себя. Схемы T_i, i=1,…,k будем называть рекурсивными. Ясно, что полная развёртка рекурсивной схемы не определена – процесс её построения не завершается. Ниже будет показано, что некоторые естественные ограничения позволяют выделить класс С-моделей, допускающий рекурсивные схемы, в котором проблема синтеза решается с полиномиальными оценками, несмотря на трудности, связанные с анализом рекурсивных определений схем.

В качестве примера опишем схему, определяющую ***n***-й элемент ряда Фибоначчи:

число_фиб(r) = r.(integer(n), integer(f),
if (= n "0") ⊃ integer(ff) □
(= n "1") ⊃ integer(sf) □
(> n "1") ⊃ число_фиб(nf), число_фиб(nnf) fi |
N:"1"→ff; N:sf→f; N:"1"→ff; N:ff→f;
subtract:n,"1" →nf.n; subtract:n,"2" →nnf.n; sum:nf.f, nnf.f →f).

2.2. Нерекурсивные детерминированные С-модели

В этом параграфе мы выделим некоторый класс С-моделей. Для этого нам понадобится понятие детерминированной схемы. Чтобы подчеркнуть факт детерминируемости (отвлекаясь пока от вопроса задания интерпретации С-модели), мы будем использовать двухальтернативную форму записи участка ветвления: ***if*** $P \supset S_1;S_2$ ***fi*** – подразумевая, что реализация S_1 осмыслена при истинности реализации P, а S_2 – в противном случае.

Определение 2.13. Схему T вида (1.2) будем называть детерминированной, если она не имеет альтернативной части либо её альтернативная часть имеет форму ***if*** $P \supset$ …;… ***fi*** .

Здесь и ниже, если это не оговаривается специально, будем полагать, что совокупность схем, составляющих С-модель, является упорядоченной.

Определение 2.14. С-модель $M = (T_1, \ldots, T_s)$ называется не рекурсивной детерминированной (НДС-) моделью, если выполнены следующие условия:

1. M – синтаксически замкнута;
2. $T_1, \ldots, T_s$ – детерминированные схемы;
3. Любая схема T_i модели определяется только через элементарные схемы либо через схемы, которые предшествуют T_i в модели M.

В рамках введенных соглашений можно использовать некоторый базовый набор правил вывода (допускающий модификации):

0) ***схема аксиом*:** $\vdash N{:}A \to A,$

***правила*:**

1) $\dfrac{\vdash F{:}A \to X,Z}{\vdash F{:}A \to X}$ (*правило сужения*),

2) $\dfrac{\vdash F{:}A \to X,Z/P \quad \vdash f{:}Z \to x/p}{\vdash F;f{:}\ A,Z/P \to X,Z,x/P\,\&\,p}$ (*правило суперпозиции*),

здесь P – конъюнкция (шкальных) условий достижимости атрибутов из Z, а p – условие допустимости x (условие достижимости – это условие, при котором в процессе построения доказательства обеспечивается достижимость некоторого атрибута, а условие допустимости – это условие, при котором атрибут имеет смысл в описании).

3) $\dfrac{\vdash F_1{:}A \to X,x/Q\&p \quad \vdash F_2{:}A \to X,x/Q\&\text{-}p}{\vdash \textit{if } p \textit{ then } F_1 \textit{ else } F_2 \textit{ fi}{:}A \to X,x/Q}$ (*правило ветвления*),

4) $\dfrac{\vdash F_1{:}A \to X,x/Q\&p \quad g_1{:}n_{\alpha_1}^{k_1}(A) \to n_{\alpha_1}^{k_1}(X),\ldots,g_s{:}n_{\alpha_s}^{k_s}(A) \to n_{\alpha_s}^{k_s}(X) \vdash F_2{:}A \to X,x/Q\&\neg p}{h = \underline{\textit{if}}\ p\ \underline{\textit{then}}\ F_1\ \underline{\textit{else}}\ F_2[h/g_1]\ldots[h/g_s]\ \underline{\textit{fi}}{:}A \to X/Q}$ (*правило рекурсии*).

ГЛАВА 3. ВВЕДЕНИЕ В VISUAL PROLOG

Пролог является результатом многолетней исследовательской работы. Первая официальная версия Пролога была разработана Аланом Кольмероэ (Alain Colmerauer) в Марсельском университете во Франции в начале 1970-х годов как инструмент для ПРОграммирования в ЛОГике. В результате своего развития появился язык выразительно более мощный, чем даже такие широко распространенные сегодня языки программирования, как Pascal и C (с диалектами) [11].

Пролог известен как декларативный язык. Это означает, что при заданных необходимых фактах и правилах, Пролог будет использовать дедуктивные умозаключения для решения задач программирования. Эта его отличительная особенность выгодно контрастирует с традиционными декларативными языками. В декларативном (процедурном) языке следует задавать компьютеру пошаговый алгоритм решения конкретной задачи, иными словами, программист должен заранее знать, как решить данную задачу. Пролог-программисту нужно предоставить только описание задачи и основные правила для ее решения. Таким образом, система Пролог предназначена, в том числе и для определения того, как найти необходимое решение.

Пролог имеет ряд преимуществ по сравнению с процедурными языками программирования, вот некоторые из них:

- для определенных задач программа на Прологе требует только одну десятую часть строк кода по сравнению с аналогичной программой на языке C++;
- благодаря декларативному (в большей степени, чем процедурному) подходу, такие хорошо известные источники ошибок, как зацикливания, устраняются с самого начала;
- Пролог «заставляет» программиста начинать с хорошо структурированного описания задачи, поэтому он может использоваться и как средство создания спецификации, и как средство реализации продукта.

Где может использоваться Visual Prolog?

Пролог является очень важным инструментом в программировании приложений искусственного интеллекта и в разработке экспертных систем.

Высокий уровень абстракции, легкость и простота в представлении сложных структур данных, возможность моделировать логические отношения между объектами и процессами существенно облегчают решение задач в различных предметных областях. По этой причине Visual Prolog широко используется для создания административных приложений, систем календарного планирования, Web-приложений, для управления большими и сложными базами данных.

Мир искусственного интеллекта

Пролог берет свое начало в исследованиях по искусственному интеллекту (ИИ) и изначально разрабатывался как язык ИИ, он прекрасно подходит для разработки экспертных систем и других соответствующих приложений. Системы, основанные на фреймах или правилах, прямой или обратный логический вывод, системы сопоставления с образцом и системы вывода с ограничивающими условиями – все это естественные и изящные выражения базовой семантики Пролога. Используя Пролог, вы можете эффективно программировать такие приложения, как базы знаний, экспертные системы, естественно-языковые интерфейсы и интеллектуальные системы управления информацией.

Универсальная среда разработки

Далее будет говориться о версии Visual Prolog компании PDC – это конкурентоспособная, универсальная среда разработки. Visual Prolog все чаще становится предметом выбора разработчиков из-за возможностей работы с развитой логикой и в то же время может выполнять те же задачи, что и системы баз данных SQL, системы разработки программ C++, другие инструментальные средства, такие как Visual Basic, Borland Delphi или IBM Visual Age. Оптимизированный компилятор Visual Prolog обеспечивает высокую скорость работы приложений.

Средства разработки Web-программ – новый, очень важный элемент Visual Prolog. Например, экспертные системы, написанные в Visual Prolog, могут быть подключены к Web-страницам, что играет важную роль в отделах поддержки, системах торговли в Internet и некоторых других Web-технологиях. Visual Prolog поддерживает объектно-ориентированный стиль программирования – мощный инструмент моделирования, который является почти стандартом де-факто в таких языках, как Object Pascal, C++, Smalltalk и т. д.

Все эти особенности делают Visual Prolog выгодной с коммерческой точки зрения инструментальной средой разработки программ.

3.1. Установка и начало работы в Visual Prolog

Этот раздел описывает процесс установки и запуск Visual Prolog на компьютере. Рассматриваются некоторые функции интегрированной среды визуальной разработки (VDE, Visual Development Environment) Visual Prolog, позволяющие выполнять представленные в книге примеры.

Обратите внимание, что Visual Prolog не устанавливает никаких DLL, не изменяет существовавшие ранее INI-файлы и не изменяет системный реестр (если вы выбираете опцию: не создавать группу Visual Prolog и не связывать PRJ- и VPR-файлы). Фактически, вы можете удалить Visual Prolog с вашего компьютера, удалив лишь каталог, где он установлен.

Программа инсталляции не создает управляющий файл инициализации (vip32_52.ini в каталоге WINDOWS) для интегрированной среды разработки Visual Prolog. Он создается автоматически при первом запуске интегрированной среды разработки.

После того как инсталляция закончена, отображается файл, содержащий последнюю информацию. Мы рекомендуем внимательно его прочитать, прежде чем начинать работать с Visual Prolog.

Итак, Visual Prolog установлен, и может быть запущен двойным щелчком мыши на пиктограмме VIP.

Рекомендации по установке Visual Prolog

Для того чтобы запускать и тестировать примеры, описанные в этой главе, во время инсталляции Visual Prolog необходимо выполнить следующие действия:

1. В диалоговом окне **Compilers** выберите установку **Visual Development Environment (VDE)** – Win32.
2. В диалоговом окне **Libraries** отметьте установку библиотек, относящихся к выбранной платформе для **VDE**.
3. В диалоговом окне **Documentation** выберите установки **Answers** и **Examples**.
4. В диалоговом окне **Final** рекомендуется выбрать флажок

Associate 32-bit VDE with Project File Extensions PRJ & VPR.

Запуск Visual Prolog с CD-ROM

Полноценная среда разработки Visual Prolog может быть запущена прямо с дистрибутивного диска. В этом случае никакой инсталляционный процесс не нужен, и VDE может быть вызван немедленно запуском файла *<CD>:\RUN\BIN\ WIN32\vip.exe*

Запуск Visual Prolog

Программа установки инсталлирует группу программ с пиктограммой, которая обычно используется для запуска среды визуальной разработки Visual Prolog. Впрочем, существует множество других вариантов запуска приложения в Windows, например, запуск исполняемого файла VIP.EXE из папки BIN\WIN\32, находящейся в основном каталоге Visual Prolog.

Если во время закрытия среды визуальной разработки был открыт проект (PRJ или VPR файл), то при следующем запуске этот проект откроется автоматически.

Если в процессе инсталляции Visual Prolog вы выбрали флаг ***Associate 32-bit VDE with Project File Extensions PRJ & VPR***, то для открытия проекта достаточно дважды щелкнуть на файле с расширением ***prj*** или ***vpr***. Среда визуальной разработки запускается и загружает выбранный проект.

Для запуска большинства примеров из данного руководства необходимо использовать Test Goal – утилиту среды визуальной разработки. Эта утилита может быть активизирована при помощи команды Project|Test Goal или комбинации клавиш <Ctrl>+<G>. Для корректного выполнения примеров с утилитой Test Goal среда визуальной разработки использует специальные настройки загружаемых проектов. Рекомендуется создать и всегда использовать специальный TestGoal-проект.

Создание TestGoal-проекта для выполнения примеров

При использовании утилиты Test Goal для выполнения примеров требуется определить некоторые (не предопределенные) опции компилятора Visual Prolog. Для этого выполните следующие действия:

1. Запустить среду визуальной разработки Visual Prolog. При первом запуске VDE (среда визуальной разработки) проект не будет загружен. Также вас проинформируют, что по умолчанию создан инициализационный файл для Visual Prolog VDE.
2. Создать новый проект.

   Выберите команду Project|New Project, активизируется диалоговое окно Application Expert.
3. Определите базовый каталог и имя проекта.

   Примеры из данного руководства следует устанавливать в подкаталог *\DOC \Examples* корневого каталога Visual Prolog. Допустим, что при установке Visual Prolog вы выбрали *C:\VIP* в качестве корневого каталога Visual Prolog. В этом случае мы рекомендуем задать имя в поле **Base Directory**: *C:\VIP\DOC\ExampIes\TestGoal*

   Это определение очень удобно для будущей загрузки исходных текстов примеров, представленных в данном руководстве.

   Имя в поле Project Name следует определить как «TestGoal».

   Также установите флажок ***Multiprogrammer Mode*** и щелкните мышью внутри поля ***Name of.PRJ File***. Вы увидите, что появится имя файла проекта TestGoal.prj.

Открытие окна редактора

Для создания нового окна редактирования вы можете использовать команду меню **File|New**. В результате появится новое окно редактирования с именем Noname.

Запуск и тестирование программы

Для проверки того, что ваша система настроена должным образом, следует напечатать следующий текст в окне:

GOAL write(«Hello world»),nl.

В терминологии языка Пролог это называется GOAL, и этого достаточно для программы, чтобы она могла быть выполнена. Для того

чтобы выполнить GOAL, вам следует активировать команду Project|Test Goal или нажать комбинацию клавиш <Ctrl>+<G>.

Результат выполнения программы будет расположен вверху в отдельном окне (на рисунке оно называется Inactive *C:\Vip\Doc\Examples\TestGoal\Obj\goalS000.exe)*, которое необходимо закрыть перед тем, как тестировать другую GOAL.

Тестирование примеров в Test Goal

Мы советуем вам попробовать сейчас открыть один из примеров в среде визуальной разработки и протестировать его, используя утилиту Test Goal. Для этого выполните следующие шаги:

Запустите среду визуальной разработки Visual Prolog.

1. Используйте команду меню **Project|Open** Project для открытия специального TestGoal-проекта.
2. Используйте команду меню **File|Open** для открытия одного из файлов chCCeNN.pro.
3. Используйте команду меню **Project|Test Goal** (или нажмите комбинацию клавиш <Ctrl>+<G>) для тестирования загруженного примера.

Test Goal найдет все возможные решения GOAL и покажет значения всех переменных, используемых в GOAL.

Обработка ошибок

Если вы допустили ошибки в программе и пытаетесь скомпилировать ее, то среда визуальной разработки отобразит окно Errors (Warnings), которое будет содержать список обнаруженных ошибок.

Дважды щелкнув на одной из этих ошибок, вы попадете на место ошибки в исходном тексте. Можно воспользоваться клавишей <F1> для вывода на экран интерактивной справочной системы Visual Prolog. Когда окно помощи откроется, щелкните по кнопке Search, наберите номер ошибки, и на экране появится соответствующее окно помощи с более полной информацией о ней.

3.2. Программирование в Логике

В Прологе (Prolog – PROgramming in LOGic) вы получаете решение задачи логическим выводом из ранее известных положений. Обычно программа на Прологе не является последовательностью действий – она представляет собой набор фактов с правилами, обеспечивающими получение заключений на основе этих фактов. Поэтому Пролог известен как декларативный язык.

Пролог базируется на предложениях Хорна, являющихся подмножеством формальной системы, называемой логикой предикатов. Логика предикатов – это простейший способ объяснить, как «работает» мышление, и она проще, чем арифметика, которой Вы давно пользуетесь.

Пролог использует упрощенную версию синтаксиса логики предикатов, он прост для понимания и очень близок к естественному языку.

Пролог включает механизм вывода, который основан на сопоставлении образцов. С помощью подбора ответов на запросы он извлекает хранящуюся (известную) информацию. Пролог пытается проверить истинность гипотезы (другими словами – ответить на вопрос), запрашивая для этого информацию, о которой уже известно, что она истинна. Прологовское знание о мире – это ограниченный набор фактов и правил, заданных в программе.

Одной из важнейших особенностей Пролога является то, что в дополнение к логическому поиску ответов на поставленные вами вопросы, он может иметь дело с альтернативами и находить все возможные решения. Вместо обычной работы от начала программы до ее конца, Пролог может возвращаться назад и просматривать более одного «пути» при решении всех составляющих задачу частей.

Таблица 3.1

Соответствие естественного языка и языка логики предикатов

Предложение на естественном языке	Предложение на языке логики предикатов
Машина красивая	nice(car)
Роза красная	red(rose)
Биллу нравятся красивые машины	likes(bill,car) if nice(car)

Логика предикатов была разработана для наиболее простого преобразования принципов логического мышления в записываемую форму. Пролог использует преимущества синтаксиса логики для разработки программного языка. В логике предикатов Вы, прежде всего, исключаете из своих предложений все несущественные слова. Затем эти предложения преобразуются путем перестановки на первое место отношения, а после него – сгруппированных объектов. В дальнейшем объекты становятся аргументами, между которыми устанавливается это отношение. В качестве примера в табл. 3.1 представлены предложения, преобразованные в соответствии с синтаксисом логики предикатов.

Факты и правила

Программист на Прологе описывает объекты (objects) и отношения (relations), а затем описывает правила (rules), при которых эти отношения являются истинными. Например, предложение «***Билл любит собак***» **(*Bill likes dogs*)** устанавливает отношение между объектами ***Bill*** и ***dogs*** (Билл и собаки); этим отношением является ***likes*** (любит). Ниже представлено правило, определяющее, когда предложение «*Билл любит*

собак» является истинным: «***Билл любит собак, если собаки хорошие***» **(*Bill likes dogs if the dogs are nice*)**.

Факты

В Прологе отношение между объектами называется фактом (***fact***). В естественном языке отношение устанавливается в предложении. В логике предикатов, используемой Прологом, отношение соответствует простой фразе (факту), состоящей из имени отношения и объекта или объектов, заключенных в круглые скобки. Как и предложение, факт завершается точкой.

Ниже представлено несколько предложений на естественном языке с отношением «*любит*» (*likes*):

Билл любит Синди. (Bill likes Cindy) ***Синди любит Билла.*** (Cindy likes Bill) ***Билл любит собак.*** (Bill likes dogs).

А теперь перепишем эти же факты, используя синтаксис Пролога:

likes(bill, cindy). *likes(cindy, bill)*, *likes(bill, dogs)*.

Факты, помимо отношений, могут выражать и свойства. Так, например, предложения естественного языка «*Kermit is green*» (*Кермит зеленый*) и «*Caitlin is girl*» (*Кейтлин – девочка*) на Прологе, выражая те же свойства, выглядят следующим образом:

green(kermit), girl(caitlin).

Правила

Правила позволяют вам вывести один факт из других фактов. Другими словами, можно сказать, что правилю – это заключение, для которого известно, что оно истинно, если одно или несколько других найденных заключений или фактов являются истинными. Ниже представлены правила, соответствующие связи «любить» (likes):***Синди любит все, что любит Билл.*** (Cindy likes everything that Bill likes) ***Кейтлин любит все зеленое.*** (Caitlin likes everything that is green). Используя эти правила, вы можете из предыдущих фактов найти некоторые вещи, которые любят Синди и Кейтлин: ***Синди любит Синди***. (Cindy likes Cindy) ***Кейтлин любит Кермит.*** (Caitlin likes Kermit).

Чтобы перевести эти правила на Пролог, вам нужно немного изменить синтаксис, подобно этому:

likes(cindy,Something):- *likes(bill, Something),*

likes(Caitlin,Something):- *green(Something)*.

Символ :- имеет смысл «если», и служит для разделения двух частей правила: заголовка и тела.

Вы можете рассматривать правило и как процедуру. Другими словами, эти правила

likes(cindy,Something):-likes(bill,Something),

likes(caitlin,Something):-green(Something),

также означают: «Чтобы доказать, что Синди любит что-то, докажите, что Билл любит это» и «Чтобы доказать, что Кейтлин любит что-то, докажите, что это что-то зеленое». С такой «процедурной» точки зрения правила могут «попросить» Пролог выполнить другие действия, отличные от доказательств фактов, – такие как напечатать что-нибудь или создать файл.

Запросы

Однократно дав языку Пролог несколько фактов, мы можем задавать вопросы, касающиеся отношений между ними. Это называется запросом (query) системы языка Пролог. Мы можем задавать Прологу такие же вопросы, которые мы могли бы задать вам об этих отношениях. Основываясь на известных, заданных ранее фактах и правилах, Вы можете ответить на вопросы об этих отношениях, в точности так же это может сделать Пролог.

На естественном языке мы спрашиваем:

Does Bill like Cindy? (Билл любит Синди?) По правилам Пролога мы спрашиваем: *likes(bill, cindy)*.

Получив такой запрос, Пролог мог бы ответить: **yes (да),**
потому что Пролог имеет факт, подтверждающий, что это так. Немного усложнив вопрос, мы могли бы спросить Вас на естественном языке:

What does Bill like? (Что любит Билл?);

По правилам Пролога мы спрашиваем:

likes(bill, What).

Заметим, что синтаксис Пролога не изменяется, когда Вы задаете вопрос: этот запрос очень похож на факт. Впрочем, важно отметить, что второй объект – **What** – начинается с большой буквы, тогда как первый объект – **bill** – нет. Это происходит потому, что **bill** – фиксированный, постоянный объект – **известная величина**, а **What – переменная**. Переменные всегда начинаются с заглавной буквы или символа подчеркивания.

Пролог всегда ищет ответ на запрос, начиная с первого факта, и перебирает все факты, пока они не закончатся. Получив запрос о том, что Билл любит, Пролог ответит:

What=cindy What=dogs 2 Solutions.

Так как ему известно, что *likes(bill, cindy)* и *likes(bill, dogs).*

Если бы Пролог спросил Вас:

What does Cindy like? (Что любит Синди?)

likes(cindy, What).

то Пролог ответил бы:

What = bill What = cindy,

What = dogs 3 solutions,

поскольку Пролог знает, что Синди любит Билла, и что Синди любит то же, что и Билл, и что Билл любит Синди и собак.

Мы могли бы задать Прологу и другие вопросы, которые можно задать человеку. Но вопросы типа ***«Какую девушку любит Билл?»*** не получат решения, т. к. Прологу в данном случае не известны факты о девушке, а он не может вывести заключение, основанное на неизвестных данных: и этом примере мы не дали Прологу какого-нибудь отношения или свойства, чтобы определить, являются ли какие-либо объекты девушками.

Размещение фактов, правил и запросов

Предположим, у вас есть следующие факты и правила:

Быстрая машина – приятная. (A fast car is fun).

Большая машина – красивая. (A big car is nice).

Маленькая машина – практичная. (A little car is practical).

Биллу нравится машина, если она приятная. (Bill likes a car if the car is fun).

Исследуя эти факты, можно сделать вывод, что Биллу нравится быстрый автомобиль. В большинстве случаев Пролог придет к подобному решению. Если бы не было фактов о быстрых автомобилях, вы не смогли бы логически вывести, какие автомобили нравятся Биллу. Вы можете делать предположения о том, какой тип машин может быть крепким, но Пролог знает только то, что вы ему скажете; Пролог не строит предположений.

Вот пример, демонстрирующий, как Пролог использует правила для ответа на запросы.

```
likes(ellen,tennis).
likes(John,football).
likes(torn,baseball).
likes (eric,swimming).
likes(mark,tennis).
likes(bill,Activity):-likes(torn,Activity).
```

Последняя строка является правилом:

likes(bill. Activity):- likes(torn. Activity).

Это правило соответствует предложению естественного языка:

Биллу нравится занятие, если Тому нравится это занятие

(Bill likes an activity if Tom likes that activity).

В данном правиле заголовок – это *likes (bill, Activity)*, а тело – *likes (torn, Activity)*. Заметим, что в этом примере нет фактов о том, что Билл любит бейсбол. Чтобы выяснить, любит ли Билл бейсбол, можно дать Прологу такой запрос:

likes(bill, baseball).

Пытаясь отыскать решение по этому запросу Пролог будет использовать правило:

likes(bill, Activity):- likes(torn, Activity).

```
predicates
  likes(symbol,symbol) – nondeterm (i,i)
clauses
  likes(ellen,tennis).
  likes(john,football).
  likes(tom,baseball).
  likes(eric,swimming).
  likes(mark,tennis).
  likes(bill,Activity):-
     likes(tom, Activity).
goal
  likes(bill,baseball).
```

Утилита Test Goal ответит в окне приложения:

yes (да).

Система использовала комбинированное правило

likes(bill, Activity):- likes(torn, Activity).

с фактом

likes (torn, baseball).

для решения, что

likes(bill, baseball).

Попробуйте также следующий запрос в GOAL-разделе:

likes(bill, tennis).

Утилита Test Goal ответит:

no (нет).

Visual Prolog ответит по на последний запрос ***«Does Bill like tennis?»*** (Любит ли Билл теннис), поскольку:

- нет фактов, которые говорят, что Билл любит теннис;
- отношение Билла к теннису не может быть логически выведено с использованием данного правила и имеющихся в распоряжении фактов.

Вполне возможно, что Билл любит теннис в реальной жизни, но ответ Visual Prolog основан только на фактах и правилах, которые Вы дали ему в тексте программы.

Переменные: общее представление

В Прологе переменные позволяют вам записывать общие факты и правила и задавать общие вопросы. В естественном языке Вы пользуетесь переменными в предложениях постоянно. Обычное предложение на английском языке может быть таким:

Bill likes the same thing as Kim. (Билл любит то же, что и Ким).

Как мы говорили, при задании переменной в Прологе первый символ имени должен быть заглавной буквой или символом подчеркивания. Например, в следующей строке **Thing** – это переменная.

likes(bill. Thing):- likesfkim, Thing).

В предшествующем примере:

likes(cindy, Something):- likes(bill, Something),

объект **Something** начинается с заглавной буквы, т. к. это переменная; он определяет что-то, что Билл любит. С таким же успехом этот объект мог бы называться X или Z.

Объекты bill и cindy начинаются со строчной буквы, т. к. они не являются переменными – это идентификаторы, имеющие постоянное значение. Visual Prolog может обрабатывать произвольные текстовые строки подобно тому, как мы оперировали символами, упомянутыми выше, если текст заключен в двойные кавычки. Следовательно, вместо bill вы могли бы также успешно написать «Bill».

Краткий обзор

Программы на языке Пролог состоят из двух типов фраз: фактов и правил, также называемых предложениями.

- Факты – это отношения или свойства, о которых известно, что они имеют значение «истина».
- Правила – это связанные отношения; они позволяют Прологу логически выводить одну порцию информации из другой. Правило принимает значение *«истина»*, если доказано, что заданный набор условий является истинным.

В Прологе все правила имеют 2 части: заголовок и тело, разделенные специальным знаком **:–**.

- Заголовок – это факт, который был бы истинным, если бы были истинными несколько условий. Это называется выводом или зависимым отношением.
- Тело – это ряд условий, которые должны быть истинными, чтобы Пролог мог доказать, что заголовок правила истинен.

Как вы уже, наверное, заметили, факты и правила – практически одно и то же, кроме того, что факты не имеют явного тела. Факты ведут себя так, как если бы они имели тело, которое всегда истинно.

Пролог всегда ищет решение, начиная с первого факта и/или правила, и просматривает весь список фактов и/или правил до конца.

Механизм логического вывода Пролога берет условия из правила (тело правила) и просматривает список известных фактов и правил, пытаясь удовлетворить условиям. Если все условия истинны, то зависимое отношение (заголовок правила) считается истинным. Если все условия не могут быть согласованы с известными фактами, то правило ничего не выводит.

Ранее в этой главе мы говорили о фактах и правилах, отношениях, основных конструкциях и запросах. Все эти термины являются частью логики и естественного языка. Сейчас мы будем обсуждать те же поня-

тия, но используя термины Пролога, – ***предложения***, ***предикаты***, ***переменные*** и ***цели***.

Предложения

По сути, есть только два типа фраз, составляющих язык Пролога: фраза может быть либо **фактом**, либо **правилом**. Эти фразы в Прологе известны под термином ***clause*** (предложение). Сердце программ на Прологе состоит из предложений.

Подробнее о фактах. Факт представляет либо свойство объекта, либо отношение между объектами. Факт самодостаточен. Прологу не требуется дополнительных сведений для подтверждения факта, и факт может быть использован как основа для логического вывода.

Подробнее о правилах. В Прологе, как и в обычной жизни, можно судить о достоверности чего-либо, логически выведя это из других фактов. **Правило** – это конструкция Пролога, которая описывает, что можно логически вывести из других данных. **Правило** – это свойство или отношение, которое достоверно, когда известно, что ряд других отношений достоверен. Синтаксически эти отношения разделены запятыми.

Рассмотрим несколько примеров работы с правилами.

Пример, иллюстрирующий правило, которое может быть использовано для того, чтобы сделать вывод, подходит ли некоторое блюдо из меню Диане:

Диана – вегетарианка и ест только то, что говорит ей ее доктор. (Diane is a vegetarian and eats only what her doctor tells her to eat).

Зная меню и предыдущее правило, Вы можете сделать вывод о том, выберет ли Диана данное блюдо. Чтобы выполнить это, Вы должны проверить, соответствует ли блюдо заданному ограничению:

- является ли *Food on menu* овощем?
- находится ли *Food_on_menu* в списке доктора?
- заключение: если оба ответа положительны, Диана может заказать *Food_on_menu*.

В Прологе подобное отношение должно быть выражено правилом, т. к. вывод основан на фактах. Вот один вариант записи правила:

```
diane_can_eat(Food_on_menu):-vegetable(Food_on_menu),
  on_doctor_list (Food_on_menu).
```

Обратите внимание, что после *vegetable(Food_on_menu)* стоит запятая. Она обозначает ***конъюнкцию*** нескольких целей и читается как **«и»**; оба правила – *vegetable(Food_on__menu)* **и** *on_doctor_list(Food_on_menu)* – должны быть истинны для истинности *diane_can_eat (Food_on_menu)*.

Вот другой пример:

Человек может купить машину, если машина ему нравится (likes), и если машина продается (for sale). Это отношение может быть переведено на язык Пролог следующим правилом:

```
can_buy(Name,Model):-person(Name),
car(Model),
likes(Name,Model),
for_sale(Model).
```

Это правило выражает следующие отношения: Name can_buy (может купить) Model, если Name является person (человеком), и Model является car (машиной), и Name likes (нравится) Model, и Model for_sale (продается). Это правило будет истинным, если истинны все 4 условия в теле правила. В листинге представлена программа, которая ищет решение проблемы покупки автомобиля.

```
predicates
  can_buy(symbol,symbol)      –
  nondeterm (o,o)
  person(symbol)      –      nonde-
  term (o)
  car(symbol) – nondeterm (o)
  likes(symbol,symbol)  –  nonde-
  term (i,i)
  for_sale(symbol)      –      nonde-
  term (i)
clauses
  can  buy(X,Y):-
    person(X),
    car(Y),
    likes(X,Y),
    for_sale(Y).
  person(kelly).
  person(judy).
  person(ellen).
  person(mark).
  car(lemon).
  car(hot_rod).
  likes(kelly, hot_rod).
  likes(judy, pizza).
  likes(ellen, tennis).
  likes(mark, tennis).
  for_sale(pizza).
  for_sale(lemon).
  for_sale(hot_rod).
goal
  can_buy(Who,What).
```

Предикаты

Отношение в Прологе называется предикатом. Аргументы – это объекты, которые связываются этим отношением; в факте likes (bill, cindy) отношение likes – это предикат, а объекты bill и cindy – аргументы.

Вот несколько примеров предикатов с различным числом аргументов:

pred(integer,symbol),
person(last, first, gender),
run(),
birthday(firstName, lastName, date).

В вышеприведенном примере показано, что предикаты могут вовсе не иметь аргументов, но использование таких предикатов ограничено. Чтобы выяснить имя Mr. Rosemont, можно применить запрос person *(rosemont, Name, male)*. Но что делать с запросом без аргументов run()? Выясним, есть ли в программе предложение run, и если run – это заголовок правила, то можно вычислить данное правило. В некоторых случаях это оказывается полезным – например, если бы вы захотели создать программу, работающую по-разному в зависимости от того, имеется ли предложение run.

Переменные

В простом запросе, чтобы найти того, кто любит теннис, можно использовать переменные. Например:

likes(X, tennis).

В этом запросе буква *X* используется как переменная для нахождения неизвестного человека. Имена переменных в Visual Prolog должны начинаться с заглавной буквы (или с символа подчеркивания), после которой может стоять любое количество букв (заглавных или строчных), цифр или символов подчеркивания. Ниже приведены правильные имена переменных:

My_first_correct_variable_name,
Sales_10_ll_86,
следующие три – неправильные:
lstattempt,
second_attempt
«disaster».

Удобно использовать в названии переменной буквы разного регистра:
IncomeAndExpenditureAccount.

Осмысленный выбор имен переменных делает программу более удобной для чтения. Например:

likes(Person, tennis) лучше, чем *likes(X, tennis).*

потому что Person имеет больше смысла, чем X.

Инициализация переменных

Вы уже могли заметить, что Пролог не имеет оператора присваивания. Это важное отличие Пролога от других языков программирования.

Переменные в Прологе инициализируются при сопоставлении с константами в фактах или правилах.

До инициализации переменная свободна; после присвоения ей значения она становится связанной. Переменная остается связанной только то время, которое необходимо для получения решения по запросу, затем Пролог освобождает ее и ищет другое решение.

Нельзя сохранить информацию, присвоив значение переменной. Переменные используются как часть процесса поиска решения, а не как хранилище информации.

В примере, приведенном в листинге, программа может быть рассмотрена в качестве иллюстрации того, как и когда переменные получают свои значения.

```
predicates
  likes(symbol,symbol) – nondeterm (o,i), nondeterm (i,i)

clauses
  likes(ellen,reading).
  likes(john,computers).
  likes(john,badminton).
  likes(leonard,badminton).
  likes(eric,swimming).
  likes(eric,reading).

goal
  likes(Person,reading),
  likes(Person,swimming).
```

Рассмотрим запрос; ***есть ли человек, который любит и чтение, и плавание?*** *likes(Person, reading),*

likes(Person, swimming).

Пролог будет решать обе части запроса посредством поиска фактов с начала и до конца программы. В первой части запроса

likes(Person, reading).

переменная **Person** свободна; ее значение неизвестно перед тем, как Пролог попытается найти решение. С другой стороны, второй аргумент, **reading**, известен. Пролог ищет факт, который соответствует первой части запроса. Первый факт в программе

likes(ellen,reading).

удовлетворяет первой части запроса, значит Пролог связывает свободную переменную **Person** со значением *ellen* соответствующим значению в факте. В то же время Пролог помешает указатель в список фактов, показывающий, как далеко продвинулась процедура поиска.

Далее, для полного разрешения запроса (поиск человека, которому нравится и чтение, и плавание) должна быть решена вторая часть запроса. Так как **Person** сейчас связана со значением *ellen*, Пролог должен искать факт

likes(ellen,swimming).

Пролог ищет этот факт от начала программы, но совпадений нет (потому что в программе нет такого факта). Вторая часть запроса ложна, если **Person** имеет значение *ellen*.

Теперь Пролог освободит переменную **Person** и попытается найти иное решение первой части запроса. Поиск другого факта, удовлетворяющего первой части запроса, начинается с указателя в списке фактов.

Пролог ищет следующего человека, кто любит чтение и находит факт

likes (eric, reading).

Переменная **Person** сейчас связана со значением *eric*, и Пролог пытается вновь найти соответствие со второй частью запроса посредством поиска в программе факта:

likes(eric, swimming).

Пролог находит совпадение (последнее предложение в программе), и запрос полностью удовлетворяется. Пролог (Test Goal) возвращает

Person=eric. 1 Solution.

Анонимные переменные

Анонимные переменные позволяют «привести в порядок» наши программы. Если вам нужна только определенная информация запроса, можно использовать анонимные переменные для игнорирования ненужных значений. В Прологе анонимные переменные обозначаются символом подчеркивания (__).

Следующий пример демонстрирует использование анонимных переменных.

```
predicates
  male(symbol) – nondeterm (o)
  female(symbol) – nondeterm (o)
  parent(symbol,symbol) – nondeterm (o,o)

clauses
  male(bill).
  male(joe).
  female(sue).
```

```
  female(tammy).
  parent(bill,joe).
  parent(sue,joe).
  parent(joe,tammy).

goal
  parent(Parent,_).
```

Анонимная переменная может быть использована на месте любой другой переменной, и ей никогда не присваивается значение.

Например, в следующем запросе нам понадобится узнать, какие люди являются родителями, но нам неинтересно, кто их дети. Пролог знает, что каждый раз, когда Вы используете символ подчеркивания в запросе, Вам не нужна информация о значении, представленном на месте переменной.

goal
parent(Parent, _).

Получив такой запрос, Пролог (Test Goal) отвечает:

```
Parent=bill
Parent=sue
Parent=joe
3 Solutions
```

В этом случае Пролог находит и выдает трех родителей, но он не выдает значения, связанные со вторым аргументом в предложении *parent*.

Анонимные переменные также можно использовать в фактах.

Цели (запросы)

До сих пор мы, говоря о вопросах, задаваемых Прологу, употребляли слово «запрос». Далее мы будем использовать более общее слово «цель». Трактовка запросов как целей такова: когда Вы даете Прологу запрос, в действительности Вы даете ему цель для выполнения. («Найди ответ на вопрос, если он существует:…»)

Цели могут быть или простыми:

likes(ellen, swimming).
likes(bill, What).

или более сложными

likes(Person, reading),likes(Person, swimming).

Цель, состоящая из двух и более частей, называется сложной целью, а каждая часть сложной цели называется подцелью.

Часто бывает нужно найти общее решение двух целей. Например, в предыдущем примере о родителях Вы могли бы поинтересоваться, которые из родителей являются родителями мужского пола. Искать решение для такого запроса можно, задав *сложную цель:*

Goal parent(Person, _),
male(Person).

Сначала Пролог попытается решить подцель parent *(Person, _)* путем поиска соответствующего предложения и последующего связывания переменной **Person** со значением, возвращенным *parent.* Значение, возвращаемое *parent*, далее предоставляется второй подцели в качестве значения, с которым будет продолжен поиск: является ли **Person** (теперь это связанная переменная) мужского пола?

male(Person).

Если цель задана корректно. Пролог ответит:

Person=bill Person=joe 2 Solutions.

Составные цели: конъюнкция и дизъюнкция

Как вы уже видели, составные цели можно использовать для поиска решения, в котором обе подцели A и B истинны (конъюнкция), разделяя подцели запятой. Вы также можете искать решения в том случае, если истинна либо подцель A, либо подцель B (дизъюнкция), разделяя подцели точкой с запятой. Ниже представлен пример программы, иллюстрирующей эту идею.

```
predicates
  car(symbol,long,integer,symbol,long) – nondeterm (o,o,o,o,i)
  truck(symbol,long,integer,symbol,long) – nondeterm (o,o,o,o,i)
  vehicle(symbol,long,integer,symbol,long) – nondeterm (o,o,o,o,i)
clauses
  car(chrysler,130000,3,red,12000).
  car(ford,90000,4,gray,25000).
  car(datsun,8000,1,red,30000).

  truck(ford,80000,6,blue,8000).
  truck(datsun,50000,5,orange,20000).
  truck(toyota,25000,2,black,25000).

  vehicle(Make,Odometer,Age,Color,Price):-
  car(Make,Odometer,Age,Color,Price);
  truck(Make,Odometer,Age,Color,Price).

goal
  car(Make,Odometer,Years_on_road,Body,25000).
```

Данная цель попытается найти описанную в предложениях машину (car), которая стоит ровно $25 000. Пролог ответит:

Make=ford, Odometer=90000, Years_on_road=4, Body=gray 1 Solution

Следует заметить, что данная цель несколько неестественна, так как скорее всего будет задан вопрос типа: **Есть ли в списке машина, стоящая меньше, чем $25 000?** (Is there a car listed that costs less than $25 000?)

Для поиска такого решения Вы можете задать Visual Prolog следующую составную цель:

car(Make, Odometer, Years_on_road, Body, Cost) – подцель **A** и

Cost < 25000.– подцель **B.**

Это и является конъюнкцией. Для разрешения этой составной цели Пролог будет пытаться по очереди решать подцели. Вначале он попытается решить:

car(Make, Odometer, Years_on_road, Body, Cost), а затем

Cost < 25000.

с переменной **Cost**, имеющей идентичное значение в обеих подцелях.

Подцель *Cost < 25000* соответствует отношению «меньше чем», которое встроено в систему Visual Prolog. Отношение «меньше чем» ничем не отличается от любого другого отношения, использующего два числовых объекта, но правильнее применять для его обозначения символ (<), помещая его между двумя объектами.

Посмотрим, является ли истинным следующее выражение, на естественном языке оно звучит так: ***Есть ли в списке автомобиль стоимостью меньше $25000 или грузовик стоимостью меньше $20000?***

При задании следующей составной цели Пролог выполнит поиск требуемого решения:

car(Make,Odometer,Years_on_road,Body,Cost), Cost < 25000 – подцель **A.**

ИЛИ

truck(Make,Odometer,Years_on_road,Body,Cost), Cost < 20000. – подцель **B.**

Этот тип составной цели является дизъюнкцией. Данная цель установила две альтернативные подцели так же, как если бы это были два предложения одного правила. Пролог будет искать все решения, удовлетворяющие обеим подцелям.

При разрешении такой составной цели Пролог вначале попытается решить первую подцель, состоящую из следующих подцелей:

car(Make, Odometer, Years_on_road, Body, Cost)

И

Cost < 25000.

Если автомобиль найдется – цель истинна; если нет – Пролог попытается разрешить вторую составную цель, состоящую из подцелей:

truck(Make, Odometer, Years_on_road, Body, Cost),

И

Cost < 20000.

Комментарии

Хорошим стилем программирования является включение в программу комментариев, объясняющих все то, что может быть непонятно кому-то другому (или даже Вам, спустя полгода). Если Вы подберете подходящие имена для переменных, предикатов и доменов, то Вам понадобится меньше комментариев, т. к. программа будет объяснять себя «сама».

Многострочные комментарии должны начинаться с символов /* (косая черта, звездочка) и завершаться символами */ (звездочка, косая черта). Для установки однострочных комментариев можно использовать либо эти же символы, либо начинать комментарий символом процента (%).

Сопоставление

В предыдущих разделах Вы познакомились с тем, как Пролог «сопоставляет вопросы и ответы», «ищет сопоставление», «сопоставляет условия с фактами», «сопоставляет переменные с константами» и т. д. Ниже рассмотрим, что же понимается под термином сопоставление (matching).

В Прологе имеется несколько примеров сопоставления одной вещи с другой:

parent (joe, tammy) сопоставимо с *parent (joe, tammy)*.

Сопоставление (сравнение) обычно использует одну или несколько свободных переменных. Например, если **X** свободна, то parent (joe, X) сопоставимо с parent (joe, tammy) и **X** принимает значение (связывается с) **tammy**.

Если же **X** уже связана, то она действует так же, как обычная константа. Таким образом, если **X** связана со значением **tammy**, то *parent (joe, X)* сопоставимо с *parent (joe, tammy)*, но *parent(joe, X)* не сопоставимо с *parent(joe, millie)*.

Как может переменная оказаться связанной при попытке Пролога сопоставить ее с чем-либо? Вспомним, что переменные не могут хранить значения, т. к. они становятся связанными только на промежуток времени, необходимый для отыскания (или попытки отыскания) одного решения цели. Поэтому имеется только одна возможность для переменной оказаться связанной – перед попыткой сопоставления, если цель требует больше одного шага, и переменная стала связанной на предыдущем шаге.

Например:

parent(joe,X), parent(X,jenny)

является корректной целью. Она означает: ***Найти кого-либо, являющегося ребенком Joe и родителем Jenny***. Здесь при достижении подцели *parent(X, jenny)* переменная **X** уже будет связана. Если для подцели *parent(X,jenny)* нет решений, Пролог «развяжет» переменную **X** и вернется назад, пытаясь найти новое решение для *parent(joe,X)*, а затем проверит, будет ли «работать» *parent(X, jenny)* с новым значением **X**.

Две свободные переменные могут сопоставляться друг с другом. Например, *parent (joe, X)* сопоставляется с *parent (joe, Y)*, связывая при этом переменные **X** и **Y** между собой. С момента «связывания» **X** и **Y** трактуются как одна переменная, и любое изменение значения одной из них приводит к немедленному соответствующему изменению другой. В случае подобного «связывания» между собой нескольких свободных переменных все они называются совмещенными свободными переменными. Некоторые методы программирования специально используют «взаимосвязывание» свободных переменных, являющихся, на самом деле, различными.

В Прологе связывание переменных (со значениями) производится двумя способами: на входе и выходе. Направление, в котором передаются значения, указывается в шаблоне потока параметров (flow pattern). В дальнейшем (для краткости) будем опускать слово «шаблон» и говорить просто «поток параметров». Когда переменная передается в предложение, она считается входным аргументом и обозначается символом (i). Когда же переменная возвращается из предложения, она является выходным аргументом и обозначается символом (o).

3.3. Программы Visual Prolog

Синтаксис

Visual Prolog разработан для того, чтобы отображать знания о свойствах и взаимосвязях. В основном, Вы уже видели, как это делается: в данной главе мы рассматривали предложения (факты и правила), предикаты, переменные и цели.

В отличие от других версий Пролога, Visual Prolog – компилятор, контролирующий типы: для каждого предиката объявляются типы объектов, которые он может использовать. Это объявление типов позволяет программам Visual Prolog быть скомпилированными непосредственно в машинные коды, при этом, скорость выполнения сравнима, а в некоторых случаях – и превышает скорости аналогичных программ на языках C и Pascal.

Теперь обсудим четыре основных раздела программ на Visual Prolog – те, где объявляются и описываются предикаты и типы аргументов, задаются правила и определяется цель программы. Далее более подробно рассмотрим синтаксис правил и объявлений. И наконец, в заключение кратко опишем другие разделы программ: базы данных, константы, различные глобальные разделы и директивы компилятора.

Основные разделы Visual Prolog-программ

Обычно программа на Visual Prolog состоит из четырех основных программных разделов. К ним относятся:

- clauses (предложения),
- predicates (предикаты),

- domains (домены),
- goal (цели).

Раздел **clauses** – это сердце Visual Prolog-программы; именно в этот раздел записываются факты и правила, которыми будет оперировать Visual Prolog, пытаясь разрешить цель программы.

Раздел **predicates** – это тот, в котором объявляются предикаты и домены (типы) их аргументов.

Раздел **domains** служит для объявления всех используемых вами доменов, не являющихся стандартными доменами Visual Prolog.

Раздел **goal** – это тот, в который Вы помещаете цель Visual Prolog-программы.

В раздел **clauses** (предложений) Вы помещаете все факты и правила, составляющие вашу программу. Основное внимание в этой главе было уделено рассмотрению предложений (фактов и правил) программы: что они означают, как их писать и т. д.

Если Вы поняли, что собой представляют факты и правила и как их записывать в Прологе, то Вы знаете, что все предложения для каждого конкретного предиката в разделе clauses должны располагаться вместе. Последовательность предложений, описывающих один предикат, называется процедурой.

Пытаясь разрешить цель, Visual Prolog (начиная с первого предложения раздела clauses) будет просматривать каждый факт и каждое правило, стремясь найти сопоставление. По мере продвижения вниз по разделу clauses он устанавливает внутренний указатель на первое предложение, являющееся частью пути, ведущего к решению. Если следующее предложение не является частью этого логического пути, то Visual Prolog возвращается к установленному указателю и ищет очередное подходящее сопоставление, перемещая указатель на него (этот процесс называется поиск с возвратом).

Раздел предикатов

Если в разделе **clauses** программы на Visual Prolog Вы описали собственный предикат, то Вы обязаны объявить его в разделе **predicates** (предикатов); в противном случае Visual Prolog не поймет, о чем Вы ему «говорите». В результате объявления предиката Вы сообщаете, к каким доменам (типам) принадлежат аргументы этого предиката.

Visual Prolog поставляется с большим набором встроенных предикатов (их не нужно объявлять), а интерактивное справочное руководство предоставляет полное их описание.

Предикаты задают факты и правила. В разделе же **predicates** все предикаты просто перечисляются с указанием типов (доменов), их аргументов. Эффективность работы Visual Prolog значительно возрастает

именно из-за того, что Вы объявляете типы объектов (аргументов), с которыми работают ваши факты и правила.

Как объявить пользовательский предикат

Объявление предиката начинается с имени этого предиката, за которым идет открывающая (левая) круглая скобка, после чего следует ноль или больше доменов (типов) аргументов предиката:

predicateName(argument_typel OptionalNamel, argument_type2 OptionalName2, ..., argument_typeN OptionalName3)

После каждого домена (типа) аргумента следует запятая, а после последнего типа аргумента – закрывающая (правая) скобка. Отметим, что, в отличие от предложений в разделе clauses, декларация предиката не завершается точкой. Доменами (типами) аргументов предиката могут быть либо стандартные домены, либо домены, объявленные вами в разделе domains. Можно указывать имена аргументов optionalName – это улучшает читаемость программы, и не сказывается на скорости ее исполнения, т. к. компилятор их игнорирует.

Имена предикатов

Имя предиката должно начинаться с буквы, за которой может располагаться последовательность букв, цифр и символов подчеркивания. Регистр букв не имеет значения, однако мы не советуем вам использовать заглавные буквы в качестве первой буквы имени предиката. Имя предиката может иметь длину до 250 символов.

В именах предикатов запрещается использовать пробел, символ минус, звездочку и другие алфавитно-цифровые символы. Корректные имена Visual Prolog могут включать символы, перечисленные в табл. 3.2.

Таблица 3.2

Символы, используемые в именах предикатов в Visual Prolog

Название символов	Примеры символов
Заглавные буквы	A, B, …, Z
Строчные буквы	a, b…z
Цифры	0, 1…9
Символ подчеркивания	_

Имена предикатов и аргументов могут состоять из любых комбинаций этих символов при условии, что Вы подчиняетесь правилам построения соответствующих имен.

Аргументы предикатов

Аргументы предикатов должны принадлежать доменам, известным Visual Prolog. Эти домены могут быть либо стандартными доменами, либо некоторыми из тех, что Вы объявили в разделе доменов.

Рассмотрим несколько примеров.

Если предикат *my_predicate(symbol,integer)* объявлен в разделе predicates следующим образом:

predicates

my_predicate(symbol, integer)

то Вам не нужно в разделе *domains* декларировать домены его аргументов, т. к. *symbol* и *integer* – стандартные домены. Но если этот же предикат Вы объявляете так:

predicates

my_predicate(name, number),

то необходимо объявить, что *name* (символический тип) и *number* (целый тип) принадлежат к стандартным доменам *symbol* и *integer*:

```
domains
name = symbol
number = integer
predicates
     my_predicate(name, number)
```

Раздел доменов

Домены позволяют задавать разные имена различным видам данных, которые в противном случае будут выглядеть абсолютно одинаково. В программах Visual Prolog объекты в отношениях (аргументы предикатов) принадлежат доменам, причем это могут быть как стандартные, так и описанные вами специальные домены.

Раздел *domains* служит двум полезным целям. Во-первых, Вы можете задать домена осмысленные имена, даже если внутренне эти домены аналогичны уже имеющимся стандартным. Во-вторых, объявление специальных доменов используется для описания структур данных, отсутствующих в стандартных доменах.

Иногда очень полезно описать новый домен – особенно, когда Вы хотите прояснить отдельные части раздела *predicates*. Объявление собственных доменов, благодаря присваиванию осмысленных имен типам аргументов, помогает документировать описываемые вами предикаты.

Рассмотрим несколько примеров.

Данный пример показывает, как объявление доменов помогает документированию предикатов:

Франк – мужчина, которому 45 лет.

Используя стандартные домены, Вы можете так объявить соответствующий предикат:

person(symbol, symbol, integer).

В большинстве случаев такое объявление будет работать хорошо. Предположим, что несколько месяцев спустя после написания программы, Вы решили скорректировать ее. Есть опасение, что подобное объявление этого предиката абсолютно ничего вам не скажет. И напротив, декларация этого же предиката, представленная ниже, поможет вам разобраться в том, что же представляют собой аргументы данного предиката:

```
domains
name, sex = symbol
age= integer
predicates
      person(name, sex, age)
```

Одним из главных преимуществ объявления собственных доменов является то, что Visual Prolog может отслеживать ошибки типов, например, такие:

same_sex(X,Y):-person(X, Sex, _), person(Sex, Y, _).

Несмотря на то, что и name и sex описываются как symbol, они не эквивалентны друг другу. Это и позволяет Visual Prolog определить ошибку, если Вы перепутаете их. Это полезно в тех случаях, когда ваши программы очень велики и сложны.

Почему же мы не можем использовать специальные домены для объявления всех аргументов, если они привносят больше смысла в обозначение аргументов? Ответ заключается в том, что аргументы с типами из специальных доменов не могут смешиваться между собой, даже если эти домены одинаковы. Именно поэтому, несмотря на то, что *name* и *sex* принадлежат одному домену *symbol*, они не могут смешиваться. Однако все собственные домены пользователя могут быть сопоставлены стандартным доменам.

Следующий пример программы при его загрузке приведет к ошибке типа:

```
domains
  product,sum = integer

predicates
  add_em_up(sum,sum,sum) – procedure (i,i,o)
  multiply_em(product,product,product) – procedure (i,i,o)
clauses
  add_em_up(X,Y,Sum):-
     Sum=X+Y.
  multiply_em(X,Y,Product):-
     Product=X*Y.
```

```
goal
  add_em_up(32,54,Sum).
Visual Prolog (Test Goal) ответит:
Sum=86
1 Solution
```

что является суммой двух целых чисел, которые Вы передали в программу.

С другой стороны, эта же программа с помощью предиката *multiply_em* умножает два аргумента. Поэкспериментируем. Если Вам нужно узнать произведение чисел 13 и 31, введите цель:

multiply_em(31, 13, Product).

Visual Prolog вернет Вам корректный результат:

Product=403 1 Solution.

Предположим, что Вам понадобилась сумма чисел 42 и 17; цель выглядит так:

add_em_up(42, 17, Sum).

А теперь допустим, Вы хотите удвоить произведение 31 на 17. Задаете следующую цель:

multiply_em(31, 17, Sum),
add_em_up(Sum, Sum, Answer)

и ждете, что Visual Prolog (Test Goal) ответит:

Sum=527, Answer=1054 1 Solution.

Вместо этого Вы получите ошибку *несоотвествия типа*. Это случилось из-за того, что имела место попытка передать результирующее значение предиката *multiply_em*, которое относится к домену product, в качестве первого и второго аргументов (которые должны относится к домену sum) в предикат *add_em_up*. Это и привело к ошибке, т. к. домен product отличается от домена sum. И хотя оба эти домена соответствуют типу integer, они являются различными.

Поэтому, если переменная в предложении используется более чем в одном предикате, она должна быть одинаково объявлена в каждом из них. Очень важно, чтобы Вы поняли концепцию описанной здесь ошибки типа, что позволит избегать сообщений об ошибках компиляции. Различные автоматические и явные преобразования типов, предлагаемые Visual Prolog, будут описаны ниже.

Как еще можно использовать объявления доменов для отслеживания ошибок типа, поможет увидеть следующий пример программы.

```
domains
  brand,color = symbol
  age = byte
  price, mileage = ulong
```

```
predicates
  car(brand,mileage,age,color,price) – nondeterm (i,i,i,i,i)

clauses
  car(chrysler,130000,3,red,12000).
  car(ford,90000,4,gray,25000).
  car(datsun,8000,1,black,30000).

goal
  car(renault,13,40000,red,12000).
```

Здесь предикат *car*, объявленный в разделе predicates, имеет 5 аргументов. Один из них относится к домену age типа byte. В семействе процессоров x86 тип byte – это 8-битное беззнаковое целое, которое может принимать значения от 0 до 255, включая границы. Аналогично, домены mileage и price типа ulong, который представляет собой 32-битные беззнаковые целые, а домены brand и color – символьного типа (symbol).

Мы обсудим стандартные домены более детально далее. А сейчас загрузите данную программу в проект TestGoal и попытайтесь вычислить по очереди следующие цели:

```
car(renault, 13, 40000, red, 12000).
car(ford, 90000, gray, 4, 25000).
car(red, 30000, 80000, datsun).
```

Каждая из них приведет к ошибке типа. В первом случае это произойдет из-за того, что age должен быть типа byte. Следовательно, Visual Prolog сможет легко определить, что при вводе этой цели объекты mileage и age в предикате car были перепутаны местами. Во втором случае были перепутаны age и color, а в третьем – попытайтесь найти ошибку самостоятельно.

Раздел цели

По существу, раздел ***goal*** (цели) аналогичен телу правила: это просто список подцелей. Цель отличается от правила лишь следующим:

- за ключевым словом ***goal*** не следует (:-);
- при запуске программы Visual Prolog автоматически выполняет цель.

Это происходит так, как будто Visual Prolog вызывает ***goal***, запуская тем самым программу, которая пытается разрешить тело правила ***goal***. Если все подцели в разделе ***goal*** истинны – программа завершается успешно. Если же какая-то подцель из раздела ***goal*** ложна, то считается, что программа завершается неуспешно (хотя чисто внешне никакой разницы в этих случаях нет, – программа просто завершит свою работу).

Декларации и правила

В Visual Prolog есть несколько встроенных стандартных доменов. Их можно использовать при декларации типов аргументов предикатов без описания в разделе domains.

Основные стандартные домены Visual Prolog:

- Short – Короткое, знаковое, (–32 768…32 767),
- Ushort – Короткое, беззнаковое, 16 бит (0…65 535),
- Long –Длинное, знаковое, 32 бит (–2 147 483 648…2 147 483 647),
- Ulong – Длинное, беззнаковое, 32 бит (0…4 294 967 295),
- Integer – Знаковое, количественное, 32 бит (–2147483648…2147483 647),
- Unsigned–Беззнаковое, количественное, 32 бит (0…4 294 967 295),
- Byte–8 бит (0…255),
- Word–16 бит (0…65 535),
- Dword–32 бит (0…4 294 967 295),
- Char–Символ, реализуемый как беззнаковый byte. Синтаксически это символ, заключенный между двумя одиночными кавычками: 'a'.
- Real–Число с плавающей запятой, реализуемое как 8 байт в соответствии с соглашением IEEE; эквивалентен типу double в C.
- String–Последовательность символов, реализуемых как указатель на байтовый массив, завершаемый нулем, как в C. Для строк допускается два формата:
  - Последовательность букв, цифр и символов подчеркивания, причем первый символ должен быть строчной буквой.
  - Последовательность символов, заключенных в двойные кавычки.
- Symbol–последовательность символов, реализуемых как указатель на вход в таблице идентификаторов, хранящей строки идентификаторов. Синтаксис – как для строк.

Синтаксически значение, принадлежащее одному из целочисленных доменов, записывается как последовательность цифр, которой в случае знакового домена может предшествовать не отделенный от нее пробелом знак минус. Имеются также восьмеричные и шестнадцатеричные синтаксисы для основных доменов.

Домены типов *byte*, *word* и *dword* наиболее удобны при работе с машинными числами. В основном используются типы *integer* и *unsigned*, а также *short* и *long*. В объявлениях доменов ключевые слова signed и unsigned могут использоваться вместе со стандартными доменами типов byte, word и dword для построения новых базовых доменов. Так:

domains

i = signed byte

создает новый базовый домен в диапазоне от –128 до +127.

Идентификаторы и строки взаимозаменяемы в вашей программе, Visual Prolog хранит их раздельно. Идентификаторы хранятся в таблице идентификаторов, а для представления используются лишь их индексы в этой таблице, но не сами строки идентификаторов. Это означает, что сопоставление идентификаторов выполняется очень быстро, а в случае если они встречаются в программе несколько раз, то и хранение их компактно. Строки же не хранятся в поисковой таблице, и при необходимости сопоставления Visual Prolog проверяет их символ за символом. Вы сами должны определять, какой домен лучше использовать в каждой конкретной программе.

Задание типов аргументов при декларации предикатов

Объявление доменов аргументов в разделе *predicates* называется заданием типов аргументов. Предположим, имеется следующая связь объектов:

Франк – мужчина, которому 45 лет.

Факт Пролога, соответствующий этому предложению естественного языка, может быть следующим:

person(frank, male, 45).

Для того чтобы объявить *person* (человек), как предикат с этими тремя аргументами, Вы можете разместить в разделе *predicates* строку вида:

person(symbol, symbol, unsigned).

Здесь для всех трех аргументов использованы стандартные домены. Отныне всякий раз при работе с предикатом *person*, Вы должны передавать ему три аргумента, причем первые два должны быть типа *symbol*, а третий – типа *integer*.

Если в программе используются только стандартные домены, то нет необходимости использовать раздел *domain*; Вы уже видели несколько программ такого типа.

Если необходимо описать предикат, который сообщал бы позицию буквы в алфавите, т. е. цель

alphabet_position(Letter, Position)

должна вернуть вам Position = 1, если Letter = a, Position = 2, если Letter = b и т. д. Предложения этого предиката могут выглядеть следующим образом:

alphabet_position(A_character, N).

Если при объявлении предиката используются только стандартные домены, то программе не нужен раздел *domains*. Предположим, что Вы хотите описать предикат так, что цель будет истинна, если *A_character* является *N*-м символом алфавита. Предложения этого предиката будут такими:

```
alphabet_position('a', 1).
alphabet_position('b', 2).
alphabet_position ('c', 3).
alphabet_position (' z ', 26).
```

Вы можете объявить данный предикат следующим образом:

```
predicates
      alphabet_position(char, unsigned)
```

и тогда Вам не будет нужен раздел domains. Если разместить все фрагменты профаммы вместе, получим:

```
predicates
alphabet_position(char, integer)
clauses
alphabet_position('a', 1).
alphabet_position('b', 2).
alphabet_position('c', 3).
……..
alphabet_position ('z', 26).
```

Ниже представлено несколько простых целей, которые Вы можете использовать:

```
alphabet_position('a',1).
alphabet_position(X, 3).
alphabet_position (' z ', What).
```

Арность (размерность)

Арность предиката – это количество аргументов, которые он принимает. Вы можете иметь два предиката с одним и тем же именем, но отличающейся арностью. В разделах *predicates* и *clauses* версии предикатов с одним именем и разной арностью должны собираться вместе; за исключением этого ограничения, различная арность всегда понимается как полное различие предикатов. Проиллюстрируем это примером.

```
domains
person = symbol
predicates
 father(person) – nondeterm (o) /* This person is a father */
 father(person,person)                –                nondeterm (o,o)
 /* One person is the father of the other person */
clauses
 father(Man):-
    father(Man,_).
 father(adam,seth).
 father(abraham,isaac).
goal
 father(X).
```

Синтаксис правил

Правила используются в Прологе в случае, когда какой-либо факт зависит от истинности другого факта или группы фактов. Ниже представлен обобщенный синтаксис правила в Visual Prolog:

Заголовок:-<Подцель>,<Подцель>,...,<Подцель>.

Тело правила состоит из одной или более подцелей. Подцели разделяются запятыми, определяя конъюнкцию, а за последней подцелью правила следует точка.

Каждая подцель выполняет вызов другого предиката Пролога, который может быть истинным или ложным. После того, как программа осуществила этот вызов, Visual Prolog проверяет истинность вызванного предиката, и если это так, то работа продолжается, но уже со следующей подцелью. Если же в процессе такой работы была достигнута точка, то все правило считается истинным; если хоть одна из подцелей ложна, то все правило ложно.

Для успешного разрешения правила Пролог должен разрешить все его подцели и создать последовательный список переменных, должным образом связав их. Если же одна из подцелей ложна, Пролог вернется назад для поиска альтернативы предыдущей подцели, а затем вновь двинется вперед, но уже с другими значениями переменных. Этот процесс называется поиск с возвратом.

Как упоминалось выше, в качестве разделителя заголовка и тела правила Пролог использует знак (:-), который читается как «если» (*if*). Но *if* Пролога отличается от *if*, написанного в других языках, например в Pascal, где условие, содержащееся в операторе *if*, должно быть указано перед телом оператора, который может быть выполнен. Данный тип оператора известен как условный оператор если/тогда (if/then). Пролог же использует другую форму логики в таких правилах. Вывод об истинности заголовка правила Пролога делается, если (после того, как) тело этого правила истинно, например, так:

«HEAD is true if BODY is true (or: if BODY can be done)»

Учитывая вышесказанное, правило Пролога соответствует условной форме **тогда/если** (**then/if**).

Автоматическое преобразование типов

Совсем не обязательно, чтобы при сопоставлении двух Пролог-переменных они принадлежали одному и тому же домену. Переменные могут быть связаны с константами из различных доменов. Такое (избирательное) смешение допускается, т. к. Visual Prolog автоматически выполняет преобразование типов (из одного домена в другой), но только в следующих случаях:

- между строками (string) и идентификаторами (symbol);

- между целыми, действительными и символами (char). При преобразовании символа в числовое значение этим значением является величина символа в коде ASCII.

Аргумент из домена mydom, который объявлен следующим образом:
domains
my_dom = <base domain> % <base domain> – это стандартный домен

может свободно смешиваться с аргументами из этого основного домена и с аргументами всех совместимых с ним стандартных доменов. Если основной домен – *string*, то с ним совместимы аргументы из домена *symbol*; если же основной домен *integer*, то с ним совместимы домены *real*, *char*, *word* и др.

Такое преобразование типов означает, например, что Вы можете:

вызвать предикат с аргументами типа *string*, задавая ему аргументы типа *symbol*, и наоборот:

- передавать предикату с аргументами типа real параметры типа integer;
- передавать предикату с аргументами типа char параметры типа integer;
- использовать в выражениях и сравнениях символы без необходимости получения их кодов в ASCII.

Другие разделы программ

Теперь, когда Вы ознакомились с такими разделами программ Visual Prolog, как *clauses*, *predicates*, *domains* и *goal*, поговорим о некоторых других, часто используемых разделах программ: *facts*, *constants* и различных глобальных (*global*) разделах.

Раздел фактов

Программа на Visual Prolog представляет собой набор фактов и правил. Иногда в процессе работы программы бывает необходимо модифицировать (изменить, удалить или добавить) некоторые из фактов, с которыми она работает. В этом случае факты рассматриваются как динамическая или внутренняя база данных, которая при выполнении программы может изменяться. Для объявления фактов программы, рассматривающихся как части динамической (или изменяющейся) базы данных,Visual Prolog включает специальный раздел – ***facts***.

Ключевое слово *facts* объявляет раздел фактов. Именно в этой секции Вы объявляете факты, включаемые в динамическую базу данных. Отметим, что в ранних версиях Visual Prolog для объявления раздела фактов использовалось ключевое слово *database*, т. е. ключевое слово *facts* – синоним устаревшего ключевого слова *database*. В Visual Prolog есть несколько встроенных предикатов, облегчающих использование динамических фактов.

Раздел констант

В своих программах на Visual Prolog Вы можете объявлять и использовать символические константы. Раздел для объявления констант

обозначается ключевым словом ***constants***, за которым следуют сами объявления, использующие следующий синтаксис:

<Id> = <Макроопределение>

Каждое <макроопределение> завершается символом новой строки и, следовательно, на одной строке может быть только одно описание константы. Объявленные таким образом константы могут позже использоваться в программах.

Рассмотрим фрагмент программы:

```
constants zero = 0
one = 1
two = 2
hundred = (10* (10–1)+10)
pi = 3.141592653
ega = 3
slash_fill = 4
red = 4
```

Перед компиляцией программы Visual Prolog заменит каждую константу на соответствующую ей строку. Например, фрагмент программы

*A = hundred*34, delay(A), setfillstyle(slash_fill, red), Circumf = pi*Diam,*

будет обрабатываться компилятором, как следующий фрагмент:

A= (10(10–1)+10)*34,delay(A),setfillstyle(4,4),Circumf= 3.141592653*Diam.*

На использование символических констант накладываются ограничения: описание константы не может ссылаться само на себя: my_number = 2*my_number/2, это приведет к сообщению об ошибке «Recursion in constant definition» (Рекурсия в описании константы);

- в описаниях констант система не различает верхний и нижний регистры. Следовательно, при использовании в разделе программы *clauses* идентификатора типа *constants*, его первая буква должна быть строчной для того, чтобы избежать путаницы между константами и переменными;
- в программе может быть несколько разделов constants, но объявление константы должно производиться перед ее использованием;
- идентификаторы констант являются глобальными и могут объявляться только один раз. Множественное объявление одного и того же идентификатора приведет к сообщению об ошибке «Constant identifier can only be declared once» (Идентификатор константы может объявляться только один раз).

Глобальные разделы

Visual Prolog позволяет объявлять некоторые разделы *domains*, *predicates*, *clauses* глобальными (а не локальными); сделать это Вы мо-

жете, объявив в своей программе специальные разделы ***global*** *domains*, ***global*** *predicates* И ***global*** *facts*.

Директивы компилятора

Visual Prolog поддерживает несколько директив компилятора, которые можно добавлять в программу для сообщения компилятору специальных инструкций по обработке вашей программы при ее компиляции. Кроме этого, Вы можете устанавливать большинство директив компилятора с помощью команды меню среды визуальной разработки Visual Prolog **(Options|Project|Compiler Options)**.

Директива include

Для того чтобы избежать многократного набора повторяющихся процедур, Вы можете использовать директиву ***include***.

Ниже приведен пример того, как это делается.

Создаете файл (например, MYSTUFF.PRO), в котором объявляете свои наиболее часто используемые предикаты (с помощью разделов domains и predicates) и даете их описание в разделе *clauses*.

В «допустимых областях» исходного текста программы размещаете строку:

include «mystuff.pro»

(«Допустимые области» – это любое место программы, в котором Вы можете расположить декларацию разделов domains, facts, predicates, clauses или goal).

При компиляции исходных текстов программы Visual Prolog вставит содержание файла MYSTUFF.PRO прямо в окончательный текст файла для компиляции.

Директиву *include* можно использовать для включения в исходный текст любого часто используемого фрагмента. Кроме того, любой включаемый в программу файл может, в свою очередь, включать другой файл (каждый файл может быть включен в вашу программу только один раз). Повторим: директива *include* может располагаться в любых «допустимых областях» программы.

Резюме

Ниже приведены основные идеи, представленные нами в этой главе.

Пролог-программа состоит из предложений, которые могут быть фразами двух типов: (фактами или правилами):

факты – это связи или свойства, о которых Вы (программист) твердо знаете, что они истинны;

правила – это зависимые связи (отношения); они позволяют Прологу выводить один фрагмент информации из другого.

Факты имеют общий вид:

property(object1, object2, ..., objectN)

или

relation(object1, object2, ..., objectN)

где *property* – это свойство объектов, а *relation* – отношение между объектами. Различия между этими понятиями несущественны, и поэтому в дальнейшем мы будем использовать термин отношение.

Каждый факт программы задает либо отношение, влияющее на один или более объектов, либо свойство одного или более объектов. Например, в факте Пролога

likes(torn, baseball)

отношение – это *likes* (нравится), а объекты – *torn* и *baseball*; Тому нравится бейсбол.

Правила имеют общую форму Заголовок:-Тело, которые выглядят так:

relation(object,object,...,object):-relation(object,...,object),

relation(object,...,object).

Имена для связей и объектов устанавливаются согласно приведенным ниже ограничениям:

- имя объекта должно начинаться со строчной буквы, за которой может быть любое число символов. Этими символами могут быть: буквы верхнего и нижнего регистров, цифры и символы подчеркивания;
- имена свойств и связей должны начинаться со строчной буквы, за которой может следовать любая комбинация букв, цифр и символов подчеркивания.

Предикат – это символическое имя (идентификатор) связи с последовательностью аргументов. Программа на Прологе – это последовательность предложений и директив, а процедура – это последовательность предложений, описывающих предикат. Предложения, принадлежащие одному предикату, должны следовать друг за другом.

Переменные позволяют Вам записывать общие факты и правила и задавать общие вопросы. Имя переменной в Visual Prolog должно начинаться с заглавной буквы или символа подчеркивания (_), после которой Вы можете использовать любое число букв (верхнего и нижнего регистра), цифр и символов подчеркивания. Переменные в Прологе получают свои значения в результате сопоставления с константами в фактах или правилах. До получения значения переменная является свободной, после – становится связанной. Вы не можете длительно хранить информацию с помощью «связывания» переменной со значением, т. к. переменная является связанной только в пределах предложения. Если в запросе Вас интересует только определенная информация, то для игнорирования не нужных Вам значений вы можете использовать анонимные переменные.

В Прологе анонимные переменные обозначаются одиночным символом подчеркивания (_). Анонимная переменная может быть использована

вместо любой другой переменной; она сопоставляется с любыми данными. Анонимная переменная никогда не принимает какого-либо значения.

Задание вопросов о фактах в программе называется запросами к системе Пролога. Более общим термином для запроса является цель (goal). Пролог пытается разрешить цель (ответить на вопрос), просматривает все факты, начиная с первого до достижения последнего из них.

Составная цель – это цель, включающая две или более частей; каждая часть составной цели называется подцелью. Составная цель может быть конъюнктивной (подцель **A** и подцель **B**) или дизъюнктивной (подцель **A** или подцель **B**).

В Прологе имеется несколько способов сопоставления одного объекта с другим:

- идентичные структуры сопоставляются друг с другом;
- свободная переменная сопоставляется с константой или с ранее связанной переменной (и становится связанной с соответствующим значением);
- две свободные переменные могут сопоставляться (и связываться) друг с другом. С момента связывания они трактуются как одна переменная: если одна из них принимает какое-либо значение, то вторая немедленно принимает то же значение.

Программа на Visual Prolog имеет следующую обобщенную структуру:

```
domains /* …
объявления доменов */
predicates /* …
объявления предикатов */
clauses /* …
предложения (правила и факты) */
goal /* …
подцель _1, подцель _2, и т. д. */
```

В разделе *clauses* вы размещаете факты и правила, с которыми будет работатьVisual Prolog, пытаясь разрешить цель программы.

В разделе вы объявляете предикаты и домены (типы) аргументов этих предикатов. Имена предикатов должны начинаться с буквы (желательно строчной), за которой следует последовательность букв, цифр и символов подчеркивания (до 250 знаков). В именах предикатов нельзя использовать символы пробел, минус, звездочка, слэш. Объявление предиката имеет следующую форму:

predicates

predicateName(argumentType1 OptionalName1, argumentType2 OptionalName2, < ... >;

argumentTypeN OptionalNameN).

Здесь *argument_type1*, …, *argument_typeN* – либо стандартные домены, либо домены, объявленные в разделе *domains*. Объявление домена аргумента и описание типа аргумента – суть одно и то же.

В разделе *domains* объявляются любые нестандартные домены, используемые вами для аргументов предикатов. Домены в Прологе являются аналогами типов в других языках. Основные стандартные домены Visual Prolog: char, byte, short, ushort, word, integer, unsigned, long, ulong, dword, real, string и symbol.

В разделе *goal* Вы задаете внутреннюю цель программы; это позволяет программе быть скомпилированной, запускаться и выполняться независимо от среды визуальной разработки (VDE).

Арность (размерность) предиката – это число принимаемых им аргументов; два предиката с одним именем могут иметь различную арность. Предикаты с отличающимися арностями должны собираться вместе, причем и в разделе predicates, и в разделе clauses, но такие предикаты рассматриваются как абсолютно разные.

Правила имеют форму:

HEAD: – <Subgoal1>, <Subgoal2>, ..., <SubgoalN>.

Для разрешения правила Пролог должен разрешить все его подцели, создав при этом соответствующее множество связанных переменных. Если же одна из подцелей ложна, Пролог возвратится назад и просмотрит альтернативные решения предыдущих подцелей, а затем вновь пойдет вперед, но с другими значениями переменных. Этот процесс называется поиск с возвратом.

Оператор Пролога :- (if) отличается от if, используемых в других языках: правило Пролога работает в соответствии с условной формой, **тогда/если**, тогда как этот оператор в других языках работает в соответствии с условной формой если/тогда.

3.4. Унификация и поиск с возвратом

Эта глава имеет четыре основных раздела. В первом детально рассматривается процесс, который использует Visual Prolog во время попытки сопоставления вызова (из подцели) с предложением (*clause*). Этот процесс поиска включает связывание определенного вызова с конкретным предложением – то, что называется унификацией (unification). В Прологе унификация реализует процедуры, которые Вам, возможно, известны из других, более традиционных языков – это такие процедуры, как передача параметра, выбор варианта, создание структуры, доступ к структуре, присваивание.

Во втором разделе показано, как Visual Prolog производит поиск решений целевого утверждения (при помощи поиска с возвратом), и как он управляет поиском. Управление поиском включает в себя методы,

позволяющие программе выполнить задание, которое было бы невозможно реализовать иным способом, либо вследствие того, что поиск продолжался бы слишком долго, либо из-за того, что система истощит запас свободной памяти.

В третьем разделе этой главы определяется предикат, который можно использовать для поддержки поиска с возвратом, и рассказывается более подробно о том, как управлять поиском с возвратом. Также вводится предикат, который можно использовать для проверки того, удовлетворено ли то или иное ограничение в программе.

С целью лучшего освещения предмета, в четвертом разделе повторяется наиболее важный учебный материал, представленный ранее, но уже с «процедурной» точки зрения. Показано, как можно достичь понимания основных аспектов Пролога – чисто описательного языка – если посмотреть на них как на процедуры.

Сопоставление и унификация

Рассмотрим программу (листинг 3.4.11) с точки зрения того, как утилита ***Test Goal*** будет отыскивать все решения следующей цели:

written_by(X, Y).

Пытаясь выполнить целевое утверждение written_by(x, Y), Visual Prolog должен проверить каждое предложение writtenby в программе. Сопоставляя аргументы X и Y с аргументами каждого предложения written_by, Visual Prolog выполняет поиск от начала программы до ее конца. Обнаружив предложение, соответствующее целевому утверждению, Visual Prolog присваивает значения свободным переменным таким образом, что целевое утверждение и предложение становятся идентичными; говорят, что целевое утверждение унифицируется с предложением. Такая операция сопоставления называется унификацией.

Листинг 3.4.1.1

```
domains
  title,author      = symbol
  pages             = unsigned

predicates
  book(title,pages) – nondeterm (i,o)
  written_by(author,title) – nondeterm (o,o)
  long_novel(title) – nondeterm (o)

clauses
  written_by(fleming, «DR NO»).
  written_by(melville, «MOBY DICK»).
```

```
  book(«MOBY DICK»,250).
  book(«DR NO»,310).

  long_novel(Title):-
    written_by(_,Title),
    book(Title,Length),
    Length > 300.

goal
  long_novel(X).
```

Поскольку X и Y являются свободными переменными в целевом утверждении, а свободная переменная может быть унифицирована с любым другим аргументом (и даже с другой свободной переменной), то целевое утверждение может быть унифицировано с первым предложением *written_by* в программе, как показано ниже:

written_by(X,Y).

written_by(fleming, «DR NO»).

Visual Prolog устанавливает соответствие, X становится связанным с fleming, а Y – с DR NO. В этот момент Visual Prolog напечатает

X=fleming, Y=«DR NO».

Поскольку Test Goal ищет все решения для заданной цели, целевое утверждение также будет унифицировано и со вторым предложением written_by:

Written_sy(melville,«MOBY DICK»). Test Goal печатает второе решение:

X=melville, Y=«MOBY DICK» 2 Solutions.

Теперь предположим, что Вы задали программе целевое утверждение

written_by(X, «MOBY DICK»).

Visual Prolog произведет сопоставление с первым предложением written_by:

written_by(X, «MOBY DICK»).

written_by(fleming, «DR NO»).

Так как «moby dick» и «dr no» не соответствуют друг другу, попытка унификации завершается неудачно. Затем Visual Prolog проверит следующий факт в программе:

written_by(melville, «MOBY DICK»).

Этот факт действительно унифицируется, и X становится связанным с melville.

Рассмотрим, как Visual Prolog выполнит целевое утверждение:

long_novel(X).

Когда Visual Prolog пытается выполнить целевое утверждение, он проверяет, действительно ли обращение может соответствовать факту или заголовку правила. В нашем случае устанавливается соответствие с

long_novel(Title).

Visual Prolog проверяет предложение для *long_novel*, пытаясь завершить сопоставление унификацией аргументов. Поскольку в целевом утверждении X – свободная переменная, то она может быть унифицирована с любым другим аргументом. *Title* также не является связанным в заголовке предложения long_novel. Целевое утверждение соответствует заголовку правила, и унификация выполняется. Впоследствии Visual Prolog будет пытаться согласовывать подцели с правилом.

long_novel(Title):-

written_by(_, Title), book(Title, Length), Length>300.

Пытаясь выполнить согласование тела правила, Visual Prolog обратится к первой подцели в теле правила – *written_by(_, Title)*. Поскольку авторство книги является несущественным, на месте аргумента author появляется анонимная переменная (_). Обращение *written_by(_, Title)* становится текущей подцелью, и Пролог ищет решение для этого обращения.

Пролог ищет соответствие с данной подцелью от вершины и до конца программы. В результате достигается унификация с первым фактом для *written_by*, а именно:

written_by(_, Title),

written_by(fleming, «DR NO»).

Переменная *Title* связывается с *«DR NO»*, и к следующей подцели *book(Title, Length)* обращение выполняется уже с этим значением переменной.

Далее Visual Prolog начинает очередной процесс поиска, пытаясь найти соответствие с обращением к *book*. Так как *Title* связан с *«DR NO»*, фактическое обращение выглядит как *book(«DR No», Length)*. Процесс поиска опять начинается с вершины программы. Заметим, что первая попытка сопоставления с предложением *book(«MOBY DICK», 250)* завершится неудачно, и Visual Prolog перейдет ко второму предложению *book* в поиске соответствия. Здесь заголовок книги соответствует подцели, и Visual Prolog связывает переменную *Length* с величиной *310*.

Теперь третье предложение в теле *long_novel* становится текущей подцелью:

Length > 300.

Visual Prolog выполняет сравнение, завершающееся успешно: *310* больше, чем *300*. В этот момент все подцели в теле правила выполнены, и, следовательно, обращение *long_novel(X)* успешно. Так как X в обращении был унифицирован с переменной *Title* в правиле, то значение, с которым связывается *Title* при подтверждении правила, возвращается и унифицируется с переменной X. Переменная *Title* в случае подтверждения правила имеет значение *«DR NO»*, поэтому Visual Prolog выведет:

X=«DR NO» 1 Solution.

Поиск с возвратом

Часто при решении реальной задачи мы придерживаемся определенного пути для ее логического завершения. Если полученный результат не дает искомого ответа, мы должны выбрать другой путь. Один из верных способов найти конец лабиринта – это поворачивать налево на каждой развилке лабиринта до тех пор, пока Вы не попадете в тупик. Тогда следует вернуться к последней развилке и попробовать свернуть вправо, после чего опять поворачивать налево на каждом встречающемся распутье. Путем методичного перебора всех возможных путей Вы, в конце концов, найдете выход.

Visual Prolog при поиске решения задачи использует именно такой метод проб и возвращений назад; этот метод называется поиск с возвратом. Если, начиная поиск решения задачи (или целевого утверждения), Visual Prolog должен выбрать между альтернативными путями, то он ставит маркер у места ветвления (называемого точкой отката) и выбирает первую подцель, которую и станет проверять. Если данная подцель не выполнится (что эквивалентно достижению тупика в лабиринте), то Visual Prolog вернется к точке отката и попробует проверить другую подцель.

Листинг 3.4.2.1

```
predicates
  likes(symbol,symbol) – nondeterm (i,o)
  tastes(symbol,symbol) – nondeterm (i,i)
  food(symbol) – nondeterm (o)
clauses
  likes(bill,X):-
     food(X),
     tastes(X,good).
  tastes(pizza,good).
  tastes(brussels_sprouts,bad).
  food(brussels_sprouts).
  food(pizza).
goal
  likes(bill,What).
```

Эта маленькая программа составлена из двух множеств фактов и одного правила. Правило, представленное отношением *likes*, утверждает, что Билл любит вкусную пищу.

Чтобы увидеть, как работает поиск с возвратом, дадим программе для решения следующее целевое утверждение:

likes(bill, What).

Когда Пролог пытается произвести согласование целевого утверждения, он начинает поиск с вершины программы.

В данном случае Пролог будет искать решение, производя с вершины программы поиск соответствия с подцелью *likes (bill, what)*.

Он обнаруживает соответствие с первым предложением в программе, и переменная *What* унифицируется с переменной X. Сопоставление с заголовком правила заставляет Visual Prolog попытаться удовлетворить это правило. Производя это, он двигается по телу правила и обращается к первой находящейся здесь подцели: *food(X)*.

Если выполняется новое обращение, поиск соответствия для этого обращения вновь начинается с вершины программы.

Пытаясь согласовать первую подцель, Visual Prolog (начиная с вершины) производит сопоставление с каждым фактом или заголовком правила, встреченным в программе.

Он обнаруживает соответствие с запросом у первого же факта, представляющего отношение *food.* Таким образом, переменная X связывается со значением *brussels_sprouts*. Поскольку существует более чем один возможный ответ на обращение *food(X)*, Visual Prolog ставит **точку возврата** (маркер) возле факта *food(brussels_sprouts)*. Эта точка поиска с возвратом указывает на то место, откуда Пролог начнет поиск очередного возможного соответствия для *food(X)*.

Когда установление соответствия обращения завершается успешно, говорят, что обращение возвращается, и может быть испытана очередная подцель. Поскольку переменная X связана с *brussels_sprouts*, следующее обращение будет выполняться так:

tastes(brussels_sprouts, good)

и Visual Prolog вновь начнет поиск с вершины программы, пытаясь согласовать это обращение. Поскольку соответствующих предложений не обнаруживается, обращение завершается неудачно, и теперь Visual Prolog запускает механизм возврата. Начиная поиск с возвратом, Пролог отступает к последней позиции, где была поставлена точка отката. В данном случае Пролог возвращается к факту *food(brussels_sprouts)*.

Единственным способом освободить переменную, однажды связанную в предложении, является откат при поиске с возвратом.

Когда Пролог отступает к точке поиска с возвратом, он освобождает все переменные, связанные после этой точки, и будет искать другое решение для исходного обращения.

Обращение было *food(X)*, так что связанность *brussels_sprouts* с X отменена. Теперь Пролог пытается заново произвести решение для этого обращения. Он обнаруживает соответствие с фактом *food(pizza);* на этот раз переменная X связывается со значением *pizza*.

Пролог переходит к следующей подцели в правиле, имея при этом новую связанную переменную. Производится новое обращение, *tastes(pizza,*

good), и начинается поиск (опять от вершины программы). На этот раз соответствие найдено, и целевое утверждение успешно выполняется.

Поскольку переменная *What* в целевом утверждении унифицирована с переменной X в правиле *likes*, а переменная X связана со значением *pizza*, переменная *What* отныне связана со значением *pizza* и Visual Prolog сообщает решение:

What=pizza 1 Solution.

После разбора предыдущего простого примера рассмотрим теперь более подробно, как в Visual Prolog работает механизм поиска с возвратом. Начнем с того, что исследуем программу (Листинг 3.4.2.2).

Листинг 3.4.2.2

```
domains
  name,thing = symbol

predicates
  likes(name,thing) – determ (o,i)
  reads(name) – procedure (o)
  is_inquisitive(name) – determ (i)

clauses
  likes(john,wine).
  likes(lance,skiing).
  likes(lance,books).
  likes(lance,films).
  likes(Z,books):-
     reads(Z),
     is_inquisitive(Z).

  reads(john).

  is_inquisitive(john).

goal
  likes(X,wine), like(X,books).
```

Рис. 3.4. Целевое дерево процесса поиска

В свете целевого утверждения, состоящего из двух подцелей:

likes(X, wine), likes(X, books).

При исследовании целевого утверждения Visual Prolog, в первую очередь, отмечает, какие подцели согласовались, а какие нет. Процесс поиска может быть представлен целевым деревом (рис. 8.4).

Перед началом исследования целевого утверждения целевое дерево состоит из двух несогласованных подцелей. На последующих изображениях целевого дерева согласованная подцель в целевом дереве будет отмечаться подчеркиванием, а соответствующее предложение – записываться под этой подцелью.

Четыре основных правила поиска с возвратом

Правило 1

Подцели должны быть согласованы по порядку, сверху вниз.

В изучаемом примере целевое дерево демонстрирует, что должны быть согласованы две подцели. Visual Prolog определяет, какую подцель ему использовать при попытке сопоставления предложения, исходя из второго основного правила поиска с возвратом.

Правило 2

Предикатные предложения проверяются в том порядке, в каком они появляются в программе, сверху вниз.

Выполняя программу, Visual Prolog находит предложение, соответствующее первому факту, определяющему предикат *likes*. Посмотрите на рис. 8.5, как теперь выглядит целевое дерево.

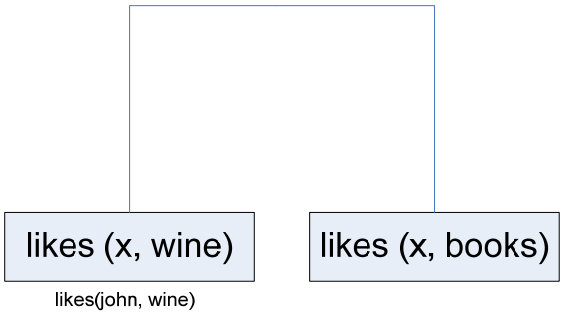

Рис. 3.5. Целевое дерево 2

Подцель *likes(X, wine)* соответствует факту *likes(John, wine)*, и X связывается со значением *john*. Visual Prolog пытается согласовать следующую справа подцель.

Обращение ко второй подцели начинает совершенно новый поиск с условием:

X=john.

Первое предложение

likes(John, wine).

не соответствует подцели

likes(X, books).

т. к. *wine* (вино) – это вовсе не то же самое, что *books* (книги). Поэтому Visual Prolog должен проверить следующее предложение, но *lance* не соответствует значению *X* (потому что в данном случае *X* связан с *john*), так что поиск продолжается с третьим предложением, определяющим предикат *likes*:

likes(Z, books):- reads(Z), is_inquisitive(Z).

Аргумент Z – переменная, поэтому она может соответствовать X. Вторые аргументы находятся в согласии, так что вызов соответствует заголовку правила. Когда X согласуется с Z, аргументы унифицируются. В результате унификации аргументов Visual Prolog приравняет значение, которое имеет X (т. е. *john*) и переменную Z. В результате переменная Z теперь также имеет значение *john*.

Теперь подцель соответствует левой части (заголовку) правила. Продолжение поиска определяется третьим фундаментальным правилом поиска с возвратом:

Правило 3

Когда подцель соответствует заголовку правила, далее должно быть согласовано тело этого правила: тело правила теперь образует новое множество подцелей для согласования.

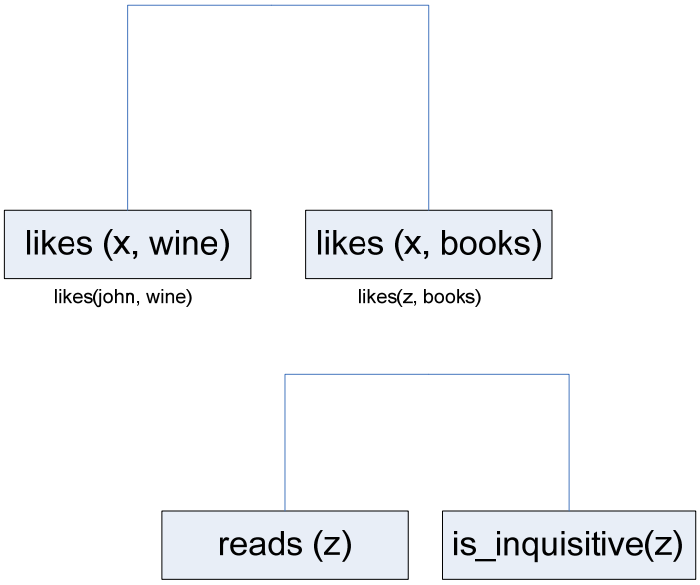

Рис. 3.6. Целевое дерево 3

Теперь целевое дерево включает в себя подцели

reads(Z) and is_inquisitive(Z).

Целевое дерево станет таким, как на рис. 8.6, где Z связана со значением *john*. Далее Visual Prolog будет искать факты, соответствующие обеим подцелям. Последнее результирующее целевое дерево изображено на рис. 8.7.

Правило 4

Целевое утверждение считается согласованным, когда соответствующий факт найден для каждой оконечности (листа) целевого дерева.

Таким образом, теперь начальное целевое утверждение согласовано.

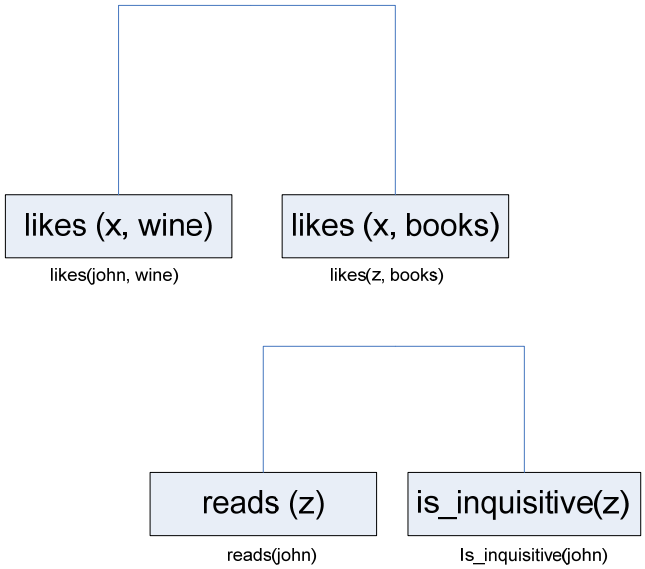

Рис. 3.7. Результирующее целевое дерево

Visual Prolog использует результат процедуры поиска по-разному, в зависимости от того, как был начат поиск. Если целевое утверждение является обращением из подцели в теле правила, то Visual Prolog пытается согласовать следующую подцель в правиле, после того как обращение возвращено. Если целевое утверждение является вопросом пользователя, то Visual Prolog непосредственно отвечает:

X=john

1 Solution.

Как вы видели в программе **Листинг 3.4.2.2**, однажды согласовав целевое утверждение, Test Goal возвращается (откатывается) назад для поиска всех альтернативных решений. Test Goal возвращается назад и в том случае, если подцель не выполняется, надеясь пересогласовать предыдущую подцель с другими предложениями.

Выполняя подцель, Visual Prolog начинает поиск с первого предложения, определяющего предикат. Затем может произойти одно из двух:

1. Visual Prolog находит соответствующее предложение, тогда:
   - если имеется другое предложение, которое, возможно, может вновь согласовать подцель, Visual Prolog выставляет указатель (с тем, чтобы отметить точку возврата) и связывает все свободные переменные в подцели (которые соответствуют значениям в предложении) с соответствующими значениями;

- если данное предложение является заголовком правила, то затем оценивается тело этого правила. Подцели в теле правила должны быть удовлетворены для успешного завершения обращения.

2. Visual Prolog не может найти соответствующее предложение. Целевое утверждение не согласуется и Visual Prolog выполняет поиск с возвратом в попытке вновь согласовать предыдущую подцель. Когда процесс достигает последней точки возврата, Visual Prolog освобождает все переменные, которым были присвоены новые значения (после того, как была поставлена точка возврата), и вновь пытается согласовать исходное обращение.
3. Visual Prolog начинает поиск с вершины программы. Когда он выполняет возврат к обращению, новый процесс поиска начинается с точки отката, выставленной последней. Если поиск безуспешен, то вновь выполняется поиск с возвратом. Если процесс поиска с возвратом исчерпал все предложения для всех подцелей, то это означает, что целевое утверждение не согласуется.

Поиск с возвратом для внутреннего целевого утверждения

Приведем еще один, усложненный пример, иллюстрирующий, как в Visual Prolog происходит поиск с возвратом, когда программа скомпилирована и выполняется как автономная исполняемая программа

```
predicates
  type(symbol,symbol) – nondeterm (o,i)
  is_a(symbol,symbol) – nondeterm (o,i)
  lives(symbol,symbol) – nondeterm (i,i)
  can_swim(symbol) – nondeterm (o)

clauses
  type(ungulate,animal).
  type(fish,animal).
  is_a(zebra,ungulate).
  is_a(herring,fish).
  is_a(shark,fish).
  lives(zebra,on_land).
  lives(frog,on_land).
  lives(frog,in_water).
  lives(shark,in_water).
  can_swim(Y):-
     type(X,animal),
     is_a(Y,X),
     lives(Y,in_water).
```

```
goal
  can_swim(What),
  write("A ",What," can swim\n").
```

После того как программа скомпилирована и запущена, Visual Prolog автоматически начнет выполнение целевого утверждения, пытаясь согласовать все подцели в разделе программы *goal*.

Visual Prolog обращается к предикату *can_swim* со свободной переменной *What*. Пытаясь выполнить это обращение, Пролог просматривает программу в поисках соответствия. Он обнаруживает соответствие с предложением, определяющим *can_swim*, и переменная *What* унифицируется с переменной Y.

Затем Visual Prolog пытается согласовать тело правила. При этом происходит обращение к первой подцели в теле правила, *type(X, animal)*, и поиск соответствия для этого обращения. Он обнаруживает соответствие с первым фактом, определяющим отношение *type*.

В этот момент X связывается с *ungulate*. Поскольку здесь налицо более чем одно возможное решение, Пролог проставляет точку возврата возле факта

type(ungulate, animal).

Имея X, связанным с *ungulate*, Visual Prolog производит обращение ко второй подцели в правиле *(is_a(Y, ungulate))* и снова ищет соответствие. Он находит его с первым фактом, *is_a(zebra, ungulate)*. Y связывается с *zebra*, и Пролог выставляет точку возврата возле факта *is_a (zebra, ungulate)*.

Теперь, имея X, связанным с *ungulate*, и Y – с *zebra*, Visual Prolog пытается согласовать последнюю подцель, *lives(zebra, in_water)*. Пролог проверяет каждое предложение lives, но в программе нет предложения *lives(zebra,in_water)*, поэтому обращение завершается неудачно, и далее начинается поиск с возвратом другого решения.

Когда Visual Prolog совершает обратный ход, процесс возвращается к последней позиции, где была помещена точка возврата. В данном случае последняя точка возврата была поставлена у второй подцели в правиле, на факте *is_a(zebra, ungulate)*.

При достижении точки возврата Visual Prolog освобождает переменные, которым были присвоены новые значения после последней точки возврата, и пытается найти другое решение для обрабатываемого обращения. В данном случае обращение было *is_a(Y, ungulate)*.

Visual Prolog продолжает спуск по предложениям в поиске другого соответствующего предложения, начиная с того места, где поиск был прекращен. Так как в программе больше нет соответствующих предложений, обращение завершается неудачно, и Пролог вновь ведет поиск с возвратом в попытке решить исходное целевое утверждение.

Теперь в качестве последней точки возврата рассматривается *type(ungulate, animal)*.

Visual Prolog освобождает переменные, использованные в исходном обращении, и пытается найти другое решение для обращения *type(X, animal)*. Поиск начинается после точки возврата. Пролог находит соответствие со следующим фактом *type* в программе *(type(fish, animal))*; **X** связывается с *fish*, и новая точка возврата ставится возле этого факта.

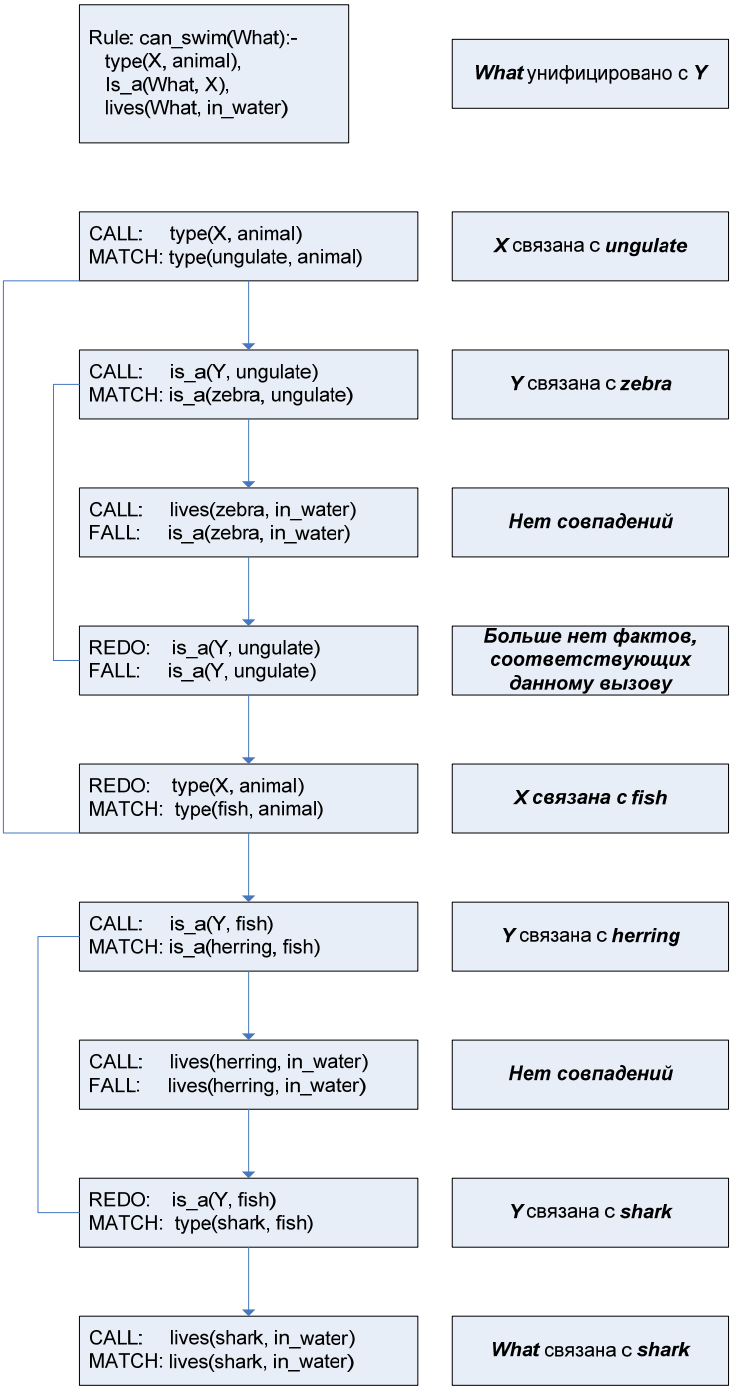

Рис. 3.8. Алгоритм работы программы can_swim

Далее Visual Prolog продвигается вниз, к очередной подцели в правиле; поскольку это уже новое обращение, поиск начинается с вершины программы для

is_a(Y, fish).

Visual Prolog находит соответствие для этого обращения, и **Y** становится связанным с *herring.*

Так как **Y** теперь связан с *herring*, следующая подцель, к которой происходит обращение, суть *lives(herring, inwater)*. Это новое обращение – поиск начинается с вершины программы.

Пролог исследует каждый факт *lives*, но не находит соответствия, и подцель не выполняется.

Visual Prolog возвращается к последней точке возврата *is_a (herring, fish)*.

Переменные, которые были связаны этим сопоставлением, теперь освобождены. Начиная с того места, где процесс был прекращен, Пролог теперь ищет новое решение для обращения is_a (**Y**, fish).

Visual Prolog находит соответствие со следующим предложением, **is_a**, **Y** становится связанным с идентификатором shark.

Пролог опять исследует последнюю подцель, имея переменную **Y**, связанную с shark. Он выполняет обращение lives (shark, inwater); поиск начинается с вершины программы (новое обращение) и обнаруживает соответствие. Последняя для правила подцель выполняется.

К этому моменту тело правила *can_swim(Y)* согласовано. Visual Prolog возвращает **Y** вызову *can_swim(What)*. Так как *What* связана с Y, а **Y** – с *shark*, то в целевом утверждении *What* связывается с *shark.*

Пролог продолжает процесс с того места в разделе **goal**, где он был остановлен, и обращается ко второй подцели в целевом утверждении.

Visual Prolog завершает программу выводом:

A shark can swim.

Управление поиском решений

Встроенный механизм поиска с возвратом в Прологе может привести к поиску ненужных решений, в результате чего теряется эффективность, например, когда желательно найти только одно решение. В других случаях может оказаться необходимым продолжать поиск дополнительных решений, даже если целевое утверждение уже согласовано. В этом разделе показаны некоторые методы, которые можно использовать для управления поиском решений ваших целевых утверждений.

Visual Prolog обеспечивает два инструментальных средства, которые дают возможность управлять механизмом поиска с возвратом: предикат ***fail***, который используется для инициализации поиска с возвратом, и ***cut*** или отсечение (обозначается «**!**») – для запрета возможности возврата.

Использование предиката fail

В определенных ситуациях бывает необходимо инициализировать выполнение поиска с возвратом, чтобы найти другие решения. Visual Prolog поддерживает специальный предикат *fail*, вызывающий неуспешное завершение, и, следовательно, инициализирует возврат. Действие предиката *fail* равносильно эффекту от сравнения *2=3* или другой невозможной подцели. Программа иллюстрирует использование этого специального предиката.

```
domains
  name = symbol
predicates
  father(name,name) – nondeterm (o,o)
  everybody – procedure ()
clauses
  father(leonard,katherine).
  father(carl,jason).
  father(carl,marilyn).
  everybody:-
     father(X,Y),
     write(X," is ",Y,"'s father\n"),
     fail.
  everybody.
goal
  everybody.
```

Test Goal найдет все решения цели *father(X,Y)* и отобразит значения всех переменных:

X=leonard, Y=katherine X=carl, Y=j ason X=carl, Y=marilyn 3 Solutions.

Но если вы скомпилируете эту программу и запустите ее (клавишей <F9> или командой меню Project|Run), то Visual Prolog найдет только первое подходящее решение для *father(X,Y)*. После того как целевое утверждение, определенное в разделе **goal**, выполнено впервые, ничто не говорит Прологу о необходимости продолжения поиска с возвратом. Поэтому обращение к *father* приведет только к одному решению. Как же найти все возможные решения? Предикат *everybody* в программе использует fail для поддержки поиска с возвратом.

Задача предиката *everybody* – найти все решения для *father* и выдать полный ответ. Сравните предыдущие ответы утилиты Test Goal с целью *father(X,Y)* и ответы на выполнение очередной цели:

```
goal
everybody.
```

отображенные сгенерированной программой:

leonard is katherine's father carl is jason's father carl is marilyn's father.

Предикат everybody использует поиск с возвратом с тем, чтобы получить все решения для *father(X, Y)*, заставляя Пролог выполнять поиск с возвратом сквозь тело правила

```
everybody:-
father(X, Y),
write (X, " is ", Y, " ' s father\n"),
fail.
```

Fail не может быть согласован (он всегда неуспешен), поэтому Visual Prolog вынужден повторять поиск с возвратом. При поиске с возвратом он возвращается к последнему обращению, которое может произвести множественные решения. Такое обращение называют недетерминированным. Недетерминированное обращение является противоположностью детерминированному обращению, которое может произвести только одно решение.

Предикат *write* не может быть вновь согласован (он не может предложить новых решений), поэтому Visual Prolog должен выполнить откат дальше, на этот раз к первой подцели в правиле.

Обратите внимание, что помещать подцель после *fail* в теле правила бесполезно. Предикат *fail* все время завершается неудачно, нет возможности для достижения подцели, расположенной после fail.

Прерывание поиска с возвратом: отсечение

Visual Prolog предусматривает возможность отсечения, которая используется для прерывания поиска с возвратом; отсечение обозначается восклицательным знаком !. Действует отсечение просто: через него невозможно совершить откат (поиск с возвратом).

Отсечение помещается в программу таким же образом, как и подцель в теле правила. Когда процесс проходит через отсечение, немедленно удовлетворяется обращение к **cut** и выполняется обращение к очередной подцели (если таковая имеется). Однажды пройдя через отсечение, уже невозможно произвести откат к подцелям, расположенным в обрабатываемом предложении перед отсечением, и также невозможно возвратиться к другим предложениям, определяющим обрабатывающий предикат (предикат, содержащий отсечение).

Существуют два основных случая применения отсечения.

- Если Вы заранее знаете, что определенные посылки никогда не приведут к осмысленным решениям (поиск решений в этом случае будет лишней тратой времени) – примените отсечение – программа станет быстрее и экономичнее. Такой прием называют зеленым отсечением.
- Если отсечения требует сама логика программы для исключения из рассмотрения альтернативных подцелей. Это – красное отсечение.

Использование отсечений

В этом разделе даются примеры, показывающие, как следует использовать отсечение, рассматриваются несколько условных правил. (rl, r2 и r3), которые определяют условный предикат r, а также несколько подцелей – a, b, c и т. д.

Предотвращение поиска с возвратом к предыдущей подцели в правиле *r1:- a,b,!,c.*

Такая запись является способом сообщить Visual Prolog о том, что Вас удовлетворит первое решение, найденное им для подцелей **a** и **b**. Имея возможность найти множественные решения при обращении к **c** путем поиска с возвратом, Пролог при этом не может произвести откат (поиск с возвратом) через отсечение и найти альтернативное решение для обращений **a** и **b**. Он также не может возвратиться к другому предложению, определяющему предикат **rl**.

В качестве конкретного примера рассмотрим программу:

```
predicates
  buy_car(symbol,symbol) – determ (i,o)
  car(symbol,symbol,integer) – nondeterm (i,o,o)
  colors(symbol,symbol) – nondeterm (i,i)

clauses
  buy_car(Model,Color):-
     car(Model,Color,Price),
     colors(Color,sexy),!,
     Price > 25000.

  car(maserati,green,25000).
  car(corvette,black,24000).
  car(corvette,red,26000).
  car(porsche,red,24000).
  colors(red,sexy).
  colors(black,mean).
  colors(green,preppy).

goal
  buy_car(corvette,Y).
```

В данном примере поставлена цель: найти *Corvette(Корвет)* приятного цвета, подходящий по стоимости. Отсечение в правиле *buycar* означает, что поскольку в базе данных содержится только один «Корвет» приятного цвета, хоть и со слишком высокой ценой, то нет нужды искать другую машину.

Получив целевое утверждение

buy_car(corvette, Y).

программа отработает следующие шаги:

Visual Prolog обращается к *car*, первой подцели для предиката *buycar*.

Выполняет проверку для первой машины, *Maserati*, которая завершается неудачно.

Затем проверяет следующее предложение *car* и находит соответствие, связывая переменную *Color* со значением *black*.

Переходит к следующему обращению и проверяет, имеет ли выбранная машина приятный цвет. Черный цвет не является приятным в данной программе, таким образом, проверка завершается неудачно.

Выполняет поиск с возвратом к обращению *car* и снова ищет *Corvette*, удовлетворяющий этому критерию.

Находит соответствие, и снова проверяет цвет. На этот раз цвет оказывается приятным, и Visual Prolog переходит к следующей подцели в правиле: к отсечению. Отсечение немедленно выполняется, «замораживая» все переменные, ранее связанные в этом предложении.

Переходит к следующей (и последней) подцели в правиле, к сравнению *Price < 25000.*

Проверка завершается неудачно, и Visual Prolog пытается совершить поиск с возвратом с целью найти другую машину для проверки. Отсечение предотвращает попытку решить последнюю подцель, и наше целевое утверждение завершается неудачно.

Предотвращение поиска с возвратом к следующему предложению

Отсечение может быть использовано, как способ сообщить Visual Prolog, что он выбрал верное предложение для определенного предиката. Например, рассмотрим следующий фрагмент:

```
r(1) :-
!, a, b, c.
r(2) :-
!,d.
r(3) :-
!,c.
r(_):-
write(«This is a catchall clause.»).
```

Использование отсечения делает предикат ***r*** детерминированным. В данном случае Visual Prolog выполняет обращение к ***r*** с единственным целым аргументом. Предположим, что произведено обращение ***r(1)***. Visual Prolog просматривает программу в поисках соответствия для обращения; он находит его с первым предложением, определяющим ***r***. Поскольку

имеется более чем одно возможное решение для данного обращения, Visual Prolog проставляет точку возврата около этого предложения.

Теперь Visual Prolog начинает обработку тела правила, проходит через отсечение и исключает возможность возвращения к другому предложению ***r***. Это отменяет точки поиска с возвратом, повышая эффективность выполнения программы, а также гарантирует, что отлавливающее ошибки предложение будет выполнено лишь в том случае, если ни одно из условий не будет соответствовать обращению к ***r***.

Обратите внимание, что конструкция такого типа весьма похожа на конструкцию *case* в других языках программирования; условие проверки записывается в заголовке правил. Вы могли бы написать такие предложения:

```
r(X):-
X = 1,!,a, b, c.
r(X):-
X = 2,!,d.
r(X):-
X = 3,!,c
r(_):-
write(«This is a catchall clause.»).
```

Следует, по возможности, помещать проверочное условие именно в заголовок правила – это повышает эффективность программы и упрощает ее чтение.

В качестве другого примера рассмотрим программу:

```
predicates
  friend(symbol,symbol) – nondeterm (i,o)
  girl(symbol) – nondeterm (i)
  likes(symbol,symbol) – nondeterm (i,i)
clauses
  friend(bill,jane):-
      girl(jane),
      likes(bill,jane),!.
  friend(bill,jim):-
      likes(jim,baseball),!.
  friend(bill,sue):-
      girl(sue).
  girl(mary).
  girl(jane).
  girl(sue).
```

```
  likes(jim,baseball).
  likes(bill,sue).
goal
  friend(bill,Who).
```

Если бы в программе не было отсечения, то Visual Prolog предложил бы два решения: Билл является другом как Джейн, так и Сью. Отсечение в первом предложений, определяющим *friend*, говорит о том, что если это предложение согласовано, то друг Билла уже найден, и нет нужды продолжать поиск других кандидатур. Поиск с возвратом может иметь место внутри предложений в попытке согласовать обращение, но, однажды обнаружив решение, Visual Prolog проходит через отсечение. Предложения *friend*, записанные так, как показано выше, возвратят одного и только одного друга Билла (если друг вообще может быть найден).

Детерминизм и отсечение

Если бы предикат *friend*, определенный в предыдущей программе, не содержал отсечений, то это был бы недетерминированный предикат (способный производить множественные решения при помощи поиска с возвратом). В предыдущих реализациях Пролога программисты должны были обращать особое внимание на недетерминированные предложения из-за сопутствующих им дополнительных требований к ресурсам памяти. Теперь Visual Prolog сам выполняет проверку на недетерминированные предложения, облегчая вашу работу.

В Прологе существует директива компилятора *check_determ*. Если вставить эту директиву в самое начало программы, то Visual Prolog будет выдавать предупреждение в случае обнаружения недетерминированных предложений в процессе компиляции.

Вы можете превратить недетерминированные предложения в детерминированные, вставляя отсечения в тело правил, определяющих данный предикат. Например, помещение отсечений в предложения, определяющие предикат *friend*, делает этот предикат детерминированным, поскольку в данном случае обращение к *friend* может возвратить одно и только одно решение.

Поиск всех решений в Test Goal

Как было описано выше, при помощи поиска с возвратом Visual Prolog не только найдет первое решение задачи, но при использовании режима Test Goal будет также способен найти все возможные решения.

Рассмотрим программу, которая содержит сведения об именах и возрастах нескольких игроков в теннисном клубе.

```
domains
  child = symbol
  age = integer
predicates
  player(child,age) – nondeterm (o,i), nondeterm (i,i)

clauses
  player(peter,9).
  player(paul,10).
  player(chris,9).
  player(susan,9).

goal
  player(Person1,9),
  player(Person2,9),
  Person1 <> Person2.
```

Спланируем турнир по пинг-понгу между девятилетними членами теннисного клуба. Каждая пара игроков должна провести между собой две игры. Задача – найти все возможные пары из тех игроков клуба, кому по девять лет. Она может быть решена заданием режиму Test Goal составной цели:

goal

player(Person1, 9), player(Person2, 9), Person1 <> Person2.

На естественном языке это прозвучало бы так: найти *Person1* в возрасте 9 лет и *Person2* в возрасте 9 лет, отличное от *Person1*.

Что будет делать Visual Prolog?

1. Попытается найти решение для первой подцели *player(Personl, 9)* и перейдет к следующей подцели только после того, как первая будет достигнута. Первая подцель согласуется сопоставлением *Personl* с *peter*.

Затем Visual Prolog попытается согласовать очередную подцель:

player(Person2, 9).

Аналогично, *Person2* сопоставляется с *peter*.

Теперь Visual Prolog переходит к третьей и последней подцели:

Person1 <> Person2.

2. Так как и *Person1* и *Person2* связаны с *peter*, эта подцель не выполняется. Вследствие этого Visual Prolog выполняет поиск с возвратом к предыдущей подцели и ищет другое решение для второй подцели:

player(Person2, 9).

Эта подцель выполняется при сопоставлении *Person2* с *chris*.

3. Теперь третья подцель:

Person1 <> Person2.

Она может быть выполнена, т. к. *peter* и *chris* отличны. Таким образом, целевое утверждение полностью согласовано путем образования турнирной пары из Криса и Питера.

4. Поскольку Test Goal должен найти все возможные решения целевого утверждения, он находит точку поиска с возвратом предыдущей цели в надежде вновь добиться успеха.

Так как *player(Person2,9)* также может быть согласовано, если принять *Person2* за *susan*, то Visual Prolog еще раз проверит третью подцель. Достигается успех (т. к. *peter* отличен от *susan*), и другое решение для всего целевого утверждения найдено.

В дальнейшем поиске решений Test Goal вновь возвращается к точке поиска с возвратом второй подцели, но все возможности для этой подцели уже исчерпаны, и поиск с возвратом теперь выполняется от первой подцели. Она вновь может быть согласована сопоставлением Personl с *chris*. Вторая подцель теперь имеет успех в результате сопоставления *Person2* с *peter*, так что третья подцель согласована, и все целевое утверждение опять выполнено. Итак, запланирована еще одна встреча, на этот раз между Крисом и Питером.

В поисках еще одного решения целевого утверждения Test Goal возвращается к точке поиска с возвратом второй подцели в правиле. Здесь *Person2* ставится в соответствие *chris*, и при этом условии проверяется третья подцель. Она не выполняется, т. к. *Person1* и *Person2* эквивалентны. Тогда Test Goal выполняет поиск с возвратом от второй подцели в поисках другого решения: *Person2* сопоставляется с *susan*, третья подцель выполнена. Test Goal назначает очередную встречу в теннисном клубе (Крис против Сюзан).

И вновь, памятуя о необходимости найти все решения, Test Goal возвращается ко второй подцели, но на этот раз безуспешно. Когда вторая подцель не выполняется, процесс возвращается к первой подцели, на этот раз, находя соответствие *Person1* с susan. Пытаясь выполнить вторую подцель, Test Goal сопоставляет *Person2* с *peter*, и впоследствии третья подцель выполнится при этих условиях. Итак, назначена пятая встреча.

Снова возврат ко второй подцели, где *Person2* сопоставляется с *chris*. Найдено шестое решение задачи о теннисном клубе и, наконец, получено полное множество турнирных пар.

Последнее исследуемое решение связывает с *susan* как *Person1*, так и *Person2* и приводит к невыполнению последней подцели. Visual Prolog должен вернуться ко второй подцели, но там не осталось никаких новых вариантов. Тогда он возвращается для поиска к первой подцели, но и здесь все возможности для *Person1* уже исчерпаны. Для данного целево-

го утверждения не может быть найдено других решений, и работа программы завершается.

Убедитесь, что Test Goal ответит следующим:

```
Person1=peter,    Person2=chris
Person1=peter,    Person2=susan
Person1=chris,    Person2=peter
Person1=chris,    Person2=susan
Person1=susan,    Person2=peter
Person1=susan,    Person2=chris
6 Solutions
```

Обратите внимание, как поиск с возвратом может вызывать вывод Test Goal избыточных решений. В нашем примере Test Goal не отметил, что Person1 = peter то же самое, что и Person2 = peter. Далее в этой главе мы покажем, как управлять поиском в Visual Prolog.

Предикат not

Следующая программа демонстрирует, как Вы можете использовать предикат *not* для того, чтобы выявить успевающего студента: студента, у которого средний балл (GPA) не менее 3.5, и у которого в настоящее время не продолжается испытательный срок.

```
domains
  name = symbol
  gpa = real

predicates
  honor_student(name) – nondeterm (o)
  student(name,gpa) – nondeterm (o,o)
  probation(name) – nondeterm (i)

clauses
  honor_student(Name):-
     student(Name,GPA),
     GPA>=3.5,
     not(probation(Name)).

  student(«Betty Blue»,3.5).
  student(«David Smith»,2.0).
  student(«John Johnson»,3.7).

  probation(«Betty Blue»).
  probation(«David Smith»).
```

```
goal
  honor_student(X).
```

При использовании предиката *not* необходимо иметь в виду следующее.

Предикат *not* будет успешным, если не может быть доказана истинность данной подцели.

Это приводит к предотвращению связывания внутри *not* несвязанных переменных. При вызове изнутри *not* подцели со свободными переменными, Visual Prolog возвратит сообщение об ошибке: ***«Free variables not allowed in not or retractall»*** (Свободные переменные не разрешены в *not* или *retract*). Это происходит вследствие того, что для связывания свободных переменных в подцели, подцель должна унифицироваться с каким-либо другим предложением и выполняться. Правильным способом управления несвязанными переменными подцели внутри *not* является использование анонимных переменных.

Вот несколько примеров корректных и некорректных предложений:

```
likes(bill, Anyone):-% Anyone – выходной аргумент,
likes(sue, Anyone), not(hates(bill, Anyone).
```

В этом примере *Anyone* связывается посредством *likes(sue, Anyone)* до того, как Visual Prolog делает вывод, что *hates(bill, Anyone)* не является истиной. Данное предложение работает корректно.

Если Вы измените его таким образом, что обращение к *not* будет выполняться первым, то получите сообщение об ошибке: ***«Free variable are not allowed in not»*** (Свободные переменные в *not* не разрешены).

```
likes(bill, Anyone):-% Это не будет работать правильн,.
not(hates(bill, Anyone)), likes(sue, Anyone).
```

Даже если Вы замените в *not(hates (bill, Anyone)) Anyone* на анонимную переменную, и предложение, таким образом, не будет возвращать ошибку, все равно получите неправильный результат.

```
likes(bill, Anyone):-% Это не будет работать правильно,
not(hates(bill, _)), likes(sue, Anyone).
```

Это предложение утверждает, что Биллу нравится кто угодно, если неизвестно ничего о том, кого Билл ненавидит, и если этот «кто-то» нравится Сью. Подлинное предложение утверждало, что Биллу нравится тот, кто нравится Сью, и при этом Билл не испытывает к этому человеку ненависти.

Неверное использование предиката *not* приведет к сообщению об ошибке или к ошибкам в логике вашей программы. Следующая программа является примером правильного использования предиката *not*.

```
predicates
  likes_shopping(symbol) – nondeterm (o)
```

```
  has_credit_card(symbol,symbol) – nondeterm (o,o)
  bottomed_out(symbol,symbol) – nondeterm (i,i)
clauses
  likes_shopping(Who):-
     has_credit_card(Who,Card),
     not(bottomed_out(Who,Card)),
     write(Who,« can shop with the »,Card, « credit card.\n»).

  has_credit_card(chris,visa).
  has_credit_card(chris,diners).
  has_credit_card(joe,shell).
  has_credit_card(sam,mastercard).
  has_credit_card(sam,citibank).

  bottomed_out(chris,diners).
  bottomed_out(sam,mastercard).
  bottomed_out(chris,visa).
goal
  likes_shopping(Who).
```

Пролог с процедурной точки зрения

Пролог – это декларативный язык. Описывая задачу в терминах фактов и правил, Вы предоставляете Visual Prolog самому искать способ решения. Другие языки программирования, такие как Pascal, Basic и C – процедурные. Это означает, что Вы должны писать подпрограммы и функции, которые подробно «объяснят» компьютеру, какие шаги должны быть сделаны для решения задачи.

Сейчас давайте оглянемся назад и рассмотрим некоторые моменты с точки зрения процедурного программирования.

Факты и правила в качестве процедур

Можно рассматривать правила Пролога как определения процедур. Например, правило:

likes(bill,Something):-likes(cindy,Something)

означает:

«Для того чтобы доказать, что Билл любит что-то, необходимо доказать, что Синди любит это».

Таким образом, видим, что предикаты типа:

say_hello:-
write(«Hello»), nl.

и

greet:-

write(«Hello, Earthlings!»), nl.

соответствуют подпрограммам и функциям в других языках программирования. Вы можете рассматривать даже факты Пролога, как процедуры; например, факт *likes(bill,pasta)* означает:

«Для того чтобы доказать, что Билл любит pasta, не нужно ничего делать, и если аргументы *Who* и *What* в вашем запросе *likes(Who,What)* – свободные переменные, то Вы можете присвоить им значения *bill* и *pasta*, соответственно».

Далее мы покажем, как известные процедуры программирования (условное ветвление, булевы выражения, безусловные переходы и возвращение результата вычисления) могут быть реализованы в Прологе.

Использование правил для условного ветвления

Одно из основных различий между правилами в Прологе и процедурами в других языках программирования заключается в том, что Пролог позволяет задавать множество альтернативных определений одной и той же процедуры. Это видно по «семейной» программе. Человек может быть предком, будучи отцом или матерью, поэтому определение предка состоит из двух правил.

Вы можете использовать множество определений так же, как вы применяете предложение case в Pascal, задавая множество пьтернативных определений для каждого значения аргумента (или множества значений аргумента). Пролог же будет перебирать одно правило за другим, пока не найдет то, которое подходит, и затем выполнит действие, заданное правилом.

```
predicates
   action(integer) – nondeterm (i)

clauses
   action(1):-
     nl,
     write(«You typed 1.»),nl.
   action(2):-
     nl,
     write(«You typed two.»),nl.
   action(3):-
     nl,
   write(«Three was what you typed.»),nl.
   action(N):-
     nl,
```

```
    N<>1, N<>2, N<>3,
    write(«I don't know that number!»).

goal
   write(«Type a number from 1 to 3: »),
   readint(Num),
   action(Num).
```

Если пользователь нажмет клавиши <1>, <2> или <3>, *action* будет вызвана с соответствующим значением аргумента и будет вызвано одно из первых трех правил этого примера.

Выполнение проверки в правиле

Посмотрите более внимательно на четвертое правило для *action*. Оно будет сопоставлено для любого аргумента, переданного правилу. Если Вы хотите быть уверенными, что оно не напечатает ***I don't know that number*** (Я не знаю такого числа), когда число попадает в правильный диапазон, – это задача для подцелей X<>1, X<>2, X<>3, где <> обозначает «не равно». Теперь, для того чтобы напечатать «Я не знаю такого числа», Пролог должен сначала доказать, что X не равен 1, 2 или 3. Если какая-либо из этих подцелей неуспешна, то Пролог попытается сделать откат и найти новые альтернативы. Но так как таких альтернатив нет, то остаток предложения никогда не будет выполнен.

Предикат *action* подразумевает, что *Num* уже связана. Если Вы вызываете *action* со свободной переменной в качестве аргумента, то компилятор сгенерирует ошибку.

Отсечение как GoTo

Предыдущая программа не совсем корректна из-за того, что после выбора и выполнения нужного правила, Пролог продолжает поиск альтернатив.

Вы могли бы сэкономить ресурсы и время, если бы указали, где нужно прекратить поиск альтернатив, используя отсечение. Это означает: «Если вы дошли до этого места, то не нужно производить откаты внутри этого правила и не нужно проверять остальные альтернативы этого правила».

Возвращение все еще возможно, но только на более высоком уровне. Если текущее правило вызывается другими правилами, и высшие правила имеют альтернативы, то они могут быть испробованы. Но отсечение отбрасывает альтернативы внутри правила и альтернативы данного правила (предиката).

Используя *cut*, эта программа может быть переписана так:

```
predicates
  action(integer) – procedure (i)
clauses
  action(1):-!,
     nl,
     write(«You typed 1.»).
  action(2):-!,
     nl,
     write(«You typed two.»).
  action(3):-!,
     nl,
     write(«Three was what you typed. »).
  action(_):-nl,
     write(«I don't know that number! »).
goal
  write(«Type a number from 1 to 3: »),
  readint(Num),
  action(Num),nl.
```

Отсечение не имеет никакого эффекта, пока оно не будет выполнено реально. В приведенном выше примере, для того чтобы выполнить отсечение, Пролог должен войти в правило, содержащее отсечение и достичь точки, где расположено отсечение.

Отсечение может быть представлено другими примерами:

```
action(X) :-
X>3,
!,write(«Too high.»).
```

В этом правиле отсечение не произведет никакого действия, пока не будет достигнута первая подцель X>3.

Заметьте, что порядок правил здесь имеет значение. В предыдущей программе Вы могли написать правила в любом порядке; только одно из них сопоставлялось с конкретным числом. Но в данном примере Вы должны быть уверены, что компьютер не сделает попытки выполнить правило, печатающее «Я не знаю такого числа», раньше, чем будут испробованы (и не выполнят своих отсечений) все предыдущие правила.

Отсечения в первой программе иногда называют красными отсечениями, т. к. они меняют логику программы. Если Вы сохраните проверки X<>1, X<>2 и X<>3, изменив программу только вставкой отсечений в каждом предложении, то Вы сделаете зеленые отсечения. Они экономят время, и тем не менее, оставляют программу такой же правильной,

как и без отсечений. Выигрыш при этом не так велик, но риск внести ошибку в программу уменьшается.

Отсечение – это мощный, но и опасный оператор Пролога. В этом отношении он соответствует предложению **GoTo**, в остальных языках программирования он многое позволяет, но делает вашу программу более трудной для понимания.

Возврат вычисленного значения

Как мы уже видели, правила и факты Пролога могут возвращать информацию в цель, которая их вызывает. Это делается путем связывания переменных, которые были ранее не связанными.

Факт

```
likes(bill, cindy). %
likes(bill, Who).
```

возвращает информацию в цель путем присваивания переменной Who значения cindy.

В листинге приведен пример того, как правило может возвращать результат вычислений.

```
predicates
  classify(integer,symbol) – nondeterm (i,i)
clauses
  classify(0,zero).
  classify(X,negative):-
     X < 0.
  classify(X,positive):-
     X > 0.

goal
  classify(45,positive).
```

Первый аргумент *classify* должен всегда получать константу или связанную переменную. Второй аргумент может быть связанной или свободной переменной, он сопоставляется с символами *zero*, *negative*, *positive* в зависимости от значения первого аргумента.

Здесь приведены несколько примеров правил, которые могут возвращать значения. Вы можете узнать, положительно ли число:

Goal classify(45, positive). Yes.

Так как 45 больше 0, только третье предложение *classify* может быть успешным. Оно сопоставляет второй аргумент с *positive*. Но второй аргумент уже равен *positive*, поэтому сопоставление успешно, и Вы получаете ответ *yes (да)*.

Если сопоставление неуспешно, Вы получаете ответ *no (нет)*:

Goal classify(45, negative).

Пролог проверяет первое предложение, но первый аргумент не равен 0 (а также второй не равен *zero*), затем он проверяет второе предложение, связав X с 45, но проверка X<0 неуспешна, после этого он проверяет третье предложение, но на этот раз второй аргумент не совпадает.

Для получения правильного ответа, а не *yes* или *no*, Вы должны вызвать *classify* со свободным вторым аргументом.

Goal classify(45, What).

What=positive

1 Solution

В данном случае цель *classify(45,What)* не сопоставляется с заголовком первого предложения, т. к. 45 не сопоставляется с 0. Первый класс использовать нельзя.

Цель *classify(45,What)* снова сопоставляется с заголовком следующего предложения, *classify(X,negative)*, связывая X с 45 и *negative* с *What*. Но подцель X<0 неуспешна, т. к. X равен 45 и неверно, что 45<0, поэтому Пролог возвращается из этого предложения, освобождая созданные связи; наконец, *classify(45,What)* сопоставляется с *classify(X,positive)*, связывая X и 45, а также *What* и *positive*, подцель X>0 правильна. Так как это успешное решение, Пролог не выполняет поиск с возвратом; он возвращается в вызывающую процедуру (которая в данном случае цель, которую Вы задали). И поскольку переменная X принадлежит к вызывающей процедуре, эта процедура может использовать ее значение – в данном случае автоматически напечатать его.

Резюме

В этой главе мы обсудили унификацию, поиск с возвратом, детерминизм, предикаты **not**, **fail** и **cut**, и рассмотрели предикаты Visual Prolog с процедурной точки зрения.

Факты и правила Пролога получают информацию при вызове с аргументами, которые могут быть константами или связанными переменными; они возвращают информацию в вызывающую процедуру путем связывания аргументов, которые являются несвязанными переменными.

Унификация – это процесс сопоставления двух предикатов и присваивания свободным переменным значений для того, чтобы сделать предикаты идентичными.

Этот механизм необходим, чтобы Visual Prolog мог определить, какое предложение вызвать и каким переменным присвоить значения. Представлены важные моменты, связанные с сопоставлением (унификацией):

когда Пролог начинает попытки достичь цели, он начинает поиск с начала программы;

когда вызов завершается успехом, говорят, что вызов возвратился, и делается попытка доказать следующую подцель;

если переменная была связана в предложении, единственный способ сделать ее снова свободной – это откат (поиск с возвратом).

Поиск с возвратом – это механизм, который указывает Visual Prolog, как искать решения для программы. Этот процесс дает Прологу возможность перебрать все известные факты и правила для решения. Рассмотрены четыре основных принципа поиска с возвратом:

- подцели должны проверяться по порядку, сверху вниз;
- предикатные предложения проверяются в том порядке, в котором они появляются в программе, сверху вниз;
- когда подцель сопоставляется с заголовком правила, тело правила должно после этого быть доказано (тело правила состоит из новых подцелей, которыеь должны быть доказаны);
- цель доказана, когда соответствующие факты найдены для каждой листьевой вершины дерева целей.

Вызов, который может дать множество решений – недетерминированный, тогда как вызов, дающий одно и только одно решение – детерминированный.

Visual Prolog дает три средства для управления направлением логического поиска в программе:

предикат ***fail*** всегда дает неуспех, он вызывает поиск с возвратом для того, чтобы искать другое решение;

предикат ***not*** дает успех, когда связанная с ним подцель не может быть доказана;

предикат ***cut*** отменяет поиск с возвратом.

Правила Пролога с процедурной точки зрения могут действовать как предложения *Case*, представляя собой булевы функции, или как оператор *GoTo* (при использовании *cut*). И возвращать вычисленные значения.

СПИСОК ЛИТЕРАТУРЫ

1. Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. – М.: Мир, 1982.
2. Alan Turing. Computing Machinery and Intelligence, – Mind, vol. LIX, *No*. 236, October 1950. – P. 433–460.
3. Дж. Сёрль. Разум мозга – компьютерная программа? // Ж. «В мире науки». – 1990. – № 3. – С 7–13.
4. Модальная логика: пер. с англ. / Р. Фейс – М.: Наука, 1974. – 520 с.
5. Мендельсон Э. Введение в математическую логику. – М.: Наука, 1976. – 322 с.
6. Логический подход к искусственному интеллекту: от модальной логики к логике баз данных: пер. с франц. / Тейз А., Грибмон П., Юлен Г. и др. – М.: Мир, 1998. – 494 с.
7. Логический подход к искусственному интеллекту: от классической логики к логическому программированию: пер. с франц. / Тейз А., Грибмон П., Луи Ж. и др. – М.: Мир, 1990. – 432 с.
8. Минц Г.Е., Тыугу Э.Х. Структурный синтез и неклассические логики. Применение методов математической логики. – Таллин, 1983.
9. Дейкстра Э. О структурной организации данных. – В кн.: Структурное программирование. – М.: Мир. – 1975.
10. Ульман Дж. Основы систем баз данных. – М.: Финансы и статистика. – 1983. – 334 с.
11. Адамченко А.Н., Кучумов А.М. Логическое программирование и Visual Prolog. – СПб.: БХВ-Петербург, 2003. – 992 с.

Учебное издание

НОВОСЕЛЬЦЕВ Виталий Борисович
КОПАНИЦА Георгий Дмитриевич

ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ И МОДЕЛИРОВАНИЕ СИСТЕМ

Учебное пособие

Издано в авторской редакции

Научный редактор
доктор физико-математических
наук, профессор *В.Б. Новосельцев*

Компьютерная верстка *К.С. Чечельницкая*
Дизайн обложки *О.Ю. Аршинова*

Подписано к печати 28.09.2011. Формат 60x84/16. Бумага «Снегурочка».
Печать XEROX. Усл. печ. л. 6,53. Уч.-изд. л. 5,89.
Заказ ___-11. Тираж 35 экз.

Национальный исследовательский Томский политехнический университет
Система менеджмента качества
Издательства Томского политехнического университета сертифицирована
NATIONAL QUALITY ASSURANCE по стандарту BS EN ISO 9001:2008

ИЗДАТЕЛЬСТВО ТПУ. 634050, г. Томск, пр. Ленина, 30
Тел./факс: 8(3822)56-35-35, www.tpu.ru