УДК 510.755
ББК 32.973
Д40

Д40 Майкл Дженесерет, Винай К. Чаудри

Введение в логическое программирование / пер. с англ. С. В. Минц – М.: ДМК Пресс, 2022. – 192 с.: ил.

ISBN 978-5-97060-968-2

Логическое программирование – это стиль программирования, в котором программы принимают форму наборов предложений на языке символической логики. В последнее время интерес к нему вырос благодаря возможности применения в дедуктивных базах данных, электронных таблицах, создании бизнес-логики при управлении предприятием и др.

Данная книга знакомит с теорией логического программирования, современными технологиями и популярными применениями. Авторы ведут читателя от изучения базовых понятий (наборы данных, запросы, обновления и т. д.) к практическому применению вычислительной логики. Книга удобно структурирована: рассмотрение новых терминов сопровождается многочисленными примерами; в конце глав приводятся упражнения, позволяющие закрепить пройденный материал.

Издание предназначено программистам различной квалификации, а также будет полезно студентам и всем желающим познакомиться с логическим программированием.

Original English language edition published by Morgan and Claypool publishers. All Rights Reserved Morgan and Claypool Publishers

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-68173-722-5 (англ.)
ISBN 978-5-97060-968-2 (рус.)

Copyright ©2020 Morgan and Claypool Publishers, 2020
© Оформление, перевод на русский язык, издание, ДМК Пресс, 2022

Оглавление

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв или оставить комментарий на странице книги **https://dmkpress.com/catalog/computer/programming/978-5-93700-968-2** в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Просканируйте код камерой смартфона и пройдите по ссылке

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства ДМК Пресс и Morgan & Claypool Publishers очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты **dmkpress@gmail.com**.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Отзывы

Эта книга XX в. представляет элегантный и инновационный взгляд на логическое программирование. В отличие от других книг наборы данных в ней представлены в качестве фундаментального понятия, устраняя разрыв между языками программирования и языками представления знаний. Обновления рассмотрены наравне с наборами данных, что приводит к разумному и практическому подходу к действиям и изменениям.

Боб Ковальски, почетный профессор Имперского колледжа Лондона (Bob Kowalski, Professor Emeritus, Imperial College London)

В мире, где глубокое обучение и Python являются темой дня, эта книга – замечательное достижение. Она знакомит читателя с основами традиционного логического программирования и разъясняет преимущества использования этой технологии для создания исполняемых спецификаций сложных систем.

Сон Цао Чан, профессор компьютерных наук университета штата Нью-Мексико (Son Cao Tran, Professor in Computer Science, New Mexico State University)

Отличное введение в основы логического программирования. Книга хорошо написана и структурирована. Концепции объясняются четко, и постепенно возрастающая сложность упражнений позволяет быстро освоить простые понятия прежде, чем переходить к более сложным идеям.

Джордж Янгер, студент Стэнфордского университета (George Younger, student, Stanford University)

Предисловие

Эта книга является учебником по логическому программированию. Она предназначена в первую очередь для использования на уровне бакалавриата. Однако ее можно использовать для мотивированных учащихся средней школы, а также в начале обучения в аспирантуре для тех, кто еще не знаком с этим материалом.

Есть только два предварительных условия. Предполагается, что учащийся знаком со множествами и операциями над ними, такими как объединение, пересечение и т. д. Также предполагается, что он владеет символической математикой на уровне алгебры средней школы или выше. Больше ничего не требуется.

Хотя опыт в области вычислительного мышления полезен, он не является обязательным. И предварительный опыт программирования не обязателен. Наоборот, мы заметили, что некоторые студенты, имеющие опыт программирования, поначалу испытывают больше трудностей, чем те, кто не является опытным программистом. Им, похоже, нужно отучиться от некоторых вещей, чтобы оценить силу и красоту логического программирования.

Применяемый здесь подход к логическому программированию возник в результате более чем 30-летних исследований, применения и преподавания этого материала в академической и коммерческой среде. Результатом этого является подход к предмету, который отличается от подхода, используемого в других книгах по этой теме, по двум основным направлениям.

Во-первых, используется модельно-теоретический подход к определению семантики, а не традиционный доказательно-теоретический. Он начинается с фундаментального понятия наборов данных, т. е. наборов основных атомов. Учитывая это фундаментальное понятие, мы определяем классические логические программы как наборы определений представлений, написанные с использованием традиционной нотации, подобной языку Prolog, но с семантикой, заданной в терминах наборов данных, а не реализации. (Мы также говорим о реализации, но об этом позже.)

Еще одним отличием от других книг по логическому программированию является то, что мы вводим фундаментальное понятие обновления, т. е. добавления и удаления основных атомов. Учитывая это фундаментальное понятие, мы представляем динамические логические программы как наборы определений действий, где действия рассматриваются как наборы одновременных обновлений. Это расширение позволяет нам говорить о логических агентах так же, как и о статических логи-

ческих программах. (Логический агент – это фактически машина состояний, в которой каждое состояние моделируется как набор данных, а каждое изменение моделируется как набор обновлений.)

В дополнение к тексту книги в печатном и электронном виде имеется веб-сайт с автоматически оцениваемыми онлайн-упражнениями, заданиями по программированию, инструментами логического программирования и примерами приложений. Веб-сайт **http://logicprogramming.stanford.edu** является бесплатным для использования и открыт для всех.

Просканируйте код камерой смартфона и пройдите по ссылке

В заключение мы хотим прежде всего поблагодарить двух людей, которые оказали глубокое влияние на нашу работу, – Джеффа Ульмана и Боба Ковальски. Джефф Ульман, наш коллега в Стэнфорде, вдохновил нас своими популярными учебниками и помог оценить глубокую связь между логическим программированием и базами данных. Боб Ковальски выслушал наши идеи, поддержал нашу работу и сотрудничал с нами в работе над некоторыми из представленных здесь материалов.

Мы также хотим отметить вклад бывшего аспиранта Абхиджита Мохапатра. Он является одним из изобретателей динамического логического программирования и создателей инструментов программирования, связанных с нашим подходом к логическому программированию. Он помогал преподавать курс, работал со студентами и вносил бесценные предложения по изложению и организации материала.

И наконец, мы благодарим студентов, которым пришлось выдержать ранние версии этого материала, во многих случаях помогая довести его до ума и проходя через эксперименты, которые не всегда были успешными. Свидетельством интеллекта этих студентов является то, что они усвоили материал, несмотря на многочисленные ошибки с нашей стороны. Их терпение и конструктивные комментарии были неоценимы и помогали нам понять, что работает, а что нет.

Майкл Дженесерет и Винай К. Чаудри

Часть I
Введение

Глава 1

Введение

1.1. Программирование в логике

Логическое программирование – это стиль программирования, в котором программы имеют форму наборов предложений на языке символической логики. Программы, написанные в этом стиле, называются *логическими программами*. Язык, на котором написаны эти программы, называется *языком логического программирования*. А компьютерная система, которая управляет созданием и выполнением логических программ, называется *системой логического программирования*.

1.2. Логические программы как исполняемые спецификации

Логическое программирование часто называют *декларативным* или *описательным* и противопоставляют его *императивному*, или *предписывающему*, подходу к программированию, связанному с традиционными языками программирования.

В императивном/предписывающем программировании программист предоставляет подробную рабочую программу для системы с точки зрения внутренних деталей обработки (таких как типы данных и назначение переменных). При написании таких программ программисты обычно принимают во внимание информацию о предполагаемых областях применения и целях программ, но эта информация редко записывается в текст программы, разве что в виде неисполняемых комментариев.

В декларативном/описательном программировании программисты явно кодируют информацию об области применения и целях программы, но не указывают внутренние детали ее обработки, оставляя за системами, выполняющими программу, право решать эти детали самостоятельно.

В качестве интуитивного примера этого различия рассмотрим задачу программирования робота для перемещения из одной точки здания

в другую. Типичная императивная программа предписывает роботу двигаться вперед на определенное расстояние (или пока его датчики не укажут на подходящий ориентир), затем роботу нужно повернуть и снова двигаться вперед и т. д., пока робот не прибудет в пункт назначения. В отличие от этого типичная декларативная программа состоит из карты и указания начальной и конечной точек на ней, а решение о том, как двигаться, оставлено на усмотрение робота.

Логическая программа является разновидностью декларативной программы, поскольку она описывает область применения программы и цели, которых программист хотел бы достичь. Она сосредоточена на том, *что* истинно и желаемо, а не на том, *как* достичь желаемых целей. В этом отношении логическая программа является скорее *спецификацией*, чем *реализацией*.

Логическое программирование практично, так как существуют хорошо известные методы выполнения логических программ и/или создания традиционных программ, достигающих тех же результатов. По этой причине логические программы иногда называют *исполняемыми спецификациями*.

1.3. Преимущества логического программирования

Логические программы обычно легче *создавать* и *модифицировать*, чем традиционные. Программисты могут обходиться малым количеством знаний о возможностях и ограничениях систем, выполняющих эти программы, или вообще без них. Кроме того, нет необходимости выбора конкретных методов достижения целей программ.

Логические программы лучше поддаются *компоновке*. При их написании программистам не нужно делать необдуманный выбор. В результате такие программы можно комбинировать друг с другом легче, чем традиционные, где произвольный выбор может порождать конфликты.

Логические программы также более *гибкие*, чем традиционные. Система, выполняющая логическую программу, может легко адаптироваться к неожиданным изменениям в своих предположениях и/или целях. Еще раз рассмотрим робота, описанного в предыдущем разделе. Если робот, выполняющий логическую программу, узнает, что коридор неожиданно закрыт, он может выбрать другой коридор. Если робота попросят забрать и доставить некоторые товары по пути, он может комбинировать маршруты, чтобы выполнить обе задачи без необходимости выполнять их по отдельности.

Наконец, логические программы более *универсальны*, чем традиционные программы, – их можно использовать для различных целей, часто

без модификации. Предположим, у нас есть таблица родителей и детей. Теперь представим, что нам даны определения для стандартных родственных отношений. Например, нам говорят, что дедушка или бабушка является родителем родителя. Это единственное определение может быть использовано в качестве основы для нескольких традиционных программ. (1) Мы можем использовать его для создания программы, которая вычисляет, является ли один человек дедушкой или бабушкой второго человека. (2) Мы можем использовать это определение, чтобы написать программы для вычисления бабушек и дедушек человека. (3) Мы можем использовать его для вычисления внуков данного человека. (4) И мы можем использовать его для составления таблицы бабушек, дедушек и внуков. В традиционном программировании мы бы написали разные программы для каждой из этих задач, и определение бабушки и дедушки не было бы *явно* закодировано ни в одной из этих программ. В логическом программировании определение может быть написано только один раз, и это определение может быть использовано для решения всех четырех задач.

В качестве другого примера (благодарим Джона Маккарти) рассмотрим тот факт, что, если два объекта сталкиваются, они обычно издают шум. Этот факт из жизни может быть использован при разработке программ для различных целей. (1) Если мы хотим разбудить кого-то, мы можем стукнуть два объекта друг о друга. (2) Если мы хотим никого не разбудить, мы будем следить за тем, чтобы предметы не сталкивались. (3) Если мы видим вдалеке две машины и слышим удар, можем сделать вывод, что они столкнулись. (4) Если видим, как две машины приближаются друг к другу, но ничего не слышим, можем предположить, что они не столкнулись.

1.4. Области применения логического программирования

Логическое программирование может быть плодотворно использовано практически в любой прикладной области. Однако оно имеет особую ценность в областях, характеризующихся большим количеством определений, ограничений и правил действий, особенно там, где эти определения, ограничения и правила поступают из множества источников или часто меняются. Далее перечислены несколько областей применения, в которых логическое программирование оказалось особенно полезным.

Базы данных. Представляя таблицы баз данных как наборы простых предложений, можно использовать логику для поддержки баз данных. Например, язык логики может быть использован для:

определения виртуальных представлений данных в терминах явно хранимых таблиц; кодирования ограничений баз данных; определения политик управления доступом; написания правил обновления.

Электронные таблицы / рабочие листы. Электронные таблицы (иногда называемые рабочими листами) обобщают, чтобы включить логические ограничения, а также традиционные арифметические формулы. Примеров таких ограничений множество. Например, в приложениях, связанных с составлением расписания, могут быть ограничения по времени или ограничения на то, кто может бронировать те или иные комнаты; в области бронирования билетов – ограничения на взрослых и младенцев; в академических программах – ограничения на количество курсов различных типов, которые должны пройти студенты.

Интегрирование данных. Язык логики может быть использован для связи понятий в различных словарях, что позволит получить доступ к многочисленным разнородным источникам данных, давая каждому пользователю иллюзию единой базы данных, закодированной в его собственном словаре.

Управление предприятием. Логическое программирование имеет особую ценность в реализации бизнес-правил различного рода. Внутренние бизнес-правила включают политики предприятия (например, утверждение расходов) и рабочий процесс (кто, что и когда делает). Внешние бизнес-правила включают детали контрактов с другими предприятиями, правила конфигурации и ценообразования для продуктов компании и т. д.

Вычислительное право. Вычислительное право – это отрасль информатики, занимающаяся представлением правил и положений в цифровой форме. Кодирование законов в цифровой форме позволяет проводить автоматизированный правовой анализ и создавать технологии, делающие этот анализ доступным для граждан, контролеров и правоприменителей, а также специалистов в области права.

Общие игры. Игроки общих игр – это системы, способные принимать описания произвольных игр во время их выполнения и использовать такие описания для успешной игры без вмешательства человека. Другими словами, они не знают правил до начала игры. Логическое программирование широко используется в общих игровых системах как предпочтительный способ формализации описаний игр.

1.5. Базовое логическое программирование

За многие годы были изучены различные виды логического программирования (базовое логическое программирование, классическое логическое программирование, транзакционное логическое программирование, логическое программирование ограничений, дизъюнктивное логическое программирование, программирование наборов ответов, индуктивное логическое программирование и т. д.). Одновременно были разработаны различные языки логического программирования (например, Datalog, Prolog, Epilog, Golog, Progol, LPS и т. д.). В данной книге мы сосредоточимся на базовом логическом программировании, варианте транзакционного логического программирования, и используем язык Epilog для написания примеров.

В базовом логическом программировании состояния приложения моделируются как наборы простых фактов (называемые *наборами данных*), и пишутся *правила* для определения абстрактных *представлений* фактов в наборах данных. Изменения состояния моделируются как *примитивные обновления* наборов данных, т. е. наборы добавлений и удалений фактов, и пишутся другие *правила* для определения *составных действий* в терминах примитивных обновлений.

Epilog (язык, который используется в этой книге) тесно связан с Datalog и Prolog. Их синтаксисы почти идентичны. И эти три языка хорошо упорядочены с точки зрения выразительных возможностей: Datalog является подмножеством Prolog, а Prolog – подмножеством Epilog. Для простоты мы используем синтаксис Epilog на протяжении всего курса и говорим об интерпретаторе и компиляторе Epilog. Таким образом, когда в дальнейшем упоминается Datalog, имеется в виду подмножество Datalog в Epilog, и когда упоминается Prolog, имеется в виду подмножество Prolog в Epilog.

Как мы увидим, все три этих языка (Datalog, Prolog и Epilog) менее выразительны, чем языки, связанные с более сложными формами логического программирования (такими как дизъюнктивное логическое программирование и программирование наборов ответов). Это ограничивает то, что мы можем формулировать, однако получаемые программы лучше с точки зрения вычислительных возможностей и в большинстве случаев более практичны, чем программы, написанные на более выразительных языках. Более того, благодаря этим ограничениям Datalog, Prolog и Epilog просты для понимания и, следовательно, имеют педагогическую ценность как введение в более сложные языки логического программирования.

В соответствии с нашим акцентом на базовое логическое программирование материал курса разделен на пять частей. В части I дан обзор основ логического программирования, а также представление о *наборах*

данных, в части II мы поговорим о *запросах* и *обновлениях*, в части III – об *определениях представлений*, в части VI сосредоточимся на *определениях операций*. И в части V рассмотрим *вариации*, т. е. другие формы логического программирования.

Исторические заметки

В середине 50-х гг. прошлого века ученые-компьютерщики начали концентрироваться на разработке высокоуровневых языков программирования. В качестве вклада в эти усилия Джон Маккарти предложил язык символической логики и сформулировал идеал декларативного программирования. Он изложил эти идеи в основополагающей работе, опубликованной в 1958 г., где описывается тип системы, которую он назвал *обучаемой системой*.

> «Главное преимущество, которое мы ожидаем от обучаемой системы, заключается в том, что ее функционирование можно будет улучшить, просто рассказывая ей об ее <…> окружении и о том, чего от нее хотят. Для того чтобы это сделать, от системы потребуется мало предварительных знаний, а может быть, и вообще никаких».

Идея декларативного программирования привлекла внимание последующих исследователей, в частности Боба Ковальски, одного из отцов логического программирования, и Эда Фейгенбаума, изобретателя инженерии знаний. В статье, написанной в 1974 г., Фейгенбаум следующим образом переформулировал идеал Маккарти:

> «Потенциальное использование людьми компьютеров для выполнения задач может быть представлено в виде спектра инструкций, которые необходимо дать компьютеру, чтобы он выполнил свою работу. Назовем его спектром "что–как". На одном конце спектра пользователь употребляет свой интеллект, чтобы проинструктировать машину в точности, как именно выполнять свою работу шаг за шагом. <…> На другом конце спектра находится пользователь со своей реальной проблемой. <…> Он стремится передать то, что он хочет получить <…> без необходимости подробно излагать все требуемые для этого действия машины».

Развитие логического программирования в его нынешней форме можно проследить по дебатам о декларативном и процедурном представлении знаний в сообществе искусственного интеллекта.

Сторонники процедурных представлений во главе с Марвином Мински и Сеймуром Пейпертом были в основном сосредоточены в Массачу-

сетском технологическом институте. Язык Planner, разработанный там же, стал первым языком, возникшим в рамках процедурной парадигмы, хотя он был основан на логических методах доказательства. В Planner был реализован шаблонный вызов процедурных планов из целей (т. е. редукция целей или обратная цепочка) и утверждений (т. е. прямая цепочка). Наиболее значимой реализацией Planner было подмножество Planner, называемое Micro-Planner, созданное Джерри Сассманом, Юджином Чарниаком и Терри Виноградом. Оно было применено в разработанной Виноградом программе понимания естественного языка SHRDLU, что было знаковым событием в то время.

Сторонники декларативных представлений были сосредоточены в Стэнфорде (связанные с Джоном Маккарти, Бертрамом Рафаэлем и Корделлом Грином) и в Эдинбурге (представленные Джоном Аланом Робинсоном, Пэтом Хейсом и Робертом Ковальски). Хейс и Ковальски пытались примирить основанный на логике декларативный подход к представлению знаний с процедурным подходом языка Planner. В 1973 г. Хейс разработал эквивалентный язык Golux, в котором различные процедуры можно было получить путем изменения процедуры доказательства теорем. Ковальски, с другой стороны, разработал SLD-решение, вариант SL-решения, и показал, как он рассматривает импликации как процедуры редукции целей. Ковальски сотрудничал с Колмерауэром в Марселе, который развил эти идеи при разработке языка программирования Prolog, реализованного летом и осенью 1972 г. Первой программой на языке Prolog, также написанной в 1972 г. и примененной в Марселе, была французская система ответов на вопросы. Использование Prolog в качестве практического языка программирования получило большой импульс после разработки компилятора Дэвидом Уорреном в Эдинбурге в 1977 г.

Глава 2

Наборы данных

2.1. Введение

Наборы данных – это коллекции фактов о каких-либо аспектах жизни. Они могут использоваться как сами по себе для кодирования информации, так и в сочетании с логическими программами для формирования более сложных информационных систем, как будет показано в последующих главах.

Мы начинаем эту главу с разговора о составлении набора представлений о нашем мире. Затем представим формальный язык для кодирования информации о наших представлениях в виде наборов данных и приведем несколько примеров наборов данных, закодированных в этом языке. И наконец, обсудим вопросы, связанные с переосмыслением области применения этого языка и кодированием представлений в виде наборов данных с различными словарями.

2.2. Формирование представлений

Когда мы думаем о мире, мы обычно думаем в терминах объектов и отношений между ними. К *объектам* относятся такие вещи, как люди, учреждения и здания. *Отношения* включают, например, отцовство, дружбу, назначение на должность, расположение офиса и т. д. Одним из способов представления такой информации являются графы. В качестве примера рассмотрим граф, показанный ниже. Узлы здесь представляют объекты, а стрелки – отношения между ними.

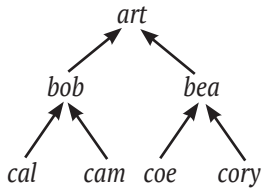

Рис. 2.1. Представление информации графом

В качестве альтернативы мы можем представить такую информацию в виде таблиц, например закодировать информацию из графа в виде следующей таблицы.

Таблица 2.1. Представление информации таблицей

Родитель	
art	bob
art	bea
bob	cal
bob	cam
bea	coe
bea	cory

Другой возможностью является кодирование отдельных отношений в виде предложений на формальном языке. Например, мы можем представить нашу информацию о родстве так, как показано ниже. Здесь каждый факт принимает форму предложения, состоящего из названия отношения и имен вовлеченных в него объектов.

```
parent(art,bob)
parent(art,bea)
parent(bob,cal)
parent(bob,cam)
parent(bea,coe)
parent(bea,cory)
```

Хотя графы и таблицы интуитивно привлекательны, смысловое представление является более полезным для наших целей. Поэтому далее мы будем представлять факты в виде предложений, а различные состояния мира – в виде различных наборов таких предложений.

И последнее замечание. В дальнейшем мы будем использовать слова «*связь*» и «*отношение*» как взаимозаменяемые. С математической точки зрения это не совсем корректно, поскольку между этими двумя понятиями существует тонкое различие. Однако для наших целей эта разница несущественна, и часто проще сказать «*связь*», чем «*отношение*».

2.3. Наборы данных

Набор данных – это коллекция простых фактов, которые характеризуют состояние области их применения. Факты, включенные в набор данных, считаются истинными, не включенные – ложными. Различные наборы данных характеризуют различные состояния.

Константы – это строки строчных букв, цифр, знаков подчеркивания и точек *или* произвольных символов ASCII, заключенных в двойные кавычки. По причинам, описанным в следующей главе, запрещены строки, содержащие заглавные буквы, кроме как в двойных кавычках. Примерами констант являются `a`, `b`, `comp225`, `123`, `3.14159`, `barack_obama` и `"Mind your p's and q's!"`. Константами не являются `Art` (содержит заглавную букву), p&q (содержит амперсанд), `the-house-that-jack-built` (содержит дефис). *Словарь* – это набор констант.

Далее мы будем различать три типа констант. *Символы* предназначены для обозначения объектов, *конструкторы* используются для создания составных имен объектов, *предикаты* представляют отношения между объектами.

Каждый конструктор и предикат имеет определенную *арность*, т. е. количество аргументов, допустимое в любом выражении, включающем конструктор или предикат. *Унарные* конструкторы и предикаты принимают только один аргумент, *бинарные* – два аргумента, а *тернарные* принимают три аргумента. Кроме того, мы часто говорим, что конструкторы и предикаты являются *n-арными*. Обратите внимание, что возможно существование предиката без аргументов, представляющего собой условие, которое просто истинно или ложно.

Основной элемент – это либо символ, либо составное имя. *Составное имя* – это выражение, сформированное из *n*-арного конструктора и *n* основных элементов, заключенных в круглые скобки и разделенных запятыми. Если `a` и `b` – символы, а `pair` – бинарный конструктор, то `pair(a,a)`, `pair(a,b)`, `pair(b,a)` и `pair(b,b)` – составные имена. Прилагательное *основной* здесь означает, что элемент не содержит *переменных* (которые мы обсудим в следующей главе).

Вселенная Гербранда для словаря – это множество всех основных элементов, которые могут быть образованы из символов и конструкторов словаря. Для конечного словаря без конструкторов вселенная Гербранда конечна (т. е. состоит только из символов). Для конечного словаря с конструкторами она бесконечна (т. е. содержит все символы и составные имена, которые могут быть образованы из этих символов). Вселенная Гербранда для словаря, описанного в предыдущем параграфе, показана ниже.

```
{pair(a,b), pair(a,pair(b,c)), pair(a,pair(b,pair(c,d))), ...}
```

Элемент данных / фактоид / факт – это выражение, образованное из *n*-арного предиката и *n* основных элементов, заключенных в круглые скобки и разделенных запятыми. Например, если `r` – бинарный предикат, `a` и `b` – символы, то `r(a,b)` – это элемент данных.

База Гербранда для словаря – это множество всех фактов, которые могут быть образованы из констант словаря. Например, для словаря, со-

стоящего из двух символов `a` и `b` и одного бинарного предиката `r`, база Гербранда показана ниже.

```
{r(a,a), r(a,b), r(b,a), r(b,b)}
```

Наконец, мы определяем *набор данных* как любое подмножество базы Гербранда, т. е. произвольный набор фактов, который может быть сформирован из словаря базы данных. Интуитивно мы рассматриваем данные в наборе данных как факты, которые считаем истинными; данные, которых нет в наборе данных, считаются ложными.

2.4. Пример – женское сообщество

Рассмотрим межличностные отношения в небольшом женском сообществе. В нем всего четыре участницы – Эбби, Бесс, Коди и Дана. Некоторые из девушек нравятся друг другу, а некоторые – нет.

На рис. 2.2 показан один из возможных вариантов. Галочка в первом ряду означает, что Эбби нравится Коди, а отсутствие галочки означает, что Эбби не нравятся другие девочки (включая саму ее). Бесс тоже нравится Коди, Коди нравятся все, кроме нее самой. И Дане также нравится Коди.

	Abby	Bess	Cody	Dana
Abby			✓	
Bess			✓	
Cody	✓	✓		✓
Dana			✓	

Рис. 2.2. Состояние женского сообщества

Чтобы закодировать эту информацию в виде набора данных, используем словарь с четырьмя символами (`abby`, `bess`, `cody`, `dana`) и один бинарный предикат `likes`. Используя этот словарь, можно закодировать информацию рис. 2.1, записав набор данных, показанный ниже.

```
likes(abby,cody)
likes(bess,cody)
likes(cody,abby)
likes(cody,bess)
likes(cody,dana)
likes(dana,cody)
```

Обратите внимание, что отношение `likes` не имеет никаких внутренних ограничений. Возможно, что одному человеку нравится второй,

при этом второму не нравится первый. Можно, чтобы человеку нравился только один человек, или много людей, или никто. Возможно, что всем нравятся все или никто не нравится никому.

Даже для такого маленького мира, как этот, существует множество возможных вариантов развития событий. Для четырех девушек есть 16 возможных вариантов отношения – `likes(abby,abby)`, `likes(abby,bess)`, `likes(abby,cody)`, `likes(abby,dana)`, `likes(bess,abby)` и т. д. Каждый из этих 16 вариантов может быть либо истинным, либо ложным. Существует 2^{16} (т. е. 65 536) возможных комбинаций этих истинных и ложных возможностей, и поэтому возможны 2^{16} состояний этого мира и, следовательно, 2^{16} возможных наборов данных.

2.5. Пример – родство

В качестве другого примера рассмотрим небольшой набор данных о родственных связях. Элементы в этом случае снова представляют людей. Предикаты называют свойства этих людей и их отношения друг с другом. В примере используется бинарный предикат `parent`, чтобы указать, что один человек является родителем другого. Приведенные ниже предложения представляют собой набор данных, описывающий шесть случаев родительского отношения. Человек по имени Арт является родителем человека по имени Боб и человека по имени Беа; Боб является родителем Кэла и Кэма; а Беа – родителем Ко и Кори.

```
parent(art,bob)
parent(art,bea)
parent(bob,cal)
parent(bob,cam)
parent(bea,coe)
parent(bea,cory)
```

Предикат `adult` (взрослый) – это унарное отношение, т. е. простое свойство человека, а не отношение с другими людьми. В приведенном ниже наборе данных все являются взрослыми, за исключением внуков Арта (см. предложения выше).

```
adult(art)
adult(bob)
adult(bea)
```

Пол можно выразить с помощью двух унарных предикатов `male` (мужчина) и `female` (женщина). Следующие данные показывают пол всех людей в нашем наборе данных. Обратите внимание, что, в принципе, можно использовать только одно отношение, поскольку один пол является

дополнением другого. Однако наше представление позволяет нам одинаково эффективно перечислять экземпляры обоих полов, что может быть полезно в некоторых приложениях.

```
male(art)      female(bea)
male(bob)      female(coe)
male(cal)      female(cory)
male(cam)
```

В качестве примера тернарного отношения рассмотрим данные, показанные ниже. Здесь мы используем `prefers` для обозначения того факта, что первому человеку нравится второй человек больше, чем третий. Например, в первом предложении говорится, что Арт предпочитает Беа Бобу; во втором предложении говорится, что Боб предпочитает Кэла Кэму.

```
prefers(art,bea,bob)
prefers(bob,cal,cam)
```

Обратите внимание, что порядок аргументов в таких предложениях произвольный. Учитывая значение `prefers` в нашем примере, первый аргумент обозначает субъекта, второй аргумент – это лицо, которому отдается предпочтение, а третий аргумент – лицо, которому отдается меньшее предпочтение. С таким же успехом мы могли бы интерпретировать аргументы в другом порядке. Важным является постоянство – если мы решили интерпретировать аргументы одним способом, мы должны придерживаться этой интерпретации везде.

Одно примечательное различие между этим и предыдущим примерами заключается в том, что в предыдущем есть только одно отношение (т. е. отношение `likes`), в то время как во втором есть несколько отношений (три унарных предиката, один бинарный и один тернарный).

Более тонкое и интересное различие заключается в том, что отношения в параграфе 2.5 ограничены различным образом, в то время как отношение `likes` в параграфе 2.4 не ограничено. В нем любому человеку может нравиться любой другой человек, возможны любые комбинации симпатий и антипатий. В отличие от этого в параграфе 2.5, например, человек не может быть своим собственным родителем, человек не может быть одновременно мужчиной и женщиной.

2.6. Пример – мир блоков

Мир блоков – это популярная прикладная область для иллюстрации идей в области искусственного интеллекта. Типичная сцена этого мира показана на рис. 2.3.

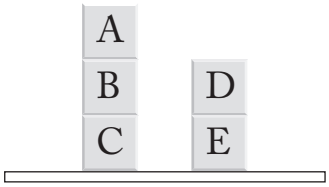

Рис. 2.3. Состояние мира блоков

Большинство людей, смотрящих на рис. 2.3, интерпретируют его как конфигурацию из пяти игрушечных кубиков. Некоторые воспринимают стол, на котором лежат блоки, как объект, но для простоты мы его игнорируем.

Для описания этой сцены мы используем словарь с пятью символами (`a`, `b`, `c`, `d`, `e`), по одному символу для каждого из пяти блоков. Каждый из этих символов символизирует блок, обозначенный соответствующей заглавной буквой на рисунке.

В пространственном представлении мира блоков существует множество значимых отношений. Например, имеет смысл говорить об отношении, которое существует между двумя блоками, если один и только один из них опирается на другой. В дальнейшем будем использовать предикат `on` для обозначения этого отношения. Также можно говорить об отношении, которое существует между двумя блоками тогда и только тогда, когда один из них находится где-либо над другим, т. е. первый опирается на второй или опирается на блок, который опирается на второй, и т. д. Говоря об этом отношении, в дальнейшем используем предикат `above`. Существует отношение, которое имеет место для трех блоков, уложенных друг на друга. Используем предикат `stack` для обозначения этого отношения. Применим предикат `clear` для обозначения отношения, которое имеет место для блока тогда и только тогда, когда на нем нет блока. Мы используем предикат `table` для обозначения отношения, которое имеет место для блока тогда и только тогда, когда этот блок покоится на столе.

Арность этих предикатов определяется их назначением. Поскольку предикат `on` служит для обозначения отношения между двумя блоками, он имеет арность 2. Аналогично предикат `above` имеет арность 2. Предикат `stack` имеет арность 3. Предикаты `clear` и `table` имеют арность 1.

Учитывая этот словарь, опишем сцену на рис. 2.3, составив предложения, которые указывают, какие отношения существуют между объектами или группами объектов. Начнем с `on`. Следующие предложения говорят о том, истинно или ложно каждое предложение, основанное на отношениях.

```
on(a,b)
on(b,c)
on(d,e)
```

Существует четыре факта above. Отношение above содержит те же пары блоков, что и отношение on, но оно включает один дополнительный факт для блоков a и c.

```
above(a,b)
above(b,c)
above(a,c)
above(d,e)
```

Аналогичным образом можно закодировать отношение стека и отношение above. Здесь есть только один стек – блок a на блоке b и блок b на блоке c.

```
stack(a,b,c)
```

Наконец, можем записать факты для clear и table.

```
clear(a)      table(c)
clear(d)      table(e)
```

Как и в примере с родством, отношения в мире блоков имеют различные ограничения. Например, блок не может быть сам на себе (т. е. недопустимо, например, on(a,a)). Более того, некоторые из этих отношений полностью определяются другими. Например, учитывая отношение on, факты обо всех других отношениях полностью определены. В следующей главе мы рассмотрим, как написать определения для таких концепций и тем самым избежать необходимости записывать отдельные факты для таких определенных понятий.

2.7. Пример – мир еды

В качестве еще одного примера этих концепций рассмотрим небольшой набор данных о еде и меню. Цель состоит в том, чтобы создать набор данных, содержащий список блюд, которые можно заказать в ресторане в разные дни недели.

Символы в данном случае бывают двух типов: дни недели – понедельник, … , пятница (monday, … , friday) – и различные типы блюд (кальмары, вишисуаз[1], говядина и т. д.).

Есть три конструктора: тернарный конструктор для меню из трех блюд (three), четырехарный конструктор для меню из четырех блюд (four) и пятиарный конструктор для меню из пяти блюд (five). Существует единственный бинарный предикат menu, который связывает дни недели и доступные блюда.

[1] Вишисуаз – густой суп, приготовленный из вареного и протертого лука-порея, лука, картофеля, сливок и куриного бульона. – *Прим. перев.*

Ниже приведен пример набора данных с использованием этого словаря. В понедельник ресторан предлагает меню из трех блюд – кальмаров (`calamari`), говядины (`beef`) и коржика (`shortcake`), а также другое меню из трех блюд – пюре (`puree`), говядины и мороженого (`ice cream`) на десерт. Во вторник предлагается одно из тех же меню из трех блюд, а также меню из четырех блюд – консоме (`consomme`), зелени (`green`), ягненка (`lamb`), пахлавы (`baklava`). В среду ресторан предлагает только одно меню из четырех блюд, аналогичное приготовленному накануне. В четверг предлагается меню из пяти блюд – вишисуаз (`vichyssoise`), цезаря (`caesar`), форели (`trout`), курицы (`chicken`), тирамису (`tiramisu`), а в пятницу – другое меню из пяти блюд (в том числе суфле (`souffle`)).

```
menu(monday,three(calamari,beef,shortcake))
menu(monday,three(puree,beef,icecream))
menu(tuesday,three(puree,beef,icecream))
menu(tuesday,four(consomme,green,lamb,baklava))
menu(wednesday,four(consomme,green,lamb,baklava))
menu(thursday,five(vichyssoise,caesar,trout,chicken,tiramisu))
menu(friday,five(vichyssoise,green,trout,beef,souffle))
```

Обратите внимание, что, хотя здесь есть конструкторы, набор данных имеет конечный размер. На самом деле существуют сильные ограничения на то, какие предложения имеют смысл. Например, только символы, обозначающие дни недели, появляются в качестве первого аргумента отношения `menu`. Только символы, представляющие продукты питания, появляются в качестве аргументов в составных именах. И только целые числа выступают в качестве второго аргумента отношения `menu`. Обратите внимание, что составные имена здесь не являются вложенными. Эти ограничения часто встречаются в наборах данных. Позже мы покажем, как можно формализовать такие ограничения.

2.8. Переформулирование

Независимо от того, как мы выбираем концептуализацию (представление) мира, важно понимать, что существуют и другие концептуализации. Более того, не обязательно должно существовать соответствие между объектами, функциями и отношениями в одной концептуализации и объектами, функциями и отношениями в другой.

В некоторых случаях изменение концептуализации мира может сделать невозможным выразить определенные виды знаний. Известным примером этого является полемика в физике между взглядом на свет как волновое явление и взглядом на него в терминах частиц. Каждая концепция позволяла физикам объяснять различные аспекты поведения света, но ни одна из них сама по себе не была достаточной. Проти-

воречия были устранены только после того, как эти два взгляда были объединены в современной квантовой физике.

В других случаях изменение концептуализации может усложнить выражение знания, не обязательно делая его невозможным. Хорошим примером этого – опять же в физике – является изменение точки зрения. Опираясь на геоцентрическую точку зрения Аристотеля на Вселенную, астрономы испытывали большие трудности при объяснении движения Луны и планет. Данные в рамках аристотелевской концепции объяснения (с помощью эпициклов и т. д.) были чрезвычайно громоздки. Переход к гелиоцентрическому взгляду быстро привел к более понятной теории.

В связи с этим возникает вопрос, что делает одну концептуализацию более подходящей, чем другая. В настоящее время исчерпывающего ответа на этот вопрос не существует. Однако есть несколько вопросов, которые заслуживают особого внимания.

Одним из таких вопросов является *степень детализации,* связанная с той или иной концептуализацией. Выбор слишком высокой степени детализации может сделать формализацию знаний непомерно утомительной. Выбор слишком низкой может сделать ее невозможной.

В качестве примера этой проблемы рассмотрим концептуализацию сцены в мире блоков, в которой объектами изучения являются атомы, из которых состоят блоки на картинке. Каждый блок состоит из огромного количества атомов, поэтому область исследования чрезвычайно велика. Хотя, в принципе, возможно описать сцену на таком уровне детализации, это бессмысленно, если нас интересует только вертикальное соотношение блоков, состоящих из этих атомов. Конечно, для химика, интересующегося составом блоков, атомный взгляд на сцену может быть более подходящим, а наша концептуализация в терминах блоков имеет слишком малую степень детализации.

Абстракция неразличимости – это форма переформулирования объектов, которая имеет дело со степенью детализации. Если несколько объектов набора данных удовлетворяют всем одинаковым условиям, при соответствующих обстоятельствах можно рассматривать эти объекты как один. Это может снизить стоимость обработки запросов за счет отмены избыточных вычислений.

Другим способом переосмысления мира является представление отношений как объектов в области изучения. Преимущество этого способа заключается в том, что он позволяет нам рассматривать свойства свойств.

В качестве примера рассмотрим представление мира блоков, в котором есть пять блоков, три унарных предиката, каждый из которых соответствует своему цвету, и нет конструкторов. Такая концептуализация позволяет нам рассматривать цвета блоков, но не свойства этих цветов.

Мы можем исправить этот недостаток, *переопределив* различные цветовые отношения как объекты сами по себе и добавив отношение, связывающее блоки с цветами. Так как цвета являются объектами, мы можем добавить отношения, которые характеризуют их, например теплый, холодный и т. д.

Существует и обратное действие – переход от объектов к отношениям – *релятивизация*. Сочетание релятивизации и переопределения – это обычный способ перехода от одной концептуализации к другой.

Обратите внимание, что в этом обсуждении не было уделено никакого внимания вопросу о том, существуют ли реально объекты концептуализации мира, принятой человеком. Мы не приняли ни точку зрения реализма, который утверждает, что объекты в концептуализации человека реально существуют, ни номинализма, утверждающего, что представления человека не имеют внешнего существования. Концептуализации – это наши изобретения, и их оправдание основывается исключительно на их полезности. Это отсутствие приверженности указывает на существенную онтологическую неразборчивость логического программирования: любые концептуализации мира приемлемы, и мы ищем те из них, которые полезны для наших целей.

2.9. Упражнения

2.1. Рассмотрите представленный выше пример – женское сообщество. Выпишите набор данных, описывающий состояние, в котором каждая девушка любит только себя и никого больше.

2.2. Рассмотрите вариант примера – женское сообщество, в котором есть единственное бинарное отношение, называемое `friend`. `friend` отличается от `likes` двумя особенностями. Оно нерефлексивно, т. е. девушка не может дружить сама с собой; и оно симметрично, т. е. если одна девушка дружит со второй, то вторая девушка дружит с первой. Выпишите набор данных, описывающий состояние, которое точно удовлетворяет нерефлексивности и симметричности отношения `friend`, причем такой, в котором верны ровно шесть фактов `friend`. Обратите внимание, что это можно сделать несколькими способами.

2.3. Рассмотрите вариант примера – женское сообщество, в котором есть единственное бинарное отношение `younger`. Оно отличается от `likes` по трем признакам. Оно нерефлексивно, т. е. девушка не может быть моложе самой себя. Оно антисимметрично, т. е. если одна девушка моложе, чем вторая, то вторая не моложе первой. Оно является транзитивным, т. е. если одна девушка младше второй, а вторая младше третьей, то первая младше третьей. Выпишите набор данных, описывающий состояние, которое удовлетво-

ряет нерефлексивности, антисимметричности и транзитивности отношения `younger`, причем такой, что максимальное количество фактов `younger` истинно. Обратите внимание, что это можно сделать несколькими способами.

2.4. Человек `x` является `sibling` для человека `y` тогда и только тогда, когда `x` является братом или сестрой `y`. Запишите факты `sibling`, соответствующие фактам о родителях, показанным ниже.

```
parent(art,bob)
parent(art,bea)
parent(bob,cal)
parent(bob,cam)
parent(bea,coe)
parent(bea,cory)
```

2.5. Рассмотрите состояние мира блоков, изображенное на рис. 2.4. Выпишите все факты `above`, которые истинны в этом состоянии.

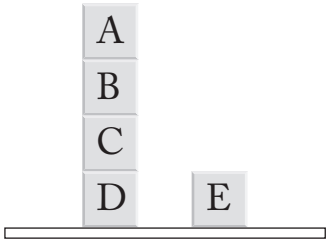

Рис. 2.4. Состояние мира блоков для упражнения

2.6. Рассмотрите мир с n символами и одним бинарным предикатом. Сколько различных фактов может быть записано на этом языке?

$$n, 2n, n^2, 2^n, n^n, 2^{n^2}, 2^{2^n}$$

2.7. Рассмотрите мир с n символами и одним бинарным предикатом. Сколько различных наборов данных возможно для этого языка?

$$n, 2n, n^2, 2^n, n^n, 2^{n^2}, 2^{2^n}$$

2.8. Рассмотрите мир с n символами и одним бинарным предикатом; предположите, что бинарное отношение является функциональным, т. е. каждый символ в первой позиции сопряжен с точно одним символом во второй позиции. Сколько различных наборов данных удовлетворяет этому ограничению?

$$n, 2n, n^2, 2^n, n^n, 2^{n^2}, 2^{2^n}$$